Laravel 10
官方文件

核心架構

請求的生命週期 | 服務容器 | 服務提供者 | Facades

基礎功能

路由 | 中介軟體 | CSRF 保護 | Controller | HTTP 請求 | HTTP 響應 | 檢視 | Blade 範本 | Vite 編譯 Assets | 生成 URL | Session | 表單驗證 | 錯誤處理 | 日誌

繼續深入

Artisan 命令列 | 廣播 | 快取系統 | 集合 | 契約 Contract | 事件系統 | 檔案儲存 | HTTP Client | 本地化 | 郵件 | 消息通知 | Package 開發 | 處理程序管理 | 佇列 | 限流 | 任務調度

目錄

1 請求的生命週期....1
1.1 簡介....1
1.2 生命週期概述....1
1.3 關注服務提供者....2
2 服務容器....3
2.1 簡介....3
2.2 繫結....4
2.3 解析....8
2.4 方法呼叫和注入....9
2.5 容器事件....10
2.6 PSR-11....10
3 服務提供者....11
3.1 簡介....11
3.2 編寫服務提供者....11
3.3 註冊服務提供者....13
3.4 延遲載入提供者....13
4 Facades....15
4.1 簡介....15
4.2 何時使用 Facades....15
4.3 Facades 工作原理....17
4.4 即時 Facades....17
5 路由....20
5.1 基本路由....20
5.2 路由參數....22
5.3 命名路由....24
5.4 路由組....25
5.5 路由模型繫結....26
5.6 Fallback 路由....30
5.7 速率限制....30
5.8 偽造表單方法....32
5.9 訪問當前路由....32
5.10 跨域資源共享 (CORS)....32
5.11 路由快取....33
6 中介軟體....34
6.1 介紹....34
6.2 定義中介軟體....34
6.3 註冊中介軟體....35
6.4 中介軟體參數....38
6.5 可終止的中介軟體....38
7 CSRF 保護....40
7.1 簡介....40
7.2 阻止 CSRF 請求....40
7.3 X-CSRF-TOKEN....41
7.4 X-XSRF-TOKEN....42
8 Controller....43
8.1 介紹....43
8.2 編寫 controller....43
8.3 controller 中介軟體....44
8.4 資源型 controller....45

8.5 依賴注入和 controller....50
9 HTTP 請求....52
9.1 介紹....52
9.2 與請求互動....52
9.3 輸入....55
9.4 輸入過濾和規範化....59
9.5 檔案....60
9.6 組態受信任的代理....61
9.7 組態可信任的 Host....62
10 HTTP 響應....63
10.1 建立響應....63
10.2 重新導向....65
10.3 其他響應類型....67
10.4 響應宏....68
11 檢視....69
11.1 介紹....69
11.2 建立和渲染檢視....69
11.3 向檢視傳遞資料....70
11.4 查看合成器....71
11.5 最佳化檢視....73
12 Blade 範本....74
12.1 簡介....74
12.2 顯示資料....74
12.3 Blade 指令....76
12.4 元件....82
12.5 建構佈局....93
12.6 表單....95
12.7 堆疊....96
12.8 服務注入....96
12.9 渲染內聯 Blade 範本....97
12.10 渲染 Blade 片段....97
12.11 擴展 Blade....97
13 Vite 編譯 Assets....100
13.1 介紹....100
13.2 安裝和設定....100
13.3 運行 Vite....102
13.4 使用 JavaScript....103
13.5 使用樣式表....105
13.6 使用 Blade 和 路由....105
13.7 Script & Style 標籤的屬性....108
13.8 高級定製....110
14 生成 URL....112
14.1 簡介....112
14.2 基礎....112
14.3 命名路由的 URLs....112
14.4 controller 行為的 URL....114
14.5 預設值....114
15 HTTP Session....116
15.1 簡介....116
15.2 使用 Session....117
15.3 Session 阻塞....119

15.4 新增自訂 Session 驅動....120
16 表單驗證....122
16.1 簡介....122
16.2 快速開始....122
16.3 表單請求驗證....126
16.4 手動建立驗證器....129
16.5 處理驗證欄位....132
16.6 使用錯誤消息....132
16.7 可用的驗證規則....134
16.8 有條件新增規則....134
16.9 驗證陣列....136
16.10 驗證檔案....138
16.11 驗證密碼....138
16.12 自訂驗證規則....140
17 錯誤處理....143
17.1 介紹....143
17.2 組態....143
17.3 異常處理....143
17.4 HTTP 異常....147
18 日誌....149
18.1 介紹....149
18.2 組態....149
18.3 建構日誌堆疊....151
18.4 寫入日誌消息....151
18.5 Monolog 通道定製....154
19 Artisan 命令列....157
19.1 介紹....157
19.2 編寫命令....158
19.3 定義輸入期望....160
19.4 命令 I/O....162
19.5 註冊命令....165
19.6 以程式設計方式執行命令....165
19.7 訊號處理....167
19.8 Stub 定製....167
19.9 事件....167
20 廣播....168
20.1 介紹....168
20.2 伺服器端安裝....168
20.3 客戶端安裝....170
20.4 概念概述....171
20.5 定義廣播事件....173
20.6 授權頻道....176
20.7 廣播事件....178
20.8 接收廣播....180
20.9 存在頻道....181
20.10 模型廣播....182
20.11 客戶端事件....185
20.12 通知....186
21 快取系統....187
21.1 簡介....187
21.2 組態....187

21.3 快取用法....188
21.4 快取標籤....191
21.5 原子鎖....192
21.6 新增自訂快取驅動....193
21.7 事件....194
22 集合....196
22.1 介紹....196
22.2 可用的方法....197
22.3 Higher Order Messages....197
22.4 惰性集合....197
23 契約 Contract....200
23.1 簡介....200
23.2 何時使用 Contract....200
23.3 如何使用 Contract....200
23.4 Contract 參考....201
24 事件系統....202
24.1 介紹....202
24.2 註冊事件和監聽器....202
24.3 定義事件....205
24.4 定義監聽器....205
24.5 佇列事件監聽器....206
24.6 調度事件....209
24.7 事件訂閱者....210
24.8 測試....212
25 檔案儲存....215
25.1 簡介....215
25.2 組態....215
25.3 獲取磁碟實例....219
25.4 檢索檔案....219
25.5 保存檔案....222
25.6 刪除檔案....225
25.7 目錄....225
25.8 測試....226
25.9 自訂檔案系統....227
26 HTTP Client....228
26.1 簡介....228
26.2 建立請求....228
26.3 並行請求....233
26.4 宏....234
26.5 測試....234
26.6 事件....237
27 本地化....238
27.1 簡介....238
27.2 定義翻譯字串....239
27.3 檢索翻譯字串....240
27.4 覆蓋擴展包的語言檔案....241
28 郵件....243
28.1 介紹....243
28.2 生成 Mailables....245
28.3 編寫 Mailables....245
28.4 Markdown 格式郵件....252

28.5 傳送郵件....254
28.6 渲染郵件....256
28.7 本地化郵件....257
28.8 測試郵件....258
28.9 郵件和本地開發....260
28.10 事件....260
28.11 自訂傳輸....261
29 消息通知....264
29.1 介紹....264
29.2 建立通知....264
29.3 傳送通知....264
29.4 郵件通知....268
29.5 Markdown 郵件通知....274
29.6 資料庫通知....276
29.7 廣播通知....277
29.8 簡訊通知....279
29.9 Slack 通知....280
29.10 本地化通知....283
29.11 測試....283
29.12 通知事件....284
29.13 自訂頻道....286
30 package 開發....288
30.1 介紹....288
30.2 package 發現....288
30.3 服務提供者....289
30.4 資源....289
30.5 命令....293
30.6 公共資源....293
30.7 發佈檔案組....294
31 處理程序管理....295
31.1 介紹....295
31.2 呼叫過程....295
31.3 非同步處理程序....297
31.4 平行處理....298
31.5 測試....299
32 佇列....303
32.1 簡介....303
32.2 建立任務....305
32.3 任務中介軟體....308
32.4 調度任務....313
32.5 任務批處理....321
32.6 佇列閉包....326
32.7 運行佇列工作者....327
32.8 Supervisor 組態....329
32.9 處理失敗的任務....330
32.10 從佇列中清除任務....334
32.11 監控你的佇列....334
32.12 測試....335
32.13 任務事件....337
33 限流....339
33.1 簡介....339

33.2 基本用法.....339
34 任務調度.....341
34.1 簡介.....341
34.2 定義調度.....341
34.3 運行調度程序.....346
34.4 任務輸出.....346
34.5 任務鉤子.....347
34.6 事件.....348

1 請求的生命週期

1.1 簡介

在「真實世界」中使用任何工具時，如果你瞭解該工具的工作原理，你會更加自信。應用程式開發也不例外。當您瞭解開發工具的功能時，你會覺得使用它們更舒服、更自信。

本文的目的是讓您對 Laravel 框架的工作原理有一個良好的、高層次的理解。通過更好地瞭解整個框架，一切感覺都不那麼「神奇」，你將更有信心建構你的應用程式。如果你不明白所有的規則，不要灰心！只要試著對正在發生的事情有一個基本的掌握，你的知識就會隨著你探索文件的其他部分而增長。

1.2 生命週期概述

1.2.1 第一步

Laravel 應用程式的所有請求的入口點都是 `public/index.php` 檔案。所有請求都由你的 web 伺服器（Apache/Nginx）組態定向到此檔案。那個 `index.php` 檔案不包含太多程式碼。相反，它是載入框架其餘部分的起點。

`index.php` 檔案將載入 Composer 生成的自動載入器定義，然後從 `bootstrap/app.php` 中檢索 Laravel 應用程式的實例。Laravel 本身採取的第一個操作是建立應用 / 服務容器 的實例。

1.2.2 HTTP / Console 核心

接下來，根據進入應用的請求類型，傳入的請求將被傳送到 HTTP 核心或者 Console 核心。這兩個核心充當所有請求流經的中心位置。現在，我們只關注位於 `app/Http/Kernel.php` 中的 HTTP 核心。

HTTP 核心繼承了 `Illuminate\Foundation\Http\Kernel` 類，該類定義了一個將在執行請求之前運行的 `bootstrappers` 陣列。這些啟動載入器用來組態異常處理、組態日誌、檢測應用程式環境，並執行在實際處理請求之前需要完成的其他任務。通常，這些類處理你無需擔心的內部 Laravel 組態。

HTTP 核心還定義了一個 HTTP 中介軟體列表，所有請求在被應用程式處理之前都必須通過該列表。這些中介軟體處理讀寫 HTTP session ，確定應用程式是否處於維護模式， 校驗 CSRF 令牌, 等等。我們接下來會做詳細的討論。

HTTP 核心的 `handle` 方法的簽名非常簡單：它接收 `Request` 介面並返回 `Response` 介面。把核心想像成一個代表整個應用程式的大黑匣子。向它提供 HTTP 請求，它將返回 HTTP 響應。

1.2.3 服務提供者

最重要的核心引導操作之一是為應用程式載入服務提供者。應用程式的所有服務提供程序都在 `config/app.php` 檔案中的 `providers` 陣列。

Laravel 將遍歷這個提供者列表並實例化它們中的每一個。實例化提供程序後，將在所有提供程序上呼叫 `register` 方法。然後，一旦所有的提供者都被註冊了，就會對每個提供程序呼叫 `boot` 方法。服務提供者可能依賴於在執行 `boot` 方法時註冊並可用的每個容器繫結。

服務提供者負責引導框架的所有不同元件，如資料庫、佇列、驗證和路由元件。基本上，Laravel 提供的每個主要功能都是由服務提供商引導和組態的。由於它們引導和組態框架提供的許多特性，服務提供者是整個 Laravel 引導過程中最重要的部分。

1.2.4 路由

應用程式中最重要的服務提供者之一是 `App\Providers\RouteServiceProvider`。此服務提供者載入應用程式的 `routes` 目錄中包含的路由檔案。繼續，打開 `RouteServiceProvider` 程式碼，看看它是如何工作的！

一旦應用程式被引導並且所有服務提供者都被註冊，`Request` 將被傳遞給路由器進行調度。路由器將請求傳送到路由或 controller ，並運行任何路由特定的中介軟體。

中介軟體為過濾或檢查進入應用程式的 HTTP 請求提供了一種方便的機制。例如，Laravel 包含一個這樣的中介軟體，用於驗證應用程式的使用者是否經過身份驗證。如果使用者未通過身份驗證，中介軟體將使用者重新導向到登錄頁。但是，如果使用者經過身份驗證，中介軟體將允許請求進一步進入應用程式。一些中介軟體被分配給應用程式中的所有路由，比如那些在 HTTP 核心的`$middleware` 屬性中定義的路由，而一些只被分配給特定的路由或路由組。你可以通過閱讀完整的中介軟體文件來瞭解關於中介軟體的資訊。

如果請求通過了所有匹配路由分配的中介軟體，則執行路由或 controller 方法，並通過路由的中介軟體鏈路返回路由或 controller 方法的響應。

1.2.5 最後

一旦路由或 controller 方法返回一個響應，該響應將通過路由的中介軟體返回，從而使應用程式有機會修改或檢查傳出的響應。

最後，一旦響應通過中介軟體返回，HTTP 核心的 `handle` 方法將返迴響應對象，並且 `index.php` 檔案在返回的響應上呼叫 `send` 方法。`send` 方法將響應內容傳送到使用者的 Web 瀏覽器。至此，我們已經完成了整個 Laravel 請求生命週期的旅程！

1.3 關注服務提供者

服務提供者確實是引導 Laravel 應用程式的關鍵。建立應用程式實例，註冊服務提供者，並將請求傳遞給引導應用程式。就這麼簡單！

牢牢掌握服務提供者的建構和其對 Laravel 應用處理機制的原理是非常有價值的。你的應用的默認服務提供會存放在 `app/Providers` 目錄下面。

默認情況下，`AppServiceProvider` 是空白的。這裡是用於你新增應用自身的引導處理和服務容器繫結的一個非常棒的地方。在大型項目中，你可能希望建立多個服務提供者，每個服務提供者都為應用程式使用的特定服務提供更細粒度的引導。

2 服務容器

2.1 簡介

Laravel 服務容器是一個用於管理類依賴以及實現依賴注入的強有力工具。依賴注入這個名詞表面看起來花哨，實質上是指：通過建構函式，或者某些情況下通過「setter」方法將類依賴「注入」到類中。

我們來看一個簡單的例子：

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Repositories\UserRepository;
use App\Models\User;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * 建立一個新的 controller 實例
     */
    public function __construct(
        protected UserRepository $users,
    ) {}

    /**
     * 展示給定使用者的資訊
     */
    public function show(string $id): View
    {
        $user = $this->users->find($id);

        return view('user.profile', ['user' => $user]);
    }
}
```

在此示例中，`UserController` 需要從資料來源中檢索使用者。 因此，我們將 **注入** 一個能夠檢索使用者的服務。 在這種情況下，我們的 `UserRepository` 很可能使用 Eloquent 從資料庫中檢索使用者資訊。 然而，由於儲存庫是注入的，我們可以很容易地用另一個實現替換它。 這種方式的便利之處也體現在：當需要為應用編寫測試的時候，我們也可以很輕鬆地 「模擬」 或者建立一個 `UserRepository` 的偽實現來操作。

深入理解服務容器，對於建構一個強大的、大型的應用，以及對 Laravel 核心本身的貢獻都是至關重要的。

2.1.1 零組態解決方案

如果一個類沒有依賴項或只依賴於其他具體類（而不是介面），則不需要指定容器如何解析該類。例如，你可以將以下程式碼放在 `routes/web.php` 檔案中：

```
<?php

class Service
{
    // ...
}

Route::get('/', function (Service $service) {
```

```
    die(get_class($service));
});
```

在這個例子中，點選應用程式的 / 路由將自動解析 Service 類並將其注入到路由的處理程序中。 這是一個有趣的改變。 這意味著你可以開發應用程式並利用依賴注入，而不必擔心臃腫的組態檔案。

很榮幸的通知你，在建構 Laravel 應用程式時，你將要編寫的許多類都可以通過容器自動接收它們的依賴關係，包括 controller、 事件監聽器、 中介軟體 等等。 此外，你可以在 佇列系統 的 handle 方法中鍵入提示依賴項。 一旦你嘗到了自動和零組態依賴注入的力量，你就會覺得沒有它是不可以開發的。

2.1.2 何時使用容器

得益於零組態解決方案，通常情況下，你只需要在路由、 controller 、事件偵聽器和其他地方鍵入提示依賴項，而不必手動與容器打交道。例如，可以在路由定義中鍵入 Illuminate\Http\Request 對象，以便輕鬆訪問當前請求的 Request 類。儘管我們不必與容器互動來編寫此程式碼，但它在幕後管理著這些依賴項的注入：

```
use Illuminate\Http\Request;

Route::get('/', function (Request $request) {
    // ...
});
```

在許多情況下，由於自動依賴注入和 facades ，你在建構 Laravel 應用程式，而無需手動繫結或解析容器中的任何內容。 **那麼，你什麼時候會手動與容器打交道呢？** 讓我們來看看下面兩種情況。

首先，如果你編寫了一個實現介面的類，並希望在路由或類的建構函式上鍵入該介面的提示，則必須 告訴容器如何解析該介面。第二，如果你正在 編寫一個 Laravel 包 計畫與其他 Laravel 開發人員共享，那麼你可能需要將包的服務繫結到容器中。

2.2 繫結

2.2.1 基礎繫結

2.2.1.1 簡單繫結

幾乎所有的服務容器繫結都會在 服務提供者 中註冊，下面示例中的大多數將演示如何在該上下文（服務提供者）中使用容器。

在服務提供者中，你總是可以通過 $this->app 屬性訪問容器。我們可以使用 bind 方法註冊一個繫結，將我們希望註冊的類或介面名稱與返回類實例的閉包一起傳遞:

```
use App\Services\Transistor;
use App\Services\PodcastParser;
use Illuminate\Contracts\Foundation\Application;

$this->app->bind(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

注意，我們接受容器本身作為解析器的參數。然後，我們可以使用容器來解析正在建構的對象的子依賴。

如前所述，你通常會在服務提供者內部與容器進行互動；但是，如果你希望在服務提供者外部與容器進行互動，則可以通過 App facade 進行:

```
use App\Services\Transistor;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Support\Facades\App;
```

```
App::bind(Transistor::class, function (Application $app) {
    // ...
});
```

技巧 如果類不依賴於任何介面，則不需要將它們繫結到容器中。不需要指示容器如何建構這些對象，因為它可以使用反射自動解析這些對象。

2.2.1.2 單例的繫結

`singleton` 方法將類或介面繫結到只應解析一次的容器中。解析單例繫結後，後續呼叫容器時將返回相同的對象實例：

```
use App\Services\Transistor;
use App\Services\PodcastParser;
use Illuminate\Contracts\Foundation\Application;

$this->app->singleton(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

2.2.1.3 繫結範疇單例

`scoped` 方法將一個類或介面繫結到容器中，該容器只應在給定的 Laravel 請求 / 作業生命週期內解析一次。雖然該方法與 `singleton` 方法類似，但是當 Laravel 應用程式開始一個新的「生命週期」時， 使用 `scoped` 方法註冊的實例 將被刷新，例如當 Laravel Octane 工作者處理新請求或 Laravel 佇列系統處理新作業時：

```
use App\Services\Transistor;
use App\Services\PodcastParser;
use Illuminate\Contracts\Foundation\Application;

$this->app->scoped(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

2.2.1.4 繫結實例

你也可以使 `instance` 方法將一個現有的對象實例繫結到容器中。給定的實例總會在後續對容器的呼叫中返回:

```
use App\Services\Transistor;
use App\Services\PodcastParser;

$service = new Transistor(new PodcastParser);

$this->app->instance(Transistor::class, $service);
```

2.2.2 將介面繫結實例

服務容器的一個非常強大的特性是它能夠將介面繫結到給定的實例。例如，我們假設有一個 `EventPusher` 介面和一個 `RedisEventPusher` 實例。一旦我們編寫了這個介面的 `RedisEventPusher` 實例，我們就可以像這樣把它註冊到服務容器中:

```
use App\Contracts\EventPusher;
use App\Services\RedisEventPusher;

$this->app->bind(EventPusher::class, RedisEventPusher::class);
```

這條語句告訴容器，當類需要 `EventPusher` 的實例時，它應該注入 `RedisEventPusher`。現在我們可以在由容器解析的類的建構函式中輸入 `EventPusher` 介面。記住， controller 、事件監聽器、中介軟體和 Laravel 應用程式中的各種其他類型的類總是使用容器進行解析的:

```
use App\Contracts\EventPusher;

/**
 * Create a new class instance.
 */
public function __construct(
    protected EventPusher $pusher
) {}
```

2.2.3 上下文繫結

譯者註：所謂「上下文繫結」就是根據上下文進行動態的繫結，指依賴的上下文關係。

有時你可能有兩個類使用相同的介面，但是你希望將不同的實現分別注入到各自的類中。例如，兩個 controller 可能依賴於 `Illuminate\Contracts\Filesystem\Filesystem` 契約 的不同實現。Laravel 提供了一個簡單流暢的方式來定義這種行為：

```
use App\Http\Controllers\PhotoController;
use App\Http\Controllers\UploadController;
use App\Http\Controllers\VideoController;
use Illuminate\Contracts\Filesystem\Filesystem;
use Illuminate\Support\Facades\Storage;

$this->app->when(PhotoController::class)
          ->needs(Filesystem::class)
          ->give(function () {
              return Storage::disk('local');
          });

$this->app->when([VideoController::class, UploadController::class])
          ->needs(Filesystem::class)
          ->give(function () {
              return Storage::disk('s3');
          });
```

2.2.4 繫結原語

有時，你可能有一個接收一些注入類的類，但也需要一個注入的原語值，如整數。你可以很容易地使用上下文繫結來，注入類可能需要的任何值:

```
use App\Http\Controllers\UserController;

$this->app->when(UserController::class)
          ->needs('$variableName')
          ->give($value);
```

有時，類可能依賴於 標籤 實例的陣列。使用 `giveTagged` 方法，你可以很容易地注入所有帶有該標籤的容器繫結:

```
$this->app->when(ReportAggregator::class)
    ->needs('$reports')
    ->giveTagged('reports');
```

如果你需要從應用程式的某個組態檔案中注入一個值，你可以使用 `giveConfig` 方法:

```
$this->app->when(ReportAggregator::class)
    ->needs('$timezone')
    ->giveConfig('app.timezone');
```

2.2.5 繫結變長參數類型

有時，你可能有一個使用可變建構函式參數接收類型對象陣列的類：

```
<?php

use App\Models\Filter;
use App\Services\Logger;

class Firewall
{
    /**
     * 過濾器實例組
     *
     * @var array
     */
    protected $filters;

    /**
     * 建立一個類實例
     */
    public function __construct(
        protected Logger $logger,
        Filter ...$filters,
    ) {
        $this->filters = $filters;
    }
}
```

使用上下文繫結，你可以通過提供 `give` 方法一個閉包來解決這個依賴，該閉包返回一個已解析的 `Filter` 實例陣列：

```
$this->app->when(Firewall::class)
           ->needs(Filter::class)
           ->give(function (Application $app) {
                 return [
                      $app->make(NullFilter::class),
                      $app->make(ProfanityFilter::class),
                      $app->make(TooLongFilter::class),
                 ];
           });
```

為方便起見，你也可以只提供一個類名陣列，以便在 `Firewall` 需要 `Filter` 實例時由容器解析:

```
$this->app->when(Firewall::class)
           ->needs(Filter::class)
           ->give([
               NullFilter::class,
               ProfanityFilter::class,
               TooLongFilter::class,
           ]);
```

2.2.5.1 變長參數的關聯標籤

有時，一個類可能具有類型提示為給定類的可變依賴項（`Report ...$reports`）。使用 `needs` 和 `giveTagged` 方法，你可以輕鬆地為給定依賴項注入所有帶有該 標籤 的所有容器繫結：

```
$this->app->when(ReportAggregator::class)
    ->needs(Report::class)
    ->giveTagged('reports');
```

2.2.6 標籤

有時，你可能需要解決所有特定「類別」的繫結。例如，也許你正在建構一個報告分析器，它接收許多不同的 `Report` 介面實現的陣列。註冊 `Report` 實現後，你可以使用 `tag` 方法為它們分配標籤：

```
$this->app->bind(CpuReport::class, function () {
    // ...
```

```
});

$this->app->bind(MemoryReport::class, function () {
    // ...
});

$this->app->tag([CpuReport::class, MemoryReport::class], 'reports');
```

一旦服務被打上標籤，你就可以通過容器的 `tagged` 方法輕鬆地解析它們：

```
$this->app->bind(ReportAnalyzer::class, function (Application $app) {
    return new ReportAnalyzer($app->tagged('reports'));
});
```

2.2.7 繼承繫結

`extend` 方法允許修改已解析的服務。例如，解析服務時，可以運行其他程式碼來修飾或組態服務。`extend` 方法接受閉包，該閉包應返回修改後的服務作為其唯一參數。閉包接收正在解析的服務和容器實例：

```
$this->app->extend(Service::class, function (Service $service, Application $app) {
    return new DecoratedService($service);
});
```

2.3 解析

2.3.1 `make` 方法

你可以使用 `make` 方法從容器中解析出一個類實例。`make` 方法接受你要解析的類或介面的名稱：

```
use App\Services\Transistor;

$transistor = $this->app->make(Transistor::class);
```

如果你的某些類依賴關係無法通過容器解析，請通過將它們作為關聯陣列傳遞到 `makeWith` 方法中來注入它們。例如，我們可以手動傳遞 `Transistor` 服務所需的 `$id` 建構函式參數：

```
use App\Services\Transistor;

$transistor = $this->app->makeWith(Transistor::class, ['id' => 1]);
```

如果你不在服務提供程序外部的程式碼位置中，並且沒有訪問 `$app` 變數的權限，你可以使用 `App` facade 或 `app` helper 來從容器中解析出一個類實例：

```
use App\Services\Transistor;
use Illuminate\Support\Facades\App;

$transistor = App::make(Transistor::class);

$transistor = app(Transistor::class);
```

如果你想將 Laravel 容器實例本身注入到由容器解析的類中，你可以在你的類的建構函式上進行類型提示，指定 `Illuminate\Container\Container` 類型：

```
use Illuminate\Container\Container;

/**
 * 建立一個新的類實例。
 */
public function __construct( protected Container $container ) {}
```

2.3.2 自動注入

或者，你可以在由容器解析的類的建構函式中類型提示依賴項，包括 controller、事件監聽器、中介軟體 等。此外，你可以在 佇列作業 的 handle 方法中類型提示依賴項。在實踐中，這是大多數對象應該由容器解析的方式。

例如，你可以在 controller 的建構函式中新增一個 repository 的類型提示，然後這個 repository 將會被自動解析並注入類中：

```
<?php

namespace App\Http\Controllers;

use App\Repositories\UserRepository;
use App\Models\User;

class UserController extends Controller
{
    /**
     * 建立一個 controller 實例
     */
    public function __construct(
        protected UserRepository $users,
    ) {}

    /**
     * 使用給定的 ID 顯示 user
     */
    public function show(string $id): User
    {
        $user = $this->users->findOrFail($id);

        return $user;
    }
}
```

2.4 方法呼叫和注入

有時你可能希望呼叫對象實例上的方法，同時允許容器自動注入該方法的依賴項。例如，給定以下類：

```
<?php

namespace App;

use App\Repositories\UserRepository;

class UserReport
{
    /**
     * 生成新的使用者報告
     */
    public function generate(UserRepository $repository): array
    {
        return [
            // ...
        ];
    }
}
```

你可以通過容器呼叫 generate 方法，如下所示：

```
use App\UserReport;
use Illuminate\Support\Facades\App;
```

```
$report = App::call([new UserReport, 'generate']);
```

`call` 方法接受任何可呼叫的 PHP 方法。容器的 `call` 方法甚至可以用於呼叫閉包，同時自動注入其依賴項：

```
use App\Repositories\UserRepository;
use Illuminate\Support\Facades\App;

$result = App::call(function (UserRepository $repository) {
    // ...
});
```

2.5 容器事件

服務容器每次解析對象時都會觸發一個事件。你可以使用 `resolving` 方法監聽此事件：

```
use App\Services\Transistor;
use Illuminate\Contracts\Foundation\Application;

$this->app->resolving(Transistor::class, function (Transistor $transistor, Application
$app) {
    // 當容器解析「Transistor」類型的對象時呼叫...
});

$this->app->resolving(function (mixed $object, Application $app) {
    // 當容器解析任何類型的對象時呼叫...
});
```

如你所見，正在解析的對象將被傳遞給回呼，從而允許你在對象提供給其使用者之前設定對象的任何其他屬性。

2.6 PSR-11

Laravel 的服務容器實現了 PSR-11 介面。因此，你可以新增 PSR-11 容器介面的類型提示來獲取 Laravel 容器的實例：

```
use App\Services\Transistor;
use Psr\Container\ContainerInterface;

Route::get('/', function (ContainerInterface $container) {
    $service = $container->get(Transistor::class);

    // ...
});
```

如果無法解析給定的識別碼，將引發異常。如果識別碼從未繫結，則異常將是 `Psr\Container\NotFoundExceptionInterface` 的實例。如果識別碼已繫結但無法解析，則將拋出 `Psr\Container\ContainerExceptionInterface` 的實例。

3 服務提供者

3.1 簡介

服務提供者是所有 Laravel 應用程式的引導中心。你的應用程式，以及通過伺服器引導的 Laravel 核心服務都是通過服務提供器引導。

但是，「引導」是什麼意思呢？通常，我們可以理解為**註冊**，比如註冊服務容器繫結，事件監聽器，中介軟體，甚至是路由。服務提供者是組態應用程式的中心。

當你打開 Laravel 的 `config/app.php` 檔案時，你會看到 `providers` 陣列。陣列中的內容是應用程式要載入的所有服務提供者的類。當然，其中有很多「延遲」提供者，他們並不會在每次請求的時候都載入，只有他們的服務實際被需要時才會載入。

本篇你將會學到如何編寫自己的服務提供者，並將其註冊到你的 Laravel 應用程式中。

> **技巧** 如果你想瞭解有關 Laravel 如何處理請求並在內部工作的更多資訊，請查看有關 Laravel 的文件 請求生命週期。

3.2 編寫服務提供者

所有的服務提供者都會繼承 `Illuminate\Support\ServiceProvider` 類。大多服務提供者都包含一個 register 和一個 `boot` 方法。在 `register` 方法中，你只需要將服務繫結到 `register` 方法中， 你只需要 **將服務繫結到 服務容器**。而不要嘗試在 `register` 方法中註冊任何監聽器，路由，或者其他任何功能。

使用 Artisan 命令列工具，通過 `make:provider` 命令可以生成一個新的提供者：

```
php artisan make:provider RiakServiceProvider
```

3.2.1 註冊方法

如上所述，在 `register` 方法中，你只需要將服務繫結到 服務容器 中。而不要嘗試在 `register` 方法中註冊任何監聽器，路由，或者其他任何功能。否則，你可能會意外地使用到尚未載入的服務提供者提供的服務。

讓我們來看一個基礎的服務提供者。在任何服務提供者方法中，你總是通過 $app 屬性來訪問服務容器：

```
<?php

namespace App\Providers;

use App\Services\Riak\Connection;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider
{
    /**
     * 註冊應用服務
     */
    public function register(): void
    {
        $this->app->singleton(Connection::class, function (Application $app) {
            return new Connection(config('riak'));
```

```
        });
    }
}
```

這個服務提供者只是定義了一個 `register` 方法，並且使用這個方法在服務容器中定義了一個 `Riak\Connection` 介面。如果你不理解服務容器的工作原理，請查看其 文件.

3.2.1.1 bindings 和 singletons 的特性

如果你的服務提供器註冊了許多簡單的繫結，你可能想用 `bindings` 和 `singletons` 屬性替代手動註冊每個容器繫結。當服務提供器被框架載入時，將自動檢查這些屬性並註冊相應的繫結：

```
<?php

namespace App\Providers;

use App\Contracts\DowntimeNotifier;
use App\Contracts\ServerProvider;
use App\Services\DigitalOceanServerProvider;
use App\Services\PingdomDowntimeNotifier;
use App\Services\ServerToolsProvider;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     *  所有需要註冊的容器繫結
     *
     * @var array
     */
    public $bindings = [
        ServerProvider::class => DigitalOceanServerProvider::class,
    ];

    /**
     * 所有需要註冊的容器單例
     *
     * @var array
     */
    public $singletons = [
        DowntimeNotifier::class => PingdomDowntimeNotifier::class,
        ServerProvider::class => ServerToolsProvider::class,
    ];
}
```

3.2.2 引導方法

如果我們要在服務提供者中註冊一個 檢視合成器 該怎麼做？這就需要用到 `boot` 方法了。**該方法在所有服務提供者被註冊以後才會被呼叫**，這就是說我們可以在其中訪問框架已註冊的所有其它服務：

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;
use Illuminate\Support\ServiceProvider;

class ComposerServiceProvider extends ServiceProvider
{
    /**
     * 啟動所有的應用服務
     */
    public function boot(): void
    {
```

```
        View::composer('view', function () {
            // ...
        });
    }
}
```

3.2.2.1 啟動方法的依賴注入

你可以為服務提供者的 `boot` 方法設定類型提示。服務容器 會自動注入你所需要的依賴：

```
use Illuminate\Contracts\Routing\ResponseFactory;

/**
 * 引導所有的應用服務
 */
public function boot(ResponseFactory $response): void
{
    $response->macro('serialized', function (mixed $value) {
        // ...
    });
}
```

3.3 註冊服務提供者

所有服務提供者都是通過組態檔案 `config/app.php` 進行註冊。該檔案包含了一個列出所有服務提供者名字的 `providers` 陣列，默認情況下，其中列出了所有核心服務提供者，這些服務提供者啟動 Laravel 核心元件，比如郵件、佇列、快取等等。

要註冊提供器，只需要將其新增到陣列：

```
'providers' => [
    // 其他服務提供者

    App\Providers\ComposerServiceProvider::class,
],
```

3.4 延遲載入提供者

如果你的服務提供者 **只** 在 服務容器中註冊，可以選擇延遲載入該繫結直到註冊繫結的服務真的需要時再載入，延遲載入這樣的一個提供者將會提升應用的性能，因為它不會在每次請求時都從檔案系統載入。

Laravel 編譯並保存延遲服務提供者提供的所有服務的列表，以及其服務提供者類的名稱。因此，只有當你在嘗試解析其中一項服務時，Laravel 才會載入服務提供者。

要延遲載入提供者，需要實現 `\Illuminate\Contracts\Support\DeferrableProvider` 介面並置一個 `provides` 方法。這個 `provides` 方法返回該提供者註冊的服務容器繫結：

```
<?php

namespace App\Providers;

use App\Services\Riak\Connection;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Contracts\Support\DeferrableProvider;
use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider implements DeferrableProvider
{
    /**
     * 註冊所有的應用服務
     */
```

```
    public function register(): void
    {
        $this->app->singleton(Connection::class, function (Application $app) {
            return new Connection($app['config']['riak']);
        });
    }

    /**
     * 獲取服務提供者的服務
     *
     * @return array<int, string>
     */
    public function provides(): array
    {
        return [Connection::class];
    }
}
```

4 Facades

4.1 簡介

在整個 Laravel 文件中，你將看到通過 Facades 與 Laravel 特性互動的程式碼示例。Facades 為應用程式的服務容器中可用的類提供了「靜態代理」。在 Laravel 這艘船上有許多 Facades，提供了幾乎所有 Laravel 的特徵。

Laravel Facades 充當服務容器中底層類的「靜態代理」，提供簡潔、富有表現力的好處，同時保持比傳統靜態方法更多的可測試性和靈活性。如果你不完全理解引擎蓋下的 Facades 是如何工作的，那也沒問題，跟著流程走，繼續學習 Laravel。

Laravel 的所有 Facades 都在 `Illuminate\Support\Facades` 命名空間中定義。因此，我們可以很容易地訪問這樣一個 Facades ：

```
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\Facades\Route;

Route::get('/cache', function () {
    return Cache::get('key');
});
```

在整個 Laravel 文件中，許多示例將使用 Facades 來演示框架的各種特性。

4.1.1.1 輔助函數

為了補充 Facades，Laravel 提供了各種全域 「助手函數」，使它更容易與常見的 Laravel 功能進行互動。可以與之互動的一些常用助手函數有 `view`, `response`, `url`, `config` 等。Laravel 提供的每個助手函數都有相應的特性；但是，在專用的輔助函數文件中有一個完整的列表。

例如，我們可以使用 `response` 函數而不是 `Illuminate\Support\Facades\Response` Facade 生成 JSON 響應。由於「助手函數」是全域可用的，因此無需匯入任何類即可使用它們：

```
use Illuminate\Support\Facades\Response;

Route::get('/users', function () {
    return Response::json([
        // ...
    ]);
});

Route::get('/users', function () {
    return response()->json([
        // ...
    ]);
});
```

4.2 何時使用 Facades

Facades 有很多好處。它們提供了簡潔、易記的語法，讓你可以使用 Laravel 的功能而不必記住必須手動注入或組態的長類名。此外，由於它們獨特地使用了 PHP 的動態方法，因此它們易於測試。

然而，在使用 Facades 時必須小心。Facades 的主要危險是類的「範疇洩漏」。由於 Facades 如此易於使用並且不需要注入，因此讓你的類繼續增長並在單個類中使用許多 Facades 可能很容易。使用依賴注入，這種潛在問題通過建構函式變得明顯，告訴你的類過於龐大。因此，在使用 Facades 時，需要特別關注類的大小，以便它的責任範圍保持狹窄。如果你的類變得太大，請考慮將它拆分成多個較小的類。

4.2.1 Facades 與 依賴注入

依賴注入的主要好處之一是能夠替換注入類的實現。這在測試期間很有用，因為你可以注入一個模擬或存根並斷言各種方法是否在存根上呼叫了。

通常，真正的靜態方法是不可能 mock 或 stub 的。無論如何，由於 Facades 使用動態方法對服務容器中解析出來的對象方法的呼叫進行了代理， 我們也可以像測試注入類實例一樣測試 Facades。比如，像下面的路由：

```
use Illuminate\Support\Facades\Cache;

Route::get('/cache', function () {
    return Cache::get('key');
});
```

使用 Laravel 的 Facade 測試方法，我們可以編寫以下測試用例來驗證是否 Cache::get 使用我們期望的參數呼叫了該方法：

```
use Illuminate\Support\Facades\Cache;

/**
 *  一個基礎功能的測試用例
 */
public function test_basic_example(): void
{
    Cache::shouldReceive('get')
         ->with('key')
         ->andReturn('value');

    $response = $this->get('/cache');

    $response->assertSee('value');
}
```

4.2.2 Facades Vs 助手函數

除了 Facades，Laravel 還包含各種「輔助函數」來實現這些常用功能，比如生成檢視、觸發事件、任務調度或者傳送 HTTP 響應。許多輔助函數都有與之對應的 Facade。例如，下面這個 Facades 和輔助函數的作用是一樣的：

```
return Illuminate\Support\Facades\View::make('profile');

return view('profile');
```

Facades 和輔助函數之間沒有實際的區別。 當你使用輔助函數時，你可以像測試相應的 Facade 那樣進行測試。例如，下面的路由：

```
Route::get('/cache', function () {
    return cache('key');
});
```

在底層實現，輔助函數 cache 實際是呼叫 Cache 這個 Facade 的 get 方法。因此，儘管我們使用的是輔助函數，我們依然可以帶上我們期望的參數編寫下面的測試程式碼來驗證該方法：

```
use Illuminate\Support\Facades\Cache;

/**
 * 一個基礎功能的測試用例
 */
public function test_basic_example(): void
{
    Cache::shouldReceive('get')
         ->with('key')
         ->andReturn('value');

    $response = $this->get('/cache');
```

```
    $response->assertSee('value');
}
```

4.3 Facades 工作原理

在 Laravel 應用程式中，Facades 是一個提供從容器訪問對象的類。完成這項工作的部分屬於 `Facade` 類。Laravel 的 Facade、以及你建立的任何自訂 Facade，都繼承自 `Illuminate\Support\Facades\Facade` 類。

`Facade` 基類使用 `__callStatic()` 魔術方法將來自 Facade 的呼叫推遲到從容器解析出對象後。在下面的示例中，呼叫了 Laravel 快取系統。看一眼這段程式碼，人們可能會假設靜態的 `get` 方法正在 `Cache` 類上被呼叫：

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\Cache;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     */
    public function showProfile(string $id): View
    {
        $user = Cache::get('user:'.$id);

        return view('profile', ['user' => $user]);
    }
}
```

請注意，在檔案頂部附近，我們正在「匯入」`Cache` Facade。這個 Facade 作為訪問 `Illuminate\Contracts\Cache\Factory` 介面底層實現的代理。我們使用 Facade 進行的任何呼叫都將傳遞給 Laravel 快取服務的底層實例。

如果我們查看 `Illuminate\Support\Facades\Cache` 類，你會發現沒有靜態方法 `get`：

```
class Cache extends Facade
{
    /**
     * Get the registered name of the component.
     */
    protected static function getFacadeAccessor(): string
    {
        return 'cache';
    }
}
```

相反，`Cache` Facade 繼承了 `Facade` 基類並定義了 `getFacadeAccessor()` 方法。此方法的工作是返回服務容器繫結的名稱。當使用者引用 `Cache` Facade 上的任何靜態方法時，Laravel 會從 服務容器 中解析 `cache` 繫結並運行該對象請求的方法（在這個例子中就是 `get` 方法）

4.4 即時 Facades

使用即時 Facade, 你可以將應用程式中的任何類視為 Facade。為了說明這是如何使用的， 讓我們首先看一下一些不使用即時 Facade 的程式碼。例如，假設我們的 `Podcast` 模型有一個 `publish` 方法。 但是，為了發佈 `Podcast`，我們需要注入一個 `Publisher` 實例：

```
<?php

namespace App\Models;

use App\Contracts\Publisher;
use Illuminate\Database\Eloquent\Model;

class Podcast extends Model
{
     /**
      * Publish the podcast.
      */
     public function publish(Publisher $publisher): void
     {
          $this->update(['publishing' => now()]);

          $publisher->publish($this);
     }
}
```

將 publisher 的實現注入到該方法中，我們可以輕鬆地測試這種方法，因為我們可以模擬注入的 publisher 。但是，它要求我們每次呼叫 `publish` 方法時始終傳遞一個 publisher 實例。 使用即時的 Facades, 我們可以保持同樣的可測試性，而不需要顯式地通過 `Publisher` 實例。要生成即時 Facade，請在匯入類的名稱空間中加上 `Facades`：

```
<?php

namespace App\Models;

use Facades\App\Contracts\Publisher;
use Illuminate\Database\Eloquent\Model;

class Podcast extends Model
{
     /**
      * Publish the podcast.
      */
     public function publish(): void
     {
          $this->update(['publishing' => now()]);

          Publisher::publish($this);
     }
}
```

當使用即時 Facade 時， publisher 實現將通過使用 `Facades` 前綴後出現的介面或類名的部分來解決服務容器的問題。在測試時，我們可以使用 Laravel 的內建 Facade 測試輔助函數來模擬這種方法呼叫：

```
<?php

namespace Tests\Feature;

use App\Models\Podcast;
use Facades\App\Contracts\Publisher;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Tests\TestCase;

class PodcastTest extends TestCase
{
     use RefreshDatabase;

     /**
      * A test example.
      */
     public function test_podcast_can_be_published(): void
     {
          $podcast = Podcast::factory()->create();
```

```
        Publisher::shouldReceive('publish')->once()->with($podcast);

        $podcast->publish();
    }
}
```

5 路由

5.1 基本路由

最基本的 Laravel 路由接受一個 URI 和一個閉包，提供了一個簡單優雅的方法來定義路由和行為，而不需要複雜的路由組態檔案：

```
use Illuminate\Support\Facades\Route;

Route::get('/greeting', function () {
    return 'Hello World';
});
```

5.1.1.1 默認路由檔案

所有 Laravel 路由都定義在你的路由檔案中，它位於 `routes` 目錄。這些檔案會被你的應用程式中的 `App\Providers\RouteServiceProvider` 自動載入。`routes/web.php` 檔案用於定義 web 介面的路由。這些路由被分配給 `web` 中介軟體組，它提供了 session 狀態和 CSRF 保護等功能。定義在 `routes/api.php` 中的路由都是無狀態的，並且被分配了 `api` 中介軟體組。

對於大多數應用程式，都是以在 `routes/web.php` 檔案定義路由開始的。可以通過在瀏覽器中輸入定義的路由 URL 來訪問 `routes/web.php` 中定義的路由。例如，你可以在瀏覽器中輸入 `http://example.com/user` 來訪問以下路由：

```
use App\Http\Controllers\UserController;

Route::get('/user', [UserController::class, 'index']);
```

定義在 `routes/api.php` 檔案中的路由是被 `RouteServiceProvider` 巢狀在一個路由組內。 在這個路由組內，將自動應用 `/api` URI 前綴，所以你無需手動將其應用於檔案中的每個路由。你可以通過修改 `RouteServiceProvider` 類來修改前綴和其他路由組選項。

5.1.1.2 可用的路由方法

路由器允許你註冊能響應任何 HTTP 請求的路由

```
Route::get($uri, $callback);
Route::post($uri, $callback);
Route::put($uri, $callback);
Route::patch($uri, $callback);
Route::delete($uri, $callback);
Route::options($uri, $callback);
```

有的時候你可能需要註冊一個可響應多個 HTTP 請求的路由，這時你可以使用 `match` 方法，也可以使用 `any` 方法註冊一個實現響應所有 HTTP 請求的路由：

```
Route::match(['get', 'post'], '/', function () {
    // ...
});

Route::any('/', function () {
    // ...
});
```

> **技巧** 當定義多個相同路由時，使用 `get`， `post`， `put`， `patch`， `delete`， 和 `options` 方法的路由應該在使用 `any`， `match`， 和 `redirect` 方法的路由之前定義，這樣可以確保請求與正確的路由匹配。

5.1.1.3 依賴注入

你可以在路由的回呼方法中，以形參的方式聲明路由所需要的任何依賴項。這些依賴會被 Laravel 的 容器 自動解析並注入。 例如，你可以在閉包中聲明 `Illuminate\Http\Request` 類， 讓當前的 HTTP 請求自動注入依賴到你的路由回呼中：

```
use Illuminate\Http\Request;

Route::get('/users', function (Request $request) {
    // ...
});
```

5.1.1.4 CSRF 保護

請記住，任何指向 `POST`、`PUT`、`PATCH` 或 `DELETE` 路由(在 `web` 路由檔案中定義)的 HTML 表單都應該包含 CSRF 令牌字，否則請求會被拒絕。更多 CSRF 保護的相關資訊請閱讀 CSRF 文件：

```
<form method="POST" action="/profile">
    @csrf
    ...
</form>
```

5.1.2 重新導向路由

如果要定義一個重新導向到另一個 URI 的路由，可以使用 `Route::redirect` 方法。這個方法可以快速的實現重新導向，而不再需要去定義完整的路由或者 controller ：

```
Route::redirect('/here', '/there');
```

默認情況下，`Route::redirect` 返回 `302` 狀態碼。你可以使用可選的第三個參數自訂狀態碼：

```
Route::redirect('/here', '/there', 301);
```

或者，你也可以使用 `Route::permanentRedirect` 方法返回 `301` 狀態碼：

```
Route::permanentRedirect('/here', '/there');
```

> **警告**
> 在重新導向路由中使用路由參數時，以下參數由 Laravel 保留，不能使用：`destination` 和 `status`。

5.1.3 檢視路由

如果你的路由只需返回一個檢視，你可以使用 `Route::view` 方法。就像 `redirect` 方法，該方法提供了一個讓你不必定義完整路由或 controller 的便捷操作。這個 `view` 方法的第一個參數是 URI，第二個參數為檢視名稱。此外，你也可以在可選的第三個參數中傳入陣列，將陣列的資料傳遞給檢視：

```
Route::view('/welcome', 'welcome');

Route::view('/welcome', 'welcome', ['name' => 'Taylor']);
```

> **警告**
> 在檢視路由中使用參數時，下列參數由 Laravel 保留，不能使用：`view`、`data`, `status` 及 `headers`。

5.1.4 route:list 命令

使用 `route:list` Artisan 命令可以輕鬆提供應用程式定義的所有路線的概述：

```
php artisan route:list
```

正常情況下，`route:list` 不會顯示分配給路由的中介軟體資訊；但是你可以通過在命令中新增 `-v` 選項 來顯示路由中的中介軟體資訊：

```
php artisan route:list -v
```

你也可以通過 `--path` 來顯示指定的 URL 開頭的路由：

```
php artisan route:list --path=api
```

此外，在執行 `route:list` 命令時，可以通過提供 `--except vendor` 選項來隱藏由第三方包定義的任何路由：

```
php artisan route:list --except-vendor
```

同理，也可以通過在執行 `route:list` 命令時提供 `--only vendor` 選項來顯示由第三方包定義的路由：

```
php artisan route:list --only-vendor
```

5.2 路由參數

5.2.1 必需參數

有時你將需要捕獲路由內的 URI 段。例如，你可能需要從 URL 中捕獲使用者的 ID。你可以通過定義路由參數來做到這一點：

```
Route::get('/user/{id}', function (string $id) {
    return 'User '.$id;
});
```

也可以根據你的需要在路由中定義多個參數：

```
Route::get('/posts/{post}/comments/{comment}', function (string $postId, string
$commentId) {
    // ...
});
```

路由的參數通常都會被放在 `{}` ，並且參數名只能為字母。下劃線 (_) 也可以用於路由參數名中。路由參數會按路由定義的順序依次注入到路由回呼或者 controller 中，而不受回呼或者 controller 的參數名稱的影響。

5.2.1.1 必填參數

如果你的路由具有依賴關係，而你希望 Laravel 服務容器自動注入到路由的回呼中，則應在依賴關係之後列出路由參數：

```
use Illuminate\Http\Request;

Route::get('/user/{id}', function (Request $request, string $id) {
    return 'User '.$id;
});
```

5.2.2 可選參數

有時，你可能需要指定一個路由參數，但你希望這個參數是可選的。你可以在參數後面加上 ? 標記來實現，但前提是要確保路由的相應變數有預設值：

```
Route::get('/user/{name?}', function (string $name = null) {
    return $name;
});

Route::get('/user/{name?}', function (string $name = 'John') {
```

```
    return $name;
});
```

5.2.3 正規表示式約束

你可以使用路由實例上的 `where` 方法來限制路由參數的格式。 `where` 方法接受參數的名稱和定義如何約束參數的正規表示式：

```
Route::get('/user/{name}', function (string $name) {
    // ...
})->where('name', '[A-Za-z]+');

Route::get('/user/{id}', function (string $id) {
    // ...
})->where('id', '[0-9]+');

Route::get('/user/{id}/{name}', function (string $id, string $name) {
    // ...
})->where(['id' => '[0-9]+', 'name' => '[a-z]+']);
```

為方便起見，一些常用的正規表示式模式具有幫助方法，可讓你快速將模式約束新增到路由：

```
Route::get('/user/{id}/{name}', function (string $id, string $name) {
    // ...
})->whereNumber('id')->whereAlpha('name');

Route::get('/user/{name}', function (string $name) {
    // ...
})->whereAlphaNumeric('name');

Route::get('/user/{id}', function (string $id) {
    // ...
})->whereUuid('id');

Route::get('/user/{id}', function (string $id) {
    //
})->whereUlid('id');

Route::get('/category/{category}', function (string $category) {
    // ...
})->whereIn('category', ['movie', 'song', 'painting']);
```

如果傳入請求與路由模式約束不匹配，將返回 404 HTTP 響應。

5.2.3.1 全域約束

如果你希望路由參數始終受給定正規表示式的約束，你可以使用 `pattern` 方法。 你應該在 `App\Providers\RouteServiceProvider` 類的 `boot` 方法中定義這些模式：

```
/**
 * 定義路由模型繫結、模式篩選器等。
 */
public function boot(): void
{
    Route::pattern('id', '[0-9]+');
}
```

一旦定義了模式，它就會自動應用到使用該參數名稱的所有路由：

```
Route::get('/user/{id}', function (string $id) {
    // 僅當 {id} 是數字時執行。。。
});
```

5.2.3.2 編碼正斜槓

Laravel 路由元件允許除 / 之外的所有字元出現在路由參數值中。 你必須使用 where 條件正規表示式明確允許 / 成為預留位置的一部分：

```
Route::get('/search/{search}', function (string $search) {
    return $search;
})->where('search', '.*');
```

注意：僅在最後一個路由段中支援編碼的正斜槓。

5.3 命名路由

命名路由允許為特定路由方便地生成 URL 或重新導向。通過將 name 方法連結到路由定義上，可以指定路由的名稱：

```
Route::get('/user/profile', function () {
    // ...
})->name('profile');
```

你還可以為 controller 操作指定路由名稱：

```
Route::get(
    '/user/profile',
    [UserProfileController::class, 'show']
)->name('profile');
```

注意：路由名稱應始終是唯一的。

5.3.1.1 生成命名路由的 URL

一旦你為給定的路由分配了一個名字，你可以在通過 Laravel 的 route 和 redirect 輔助函數生成 URL 或重新導向時使用該路由的名稱：

```
// 生成 URL。。。
$url = route('profile');

// 生成重新導向。。。
return redirect()->route('profile');

return to_route('profile');
```

如果命名路由定義了參數，你可以將參數作為第二個參數傳遞給 route 函數。 給定的參數將自動插入到生成的 URL 的正確位置：

```
Route::get('/user/{id}/profile', function (string $id) {
    // ...
})->name('profile');

$url = route('profile', ['id' => 1]);
```

如果你在陣列中傳遞其他參數，這些鍵 / 值對將自動新增到生成的 URL 的查詢字串中：

```
Route::get('/user/{id}/profile', function (string $id) {
    // ...
})->name('profile');

$url = route('profile', ['id' => 1, 'photos' => 'yes']);

// /user/1/profile?photos=yes
```

技巧：有時，你可能希望為 URL 參數指定請求範圍的預設值，例如當前語言環境。 為此，你可以使用 URL::defaults 方法.

5.3.1.2 檢查當前路由

如果你想確定當前請求是否路由到給定的命名路由，你可以在 Route 實例上使用 `named` 方法。 例如，你可以從路由中介軟體檢查當前路由名稱：

```
/**
 * 處理傳入請求。
  * @param  \Illuminate\Http\Request  $request
  * @param  \Closure  $next
  * @return mixed
 */
public function handle(Request $request, Closure $next): Response
{
    if ($request->route()->named('profile')) {
        // ...
    }

    return $next($request);
}
```

5.4 路由組

路由組允許你共享路由屬性，例如中介軟體，而無需在每個單獨的路由上定義這些屬性。

巢狀組嘗試智能地將屬性與其父組 “合併”。中介軟體和 `where` 條件合併，同時附加名稱和前綴。 URI 前綴中的命名空間分隔符和斜槓會在適當的地方自動新增。

5.4.1 路由中介軟體

要將 中介軟體 分配給組內的所有路由，你可以在定義組之前使用 `middleware` 方法。 中介軟體按照它們在陣列中列出的順序執行：

```
Route::middleware(['first', 'second'])->group(function () {
    Route::get('/', function () {
        // 使用第一個和第二個中介軟體。。。
    });

    Route::get('/user/profile', function () {
        // 使用第一個和第二個中介軟體。。。
    });
});
```

5.4.2 controller

如果一組路由都使用相同的 controller, 你可以使用 `controller` 方法為組內的所有路由定義公共 controller 。然後，在定義路由時，你只需要提供它們呼叫的 controller 方法：

```
use App\Http\Controllers\OrderController;

Route::controller(OrderController::class)->group(function () {
    Route::get('/orders/{id}', 'show');
    Route::post('/orders', 'store');
});
```

5.4.3 子域路由

路由組也可以用來處理子域路由。子域可以像路由 uri 一樣被分配路由參數，允許你捕獲子域的一部分以便在路由或 controller 中使用。子域可以在定義組之前呼叫 `domain` 方法來指定:

```
Route::domain('{account}.example.com')->group(function () {
    Route::get('user/{id}', function (string $account, string $id) {
        // ...
    });
});
```

> 注意：為了確保子域路由是可以訪問的，你應該在註冊根域路由之前註冊子域路由。這將防止根域路由覆蓋具有相同 URI 路徑的子域路由。

5.4.4 路由前綴

`prefix` 方法可以用給定的 URI 為組中的每個路由做前綴。例如，你可能想要在組內的所有路由 uri 前面加上 `admin` 前綴：

```
Route::prefix('admin')->group(function () {
    Route::get('/users', function () {
        // 對應 "/admin/users" 的 URL
    });
});
```

5.4.5 路由名稱前綴

`name` 方法可以用給定字串作為組中的每個路由名的前綴。例如，你可能想要用 `admin` 作為所有分組路由的前綴。因為給定字串的前綴與指定的路由名完全一致，所以我們一定要提供末尾 `.` 字元在前綴中：

```
Route::name('admin.')->group(function () {
    Route::get('/users', function () {
        // 被分配的路由名為："admin.users"
    })->name('users');
});
```

5.5 路由模型繫結

將模型 ID 注入到路由或 controller 操作時，你通常會查詢資料庫以檢索與該 ID 對應的模型。Laravel 路由模型繫結提供了一種方便的方法來自動將模型實例直接注入到你的路由中。例如，你可以注入與給定 ID 匹配的整個 `User` 模型實例，而不是注入使用者的 ID。

5.5.1 隱式繫結

Laravel 自動解析定義在路由或 controller 操作中的 Eloquent 模型，其類型提示的變數名稱與路由段名稱匹配。例如：

```
use App\Models\User;

Route::get('/users/{user}', function (User $user) {
    return $user->email;
});
```

由於 `$user` 變數被類型提示為 `App\Models\User` Eloquent 模型，並且變數名稱與 `{user}` URI 段匹配，Laravel 將自動注入 ID 匹配相應的模型實例 來自請求 URI 的值。如果在資料庫中沒有找到匹配的模型實例，將自動生成 404 HTTP 響應。

當然，使用 controller 方法時也可以使用隱式繫結。同樣，請注意 `{user}` URI 段與 controller 中的 `$user` 變數匹配，該變數包含 `App\Models\User` 類型提示：

```
use App\Http\Controllers\UserController;
use App\Models\User;
```

```
// 路由定義。。。
Route::get('/users/{user}', [UserController::class, 'show']);

// 定義 controller 方法。。。
public function show(User $user)
{
    return view('user.profile', ['user' => $user]);
}
```

5.5.1.1 軟刪除模型

通常，隱式模型繫結不會檢索已 軟刪除 的模型。但是，你可以通過將 `withTrashed` 方法連結到你的路由定義來指示隱式繫結來檢索這些模型：

```
use App\Models\User;

Route::get('/users/{user}', function (User $user) {
    return $user->email;
})->withTrashed();
```

5.5.1.2 自訂金鑰

有時你可能希望使用 `id` 外的列來解析 Eloquent 模型。為此，你可以在路由參數定義中指定列：

```
use App\Models\Post;

Route::get('/posts/{post:slug}', function (Post $post) {
    return $post;
});
```

如果你希望模型繫結在檢索給定模型類時始終使用 `id` 以外的資料庫列，則可以覆蓋 Eloquent 模型上的 `getRouteKeyName` 方法：

```
/**
 * 獲取模型的路線金鑰。
 */
public function getRouteKeyName(): string
{
    return 'slug';
}
```

5.5.1.3 自訂鍵和範圍

當在單個路由定義中隱式繫結多個 Eloquent 模型時，你可能希望限定第二個 Eloquent 模型的範圍，使其必須是前一個 Eloquent 模型的子模型。例如，考慮這個通過 slug 為特定使用者檢索部落格文章的路由定義：

```
use App\Models\Post;
use App\Models\User;

Route::get('/users/{user}/posts/{post:slug}', function (User $user, Post $post) {
    return $post;
});
```

當使用自訂鍵控隱式繫結作為巢狀路由參數時，Laravel 將自動限定查詢範圍以通過其父級檢索巢狀模型，使用約定來猜測父級上的關係名稱。 在這種情況下，假設 `User` 模型有一個名為 `posts` 的關係（路由參數名稱的複數形式），可用於檢索 `Post` 模型。

如果你願意，即使未提供自訂鍵，你也可以指示 Laravel 限定「子」繫結的範圍。為此，你可以在定義路由時呼叫 `scopeBindings` 方法：

```
use App\Models\Post;
use App\Models\User;

Route::get('/users/{user}/posts/{post}', function (User $user, Post $post) {
```

```
    return $post;
})->scopeBindings();
```

或者，你可以指示整個路由定義組使用範圍繫結：

```
Route::scopeBindings()->group(function () {
    Route::get('/users/{user}/posts/{post}', function (User $user, Post $post) {
        return $post;
    });
});
```

類似地，你可以通過呼叫 `withoutScopedBindings` 方法來明確的指示 Laravel 不做範疇繫結：

```
Route::get('/users/{user}/posts/{post:slug}', function (User $user, Post $post) {
    return $post;
})->withoutScopedBindings();
```

5.5.1.4 自訂缺失模型行為

通常，如果未找到隱式繫結模型，則會生成 404 HTTP 響應。但是，你可以通過在定義路由時呼叫 `missing` 方法來自訂此行為。`missing` 方法接受一個閉包，如果找不到隱式繫結模型，則將呼叫該閉包：

```
use App\Http\Controllers\LocationsController;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Redirect;

Route::get('/locations/{location:slug}', [LocationsController::class, 'show'])
        ->name('locations.view')
        ->missing(function (Request $request) {
            return Redirect::route('locations.index');
        });
```

5.5.2 隱式列舉繫結

PHP 8.1 引入了對 Enums. 的支援。為了補充這個特性，Laravel 允許你在你的路由定義中鍵入一個 Enums 並且 Laravel 只會在該路由段對應於一個有效的 Enum 值時呼叫該路由。否則，將自動返回 404 HTTP 響應。例如，給定以下列舉：

```
<?php

namespace App\Enums;

enum Category: string
{
    case Fruits = 'fruits';
    case People = 'people';
}
```

你可以定義一個只有在 `{category}` 路由段是 `fruits` 或 `people` 時才會被呼叫的路由。 否則，Laravel 將返回 404 HTTP 響應：

```
use App\Enums\Category;
use Illuminate\Support\Facades\Route;

Route::get('/categories/{category}', function (Category $category) {
    return $category->value;
});
```

5.5.3 顯式繫結

不需要使用 Laravel 隱式的、基於約定的模型解析來使用模型繫結。你還可以顯式定義路由參數與模型的對應方式。要註冊顯式繫結，請使用路由器的 `model` 方法為給定參數指定類。在 `RouteServiceProvider` 類的 `boot` 方法的開頭定義顯式模型繫結：

```
use App\Models\User;
use Illuminate\Support\Facades\Route;

/**
 * 定義路由模型繫結、模式篩選器等。
 */
public function boot(): void
{
    Route::model('user', User::class);

    // ...
}
```

接下來，定義一個包含 {user} 參數的路由：

```
use App\Models\User;

Route::get('/users/{user}', function (User $user) {
    // ...
});
```

由於我們已將所有 {user} 參數繫結到 App\Models\User 模型，該類的一個實例將被注入到路由中。因此，例如，對 users/1 的請求將從 ID 為 1 的資料庫中注入 User 實例。

如果在資料庫中沒有找到匹配的模型實例，則會自動生成 404 HTTP 響應。

5.5.3.1 自訂解析邏輯

如果你想定義你自己的模型繫結解析邏輯，你可以使用 Route::bind 方法。傳遞給 bind 方法的閉包將接收 URI 段的值，並應返回應注入路由的類的實例。同樣，這種定製應該在應用程式的 RouteServiceProvider 的 boot 方法中進行：

```
use App\Models\User;
use Illuminate\Support\Facades\Route;

/**
 * 定義路由模型繫結、模式篩選器等。
 */
public function boot(): void
{
    Route::bind('user', function (string $value) {
        return User::where('name', $value)->firstOrFail();
    });

    // ...
}
```

或者，你可以覆蓋 Eloquent 模型上的 resolveRouteBinding 方法。此方法將接收 URI 段的值，並應返回應注入路由的類的實例：

```
/**
 * 檢索繫結值的模型。
 *
 * @param  mixed  $value
 * @param  string|null  $field
 * @return \Illuminate\Database\Eloquent\Model|null
 */
public function resolveRouteBinding($value, $field = null)
{
    return $this->where('name', $value)->firstOrFail();
}
```

如果路由正在使用 implicit binding scoping, 則 resolveChildRouteBinding 方法將用於解析父模型的子繫結：

```
/**
```

```
     * 檢索繫結值的子模型。
     *
     * @param  string  $childType
     * @param  mixed  $value
     * @param  string|null  $field
     * @return \Illuminate\Database\Eloquent\Model|null
     */
    public function resolveChildRouteBinding($childType, $value, $field)
    {
        return parent::resolveChildRouteBinding($childType, $value, $field);
    }
```

5.6 Fallback 路由

使用 `Route::fallback` 方法，你可以定義一個在沒有其他路由匹配傳入請求時將執行的路由。通常，未處理的請求將通過應用程式的異常處理程序自動呈現「404」頁面。但是，由於你通常會在 `routes/web.php` 檔案中定義 fallback 路由，因此 web 中介軟體組中的所有中介軟體都將應用於該路由。你可以根據需要隨意向此路由新增額外的中介軟體：

```
Route::fallback(function () {
    // ...
});
```

注意：Fallback 路由應該始終是你的應用程式註冊的最後一個路由。

5.7 速率限制

5.7.1 定義速率限制器

Laravel 包括功能強大且可定製的限速服務，你可以利用這些服務來限制給定路線或一組路線的流量。首先，你應該定義滿足應用程式需求的速率限制器組態。通常，這應該在應用程式的 `App\Providers\RouteServiceProvider` 類的 `configureRateLimiting` 方法中完成，該類已經包含了一個速率限制器定義，該定義應用於應用程式 `routes/api.php` 檔案中的路由：

```
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\RateLimiter;

/**
 * 為應用程式組態速率限制器。
 */
protected function boot(): void
{
    RateLimiter::for('api', function (Request $request) {
        return Limit::perMinute(60)->by($request->user()?->id ?: $request->ip());
    });

    // ...
}
```

速率限制器是使用 `RateLimiter` 外觀的 `for` 方法定義的。for 方法接受一個速率限制器名稱和一個閉包，該閉包返回應該應用於分配給速率限制器的路由的限制組態。限制組態是 `Illuminate\Cache\RateLimiting\Limit` 類的實例。此類包含有用的「建構器」方法，以便你可以快速定義限制。速率限制器名稱可以是你希望的任何字串：

```
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\RateLimiter;
```

```
/**
 * 為應用程式組態速率限制器。
 */
protected function boot(): void
{
    RateLimiter::for('global', function (Request $request) {
        return Limit::perMinute(1000);
    });

    // ...
}
```

如果傳入的請求超過指定的速率限制，Laravel 將自動返回一個帶有 429 HTTP 狀態碼的響應。如果你想定義自己的響應，應該由速率限制返回，你可以使用 response 方法：

```
RateLimiter::for('global', function (Request $request) {
    return Limit::perMinute(1000)->response(function (Request $request, array $headers)
{
        return response('Custom response...', 429, $headers);
    });
});
```

由於速率限制器回呼接收傳入的 HTTP 請求實例，你可以根據傳入的請求或經過身份驗證的使用者動態建構適當的速率限制：

```
RateLimiter::for('uploads', function (Request $request) {
    return $request->user()->vipCustomer()
                ? Limit::none()
                : Limit::perMinute(100);
});
```

5.7.1.1 分段速率限制

有時你可能希望按某個任意值對速率限制進行分段。例如，你可能希望每個 IP 地址每分鐘允許使用者訪問給定路由 100 次。為此，你可以在建構速率限制時使用 by 方法：

```
RateLimiter::for('uploads', function (Request $request) {
    return $request->user()->vipCustomer()
                ? Limit::none()
                : Limit::perMinute(100)->by($request->ip());
});
```

為了使用另一個示例來說明此功能，我們可以將每個經過身份驗證的使用者 ID 的路由訪問限制為每分鐘 100 次，或者對於訪客來說，每個 IP 地址每分鐘訪問 10 次：

```
RateLimiter::for('uploads', function (Request $request) {
    return $request->user()
                ? Limit::perMinute(100)->by($request->user()->id)
                : Limit::perMinute(10)->by($request->ip());
});
```

5.7.1.2 多個速率限制

如果需要，你可以返回給定速率限制器組態的速率限制陣列。將根據路由在陣列中的放置順序評估每個速率限制：

```
RateLimiter::for('login', function (Request $request) {
    return [
        Limit::perMinute(500),
        Limit::perMinute(3)->by($request->input('email')),
    ];
});
```

5.7.2 將速率限制器附加到路由

可以使用 `throttle` middleware。 將速率限制器附加到路由或路由組。路由中介軟體接受你希望分配給路由的速率限制器的名稱：

```
Route::middleware(['throttle:uploads'])->group(function () {
    Route::post('/audio', function () {
        // ...
    });

    Route::post('/video', function () {
        // ...
    });
});
```

5.7.2.1 使用 Redis 節流

通常，`throttle` 中介軟體對應到 `Illuminate\Routing\Middleware\ThrottleRequests` 類。此對應在應用程式的 HTTP 核心 (App) 中定義。但是，如果你使用 Redis 作為應用程式的快取驅動程式，你可能希望更改此對應以使用 `Illuminate\Routing\Middleware\ThrottleRequestsWithRedis` 類。這個類在使用 Redis 管理速率限制方面更有效：

```
'throttle' => \Illuminate\Routing\Middleware\ThrottleRequestsWithRedis::class,
```

5.8 偽造表單方法

HTML 表單不支援 `PUT` ， `PATCH` 或 `DELETE` 請求。所以，當定義 `PUT` ， `PATCH` 或 `DELETE` 路由用在 HTML 表單時，你將需要一個隱藏的加 `_method` 欄位在表單中。該 `_method` 欄位的值將會與 HTTP 請求一起傳送。

```
<form action="/example" method="POST">
    <input type="hidden" name="_method" value="PUT">
    <input type="hidden" name="_token" value="{{ csrf_token() }}">
</form>
```

為方便起見，你可以使用 `@method` Blade 指令 生成 `_method` 輸入欄位：

```
<form action="/example" method="POST">
    @method('PUT')
    @csrf
</form>
```

5.9 訪問當前路由

你可以使用 `Route Facade` 的 `current`、`currentRouteName` 和 `currentRouteAction` 方法來訪問有關處理傳入請求的路由的資訊：

```
use Illuminate\Support\Facades\Route;

$route = Route::current(); // Illuminate\Routing\Route
$name = Route::currentRouteName(); // string
$action = Route::currentRouteAction(); // string
```

你可以參考 Route facade 的底層類 和 Route 實例 的 API 文件查看路由器和路由類上可用的所有方法。

5.10 跨域資源共享 (CORS)

Laravel 可以使用你組態的值自動響應 CORS `OPTIONS` HTTP 請求。所有 CORS 設定都可以在應用程式的

`config/cors.php` 組態檔案中進行組態。OPTIONS 請求將由默認包含在全域中介軟體堆疊中的 HandleCors middleware 自動處理。你的全域中介軟體堆疊位於應用程式的 HTTP 核心 (`App\Http\Kernel`) 中。

> 技巧：有關 CORS 和 CORS 標頭的更多資訊，請參閱 MDN 關於 CORS 的 Web 文件。

5.11 路由快取

在將應用程式部署到生產環境時，你應該利用 Laravel 的路由快取。使用路由快取將大大減少註冊所有應用程式路由所需的時間。要生成路由快取，請執行 `route:cache` Artisan 命令：

```
php artisan route:cache
```

運行此命令後，你的快取路由檔案將在每個請求上載入。請記住，如果你新增任何新路線，你將需要生成新的路線快取。因此，你應該只在項目部署期間運行 `route:cache` 命令。

你可以使用 `route:clear` 命令清除路由快取：

```
php artisan route:clear
```

6 中介軟體

6.1 介紹

中介軟體提供了一種方便的機制來檢查和過濾進入應用程式的 HTTP 請求。例如，Laravel 包含一個中介軟體，用於驗證應用程式的使用者是否經過身份驗證。如果使用者未通過身份驗證，中介軟體會將使用者重新導向到應用程式的登錄螢幕。 但是，如果使用者通過了身份驗證，中介軟體將允許請求進一步進入應用程式。

除了身份驗證之外，還可以編寫其他中介軟體來執行各種任務。例如，日誌中介軟體可能會將所有傳入請求記錄到你的應用程式。Laravel 框架中包含了幾個中介軟體，包括用於身份驗證和 CSRF 保護的中介軟體。所有這些中介軟體都位於 `app/Http/Middleware` 目錄中。

6.2 定義中介軟體

要建立新的中介軟體，請使用 `make:middleware` Artisan 命令：

```
php artisan make:middleware EnsureTokenIsValid
```

此命令將在你的 `app/Http/Middleware` 目錄中放置一個新的 `EnsureTokenIsValid` 類。在這個中介軟體中，如果提供的 `token` 輸入匹配指定的值，我們將只允許訪問路由。否則，我們會將使用者重新導向回 `home` URI：

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class EnsureTokenIsValid
{
    /**
     * 處理傳入請求。
     *
     * @param  \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\
Response)  $next
     */
    public function handle(Request $request, Closure $next): Response
    {
        if ($request->input('token') !== 'my-secret-token') {
            return redirect('home');
        }

        return $next($request);
    }
}
```

如你所見，如果給定的 `token` 與我們的秘密令牌不匹配，中介軟體將向客戶端返回 HTTP 重新導向； 否則，請求將被進一步傳遞到應用程式中。要將請求更深入地傳遞到應用程式中（允許中介軟體「通過」），你應該使用 `$request` 呼叫 `$next` 回呼。

最好將中介軟體設想為一系列「層」HTTP 請求在到達你的應用程式之前必須通過。每一層都可以檢查請求，甚至完全拒絕它。

> 技巧：所有中介軟體都通過 服務容器 解析，因此你可以在中介軟體的建構函式中鍵入提示你需要的任

何依賴項。

6.2.1.1 中介軟體和響應

當然，中介軟體可以在將請求更深入地傳遞到應用程式之前或之後執行任務。例如，以下中介軟體將在應用程式處理__請求之前__執行一些任務：

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class BeforeMiddleware
{
    public function handle(Request $request, Closure $next): Response
    {
        // 執行操作

        return $next($request);
    }
}
```

但是，此中介軟體將在應用程式處理__請求之後__執行其任務：

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class AfterMiddleware
{
    public function handle(Request $request, Closure $next): Response
    {
        $response = $next($request);

        // 執行操作

        return $response;
    }
}
```

6.3 註冊中介軟體

6.3.1 全域中介軟體

如果你希望在對應用程式的每個 HTTP 請求期間運行中介軟體，請在 `app/Http/Kernel.php` 類的 `$middleware` 屬性中列出中介軟體類。

6.3.2 將中介軟體分配給路由

如果要將中介軟體分配給特定路由，可以在定義路由時呼叫 `middleware` 方法：

```
use App\Http\Middleware\Authenticate;
```

```
Route::get('/profile', function () {
    // ...
})->middleware(Authenticate::class);
```

通過向 `middleware` 方法傳遞一組中介軟體名稱，可以為路由分配多個中介軟體：

```
Route::get('/', function () {
    // ...
})->middleware([First::class, Second::class]);
```

為了方便起見，可以在應用程式的 `app/Http/Kernel.php` 檔案中為中介軟體分配別名。默認情況下，此類的 `$middlewareAliases` 屬性包含 Laravel 中包含的中介軟體的條目。你可以將自己的中介軟體新增到此列表中，並為其分配選擇的別名：

```
// 在 App\Http\Kernel 類中。。。

protected $middlewareAliases = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
    'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
    'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,
];
```

一旦在 HTTP 核心中定義了中介軟體別名，就可以在將中介軟體分配給路由時使用該別名：

```
Route::get('/profile', function () {
    // ...
})->middleware('auth');
```

6.3.2.1 排除中介軟體

當將中介軟體分配給一組路由時，可能偶爾需要防止中介軟體應用於組內的單個路由。可以使用 `withoutMiddleware` 方法完成此操作：

```
use App\Http\Middleware\EnsureTokenIsValid;

Route::middleware([EnsureTokenIsValid::class])->group(function () {
    Route::get('/', function () {
        // ...
    });

    Route::get('/profile', function () {
        // ...
    })->withoutMiddleware([EnsureTokenIsValid::class]);
});
```

還可以從整個 組 路由定義中排除一組給定的中介軟體：

```
use App\Http\Middleware\EnsureTokenIsValid;

Route::withoutMiddleware([EnsureTokenIsValid::class])->group(function () {
    Route::get('/profile', function () {
        // ...
    });
});
```

「withoutMiddleware」方法只能刪除路由中介軟體，不適用於 全域中介軟體。

6.3.3 中介軟體組

有時，你可能希望將多個中介軟體組合在一個鍵下，以使它們更容易分配給路由。你可以使用 HTTP 核心的 `$middlewareGroups` 屬性來完成此操作。

Laravel 包括預定義 帶有 `web` 和 `api` 中介軟體組，其中包含你可能希望應用於 Web 和 API 路由的常見中介軟體。請記住，這些中介軟體組會由應用程式的 `App\Providers\RouteServiceProvider` 服務提供者自動應用於相應的 `web` 和 `api` 路由檔案中的路由：

```
/**
 * 應用程式的路由中介軟體組。
 *
 * @var array
 */
protected $middlewareGroups = [
    'web' => [
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \App\Http\Middleware\VerifyCsrfToken::class,
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],

    'api' => [
        \Illuminate\Routing\Middleware\ThrottleRequests::class.':api',
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],
];
```

中介軟體組可以使用與單個中介軟體相同的語法分配給路由和 controller 動作。同理，中介軟體組使一次將多個中介軟體分配給一個路由更加方便：

```
Route::get('/', function () {
    // ...
})->middleware('web');

Route::middleware(['web'])->group(function () {
    // ...
});
```

> 技巧：開箱即用，`web` 和 `api` 中介軟體組會通過 `App\Providers\RouteServiceProvider` 自動應用於應用程式對應的 `routes/web.php` 和 `routes/api.php` 檔案。

6.3.4 排序中介軟體

在特定情況下，可能需要中介軟體以特定的順序執行，但當它們被分配到路由時，是無法控制它們的順序的。在這種情況下，可以使用到 `app/Http/Kernel.php` 檔案的 `$middlewarePriority` 屬性指定中介軟體優先順序。默認情況下，HTTP 核心中可能不存在此屬性。如果它不存在，你可以複製下面的默認定義：

```
/**
 * 中介軟體的優先順序排序列表。
 *
 * 這迫使非全域中介軟體始終處於給定的順序。
 *
 * @var string[]
 */
protected $middlewarePriority = [
    \Illuminate\Foundation\Http\Middleware\HandlePrecognitiveRequests::class,
    \Illuminate\Cookie\Middleware\EncryptCookies::class,
    \Illuminate\Session\Middleware\StartSession::class,
```

```
    \Illuminate\View\Middleware\ShareErrorsFromSession::class,
    \Illuminate\Contracts\Auth\Middleware\AuthenticatesRequests::class,
    \Illuminate\Routing\Middleware\ThrottleRequests::class,
    \Illuminate\Routing\Middleware\ThrottleRequestsWithRedis::class,
    \Illuminate\Contracts\Session\Middleware\AuthenticatesSessions::class,
    \Illuminate\Routing\Middleware\SubstituteBindings::class,
    \Illuminate\Auth\Middleware\Authorize::class,
];
```

6.4 中介軟體參數

中介軟體也可以接收額外的參數。例如，如果你的應用程式需要在執行給定操作之前驗證經過身份驗證的使用者是否具有給定的「角色」，你可以建立一個 `EnsureUserHasRole` 中介軟體，該中介軟體接收角色名稱作為附加參數。

額外的中介軟體參數將在 `$next` 參數之後傳遞給中介軟體：

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class EnsureUserHasRole
{
    /**
     * 處理傳入請求。
     *
     * @param  \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\
Response)  $next
     */
    public function handle(Request $request, Closure $next, string $role): Response
    {
        if (! $request->user()->hasRole($role)) {
            // 重新導向。。。
        }

        return $next($request);
    }

}
```

在定義路由時，可以指定中介軟體參數，方法是使用 : 分隔中介軟體名稱和參數。多個參數應以逗號分隔：

```
Route::put('/post/{id}', function (string $id) {
    // ...
})->middleware('role:editor');
```

6.5 可終止的中介軟體

部分情況下，在將 HTTP 響應傳送到瀏覽器之後，中介軟體可能需要做一些工作。如果你在中介軟體上定義了一個 `terminate` 方法，並且你的 Web 伺服器使用 FastCGI，則在將響應傳送到瀏覽器後會自動呼叫 `terminate` 方法：

```
<?php

namespace Illuminate\Session\Middleware;

use Closure;
use Illuminate\Http\Request;
```

```
use Symfony\Component\HttpFoundation\Response;

class TerminatingMiddleware
{
    /**
     * 處理傳入的請求。
     *
     * @param  \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\
Response)  $next
     */
    public function handle(Request $request, Closure $next): Response
    {
        return $next($request);
    }

    /**
     * 在響應傳送到瀏覽器後處理任務。
     */
    public function terminate(Request $request, Response $response): void
    {
        // ...
    }
}
```

`terminate` 方法應該同時接收請求和響應。一旦你定義了一個可終止的中介軟體，你應該將它新增到 `app/Http/Kernel.php` 檔案中的路由或全域中介軟體列表中。

當在中介軟體上呼叫 `terminate` 方法時，Laravel 會從 服務容器 解析一個新的中介軟體實例。如果你想在呼叫 `handle` 和 `terminate` 方法時使用相同的中介軟體實例，請使用容器的 `singleton` 方法向容器註冊中介軟體。 通常這應該在你的 `AppServiceProvider` 的 `register` 方法中完成：

```
use App\Http\Middleware\TerminatingMiddleware;

/**
 * 註冊任何應用程式服務。
 */
public function register(): void
{
    $this->app->singleton(TerminatingMiddleware::class);
}
```

7 CSRF 保護

7.1 簡介

跨站點請求偽造是一種惡意利用，利用這種手段，代表經過身份驗證的使用者執行未經授權的命令。值得慶幸的是，Laravel 可以輕鬆保護您的應用程式免受跨站點請求偽造（CSRF）攻擊。

7.1.1.1 漏洞的解釋

如果你不熟悉跨站點請求偽造，我們討論一個利用此漏洞的示例。假設您的應用程式有一個 `/user/email` 路由，它接受 POST 請求來更改經過身份驗證使用者的電子郵件地址。最有可能的情況是，此路由希望 `email` 輸入欄位包含使用者希望開始使用的電子郵件地址。

沒有 CSRF 保護，惡意網站可能會建立一個 HTML 表單，指向您的應用程式 `/user/email` 路由，並提交惡意使用者自己的電子郵件地址：

```
<form action="https://your-application.com/user/email" method="POST">
    <input type="email" value="malicious-email@example.com">
</form>

<script>
    document.forms[0].submit();
</script>
```

如果惡意網站在頁面載入時自動提交了表單，則惡意使用者只需要誘使您的應用程式的一個毫無戒心的使用者訪問他們的網站，他們的電子郵件地址就會在您的應用程式中更改。

為了防止這種漏洞，我們需要檢查每一個傳入的 `POST`，`PUT`，`PATCH` 或 `DELETE` 請求以獲取惡意應用程式無法訪問的秘密 session 值。

7.2 阻止 CSRF 請求

Laravel 為應用程式管理的每個活動 使用者 session 自動生成 CSRF 「令牌」。此令牌用於驗證經過身份驗證的使用者是實際嚮應用程序發出請求的人。由於此令牌儲存在使用者的 session 中，並且每次重新生成 session 時都會更改，因此惡意應用程式將無法訪問它。

當前 session 的 CSRF 令牌可以通過請求的 session 或通過 `csrf_token` 輔助函數進行訪問：

```
use Illuminate\Http\Request;

Route::get('/token', function (Request $request) {
    $token = $request->session()->token();

    $token = csrf_token();

    // ...
});
```

無論何時在應用程式中定義 `POST` 、`PUT` 、`PATCH` 或 `DELETE` HTML 表單，都應在表單中包含隱藏的 CSRF `_token` 欄位，以便 CSRF 保護中介軟體可以驗證請求。為方便起見，可以使用 `@csrf` Blade 指令生成隱藏的令牌輸入欄位：

```
<form method="POST" action="/profile">
    @csrf
```

```
    <!-- 相當於。。。 -->
    <input type="hidden" name="_token" value="{{ csrf_token() }}" />
</form>
```

默認情況下包含在 web 中介軟體組中的 App\Http\Middleware\VerifyCsrfToken 中介軟體將自動驗證請求輸入中的令牌是否與 session 中儲存的令牌匹配。當這兩個令牌匹配時，我們知道身份驗證的使用者是發起請求的使用者。

7.2.1 CSRF Tokens & SPAs

如果你正在建構一個將 Laravel 用作 API 後端的 SPA，你應該查閱 Laravel Sanctum 文件，以獲取有關使用 API 進行身份驗證和防範 CSRF 漏洞的資訊。

7.2.2 從 CSRF 保護中排除 URI

有時你可能希望從 CSRF 保護中排除一組 URIs。例如，如果你使用 Stripe 處理付款並使用他們的 webhook 系統，則需要將你的 Stripe webhook 處理程序路由從 CSRF 保護中排除，因為 Stripe 不會知道要向您的路由傳送什麼 CSRF 令牌。

通常，你應該將這些類型的路由放在 App\Providers\RouteServiceProvider 應用於 routes/web.php 檔案中的所有路由的 web 中介軟體組之外。但是，現在也可以通過將路由的 URIs 新增到 VerifyCsrfToken 中介軟體的 $except 屬性來排除路由：

```
<?php

namespace App\Http\Middleware;

use Illuminate\Foundation\Http\Middleware\VerifyCsrfToken as Middleware;

class VerifyCsrfToken extends Middleware
{
    /**
     * 從 CSRF 驗證中排除的 URIs。
     *
     * @var array
     */
    protected $except = [
        'stripe/*',
        'http://example.com/foo/bar',
        'http://example.com/foo/*',
    ];
}
```

技巧：為方便起見，運行測試時自動停用所有路由的 CSRF 中介軟體。

7.3 X-CSRF-TOKEN

除了檢查 CSRF 令牌作為 POST 參數外， App\Http\Middleware\VerifyCsrfToken 中介軟體還將檢查 X-CSRF-TOKEN 請求標頭。 例如，你可以將令牌儲存在 HTML 的 meta 標籤中：

```
<meta name="csrf-token" content="{{ csrf_token() }}">
```

然後，你可以指示 jQuery 之類的庫自動將令牌新增到所有請求標頭。 這為使用傳統 JavaScript 技術的基於 AJAX 的應用程式提供了簡單、方便的 CSRF 保護：

```
$.ajaxSetup({
    headers: {
        'X-CSRF-TOKEN': $('meta[name="csrf-token"]').attr('content')
    }
```

```
});
```

7.4 X-XSRF-TOKEN

Laravel 將當前 CSRF 令牌儲存在加密的 `XSRF-TOKEN` cookie 中，該 cookie 包含在框架生成的每個響應中。您可以使用 cookie 值設定 `X-XSRF-TOKEN` 請求標頭。

由於一些 JavaScript 框架和庫（如 Angular 和 Axios ）會自動將其值放置在同一源請求的 `X-XSRF-TOKEN` 標頭中，因此傳送此 cookie 主要是為了方便開發人員。

> 技巧：默認情況下，`resources/js/bootstrap.js` 檔案包含 Axios HTTP 庫，它會自動為您傳送 `X-XSRF-TOKEN` 標頭。

8 Controller

8.1 介紹

你可能希望使用「controller」類來組織此行為，而不是將所有請求處理邏輯定義為路由檔案中的閉包。 controller 可以將相關的請求處理邏輯分組到一個類中。 例如，一個 `UserController` 類可能會處理所有與使用者相關的傳入請求，包括顯示、建立、更新和刪除使用者。 默認情況下， controller 儲存在 `app/Http/Controllers` 目錄中。

8.2 編寫 controller

8.2.1 基本 controller

如果要快速生成新 controller ，可以使用 `make:controller` Artisan 命令。默認情況下，應用程式的所有 controller 都儲存在 `app/Http/Controllers` 目錄中：

```
php artisan make:controller UserController
```

讓我們來看一個基本 controller 的示例。 controller 可以有任意數量的公共方法來響應傳入的 HTTP 請求：

```
<?php

namespace App\Http\Controllers;

use App\Models\User;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * 顯示給定使用者的組態檔案。
     */
    public function show(string $id): View
    {
        return view('user.profile', [
            'user' => User::findOrFail($id)
        ]);
    }
}
```

編寫 controller 類和方法後，可以定義到 controller 方法的路由，如下所示：

```
use App\Http\Controllers\UserController;

Route::get('/user/{id}', [UserController::class, 'show']);
```

當傳入的請求與指定的路由 URI 匹配時，將呼叫 `App\Http\Controllers\UserController` 類的 `show` 方法，並將路由參數傳遞給該方法。

> 技巧： controller 並不是 **必需** 繼承基礎類。如果 controller 沒有繼承基礎類，你將無法使用一些便捷的功能，比如 `middleware` 和 `authorize` 方法。

8.2.2 單動作 controller

如果 controller 動作特別複雜，你可能會發現將整個 controller 類專用於該單個動作很方便。為此，您可以在

controller 中定義一個 `__invoke` 方法：

```
<?php

namespace App\Http\Controllers;

use App\Models\User;
use Illuminate\Http\Response;

class ProvisionServer extends Controller
{
    /**
     * 設定新的 web 伺服器。
     */
    public function __invoke()
    {
        // ...
    }
}
```

為單動作 controller 註冊路由時，不需要指定 controller 方法。相反，你可以簡單地將 controller 的名稱傳遞給路由器：

```
use App\Http\Controllers\ProvisionServer;

Route::post('/server', ProvisionServer::class);
```

你可以使用 `make:controller` Artisan 命令的 `--invokable` 選項生成可呼叫 controller ：

```
php artisan make:controller ProvisionServer --invokable
```

> 技巧：可以使用 stub 定製 自訂 controller 範本。

8.3 controller 中介軟體

中介軟體 可以在你的路由檔案中分配給 controller 的路由：

```
Route::get('profile', [UserController::class, 'show'])->middleware('auth');
```

或者，你可能會發現在 controller 的建構函式中指定中介軟體很方便。使用 controller 建構函式中的 `middleware` 方法，你可以將中介軟體分配給 controller 的操作：

```
class UserController extends Controller
{
    /**
     * Instantiate a new controller instance.
     */
    public function __construct()
    {
        $this->middleware('auth');
        $this->middleware('log')->only('index');
        $this->middleware('subscribed')->except('store');
    }
}
```

controller 還允許你使用閉包註冊中介軟體。這提供了一種方便的方法來為單個 controller 定義內聯中介軟體，而無需定義整個中介軟體類：

```
use Closure;
use Illuminate\Http\Request;

$this->middleware(function (Request $request, Closure $next) {
    return $next($request);
});
```

8.4 資源型 controller

如果你將應用程式中的每個 Eloquent 模型都視為資源，那麼通常對應用程式中的每個資源都執行相同的操作。例如，假設你的應用程式中包含一個 `Photo` 模型和一個 `Movie` 模型。使用者可能可以建立，讀取，更新或者刪除這些資源。

Laravel 的資源路由通過單行程式碼即可將典型的增刪改查（“CURD”）路由分配給 controller 。首先，我們可以使用 Artisan 命令 `make:controller` 的 `--resource` 選項來快速建立一個 controller :

```
php artisan make:controller PhotoController --resource
```

這個命令將會生成一個 controller `app/Http/Controllers/PhotoController.php`。其中包括每個可用資源操作的方法。接下來，你可以給 controller 註冊一個資源路由：

```
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class);
```

這個單一的路由聲明建立了多個路由來處理資源上的各種行為。生成的 controller 為每個行為保留了方法，而且你可以通過運行 Artisan 命令 `route:list` 來快速瞭解你的應用程式。

你可以通過將陣列傳參到 `resources` 方法中的方式來一次性的建立多個資源 controller ：

```
Route::resources([
    'photos' => PhotoController::class,
    'posts' => PostController::class,
]);
```

8.4.1.1 資源 controller 操作處理

請求方式	請求 URI	行為	路由名稱
GET	`/photos`	index	photos.index
GET	`/photos/create`	create	photos.create
POST	`/photos`	store	photos.store
GET	`/photos/{photo}`	show	photos.show
GET	`/photos/{photo}/edit`	edit	photos.edit
PUT/PATCH	`/photos/{photo}`	update	photos.update
DELETE	`/photos/{photo}`	destroy	photos.destroy

8.4.1.2 自訂缺失模型行為

通常，如果未找到隱式繫結的資源模型，則會生成狀態碼為 404 的 HTTP 響應。 但是，你可以通過在定義資源路由時呼叫 `missing` 的方法來自訂該行為。`missing` 方法接受一個閉包，如果對於任何資源的路由都找不到隱式繫結模型，則將呼叫該閉包：

```
use App\Http\Controllers\PhotoController;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Redirect;

Route::resource('photos', PhotoController::class)
        ->missing(function (Request $request) {
            return Redirect::route('photos.index');
        });
```

8.4.1.3 軟刪除模型

通常情況下，隱式模型繫結將不會檢索已經進行了 軟刪除 的模型，並且會返回一個 404 HTTP 響應。但是，你可以在定義資源路由時呼叫 `withTrashed` 方法來告訴框架允許軟刪除的模型：

```
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class)->withTrashed();
```

當不傳遞參數呼叫 `withTrashed` 時，將在 `show`、`edit` 和 `update` 資源路由中允許軟刪除的模型。你可以通過一個陣列指定這些路由的子集傳遞給 `withTrashed` 方法：

```
Route::resource('photos', PhotoController::class)->withTrashed(['show']);
```

8.4.1.4 指定資源模型

如果你使用了路由模型的繫結 路由模型繫結 並且想在資源 controller 的方法中使用類型提示，你可以在生成 controller 的時候使用 `--model` 選項：

```
php artisan make:controller PhotoController --model=Photo --resource
```

8.4.1.5 生成表單請求

你可以在生成資源 controller 時提供 `--requests` 選項來讓 Artisan 為 controller 的 storage 和 update 方法生成 表單請求類：

```
php artisan make:controller PhotoController --model=Photo --resource --requests
```

8.4.2 部分資源路由

當聲明資源路由時，你可以指定 controller 處理的部分行為，而不是所有默認的行為：

```
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class)->only([
    'index', 'show'
]);

Route::resource('photos', PhotoController::class)->except([
    'create', 'store', 'update', 'destroy'
]);
```

8.4.2.1 API 資源路由

當聲明用於 API 的資源路由時，通常需要排除顯示 HTML 範本的路由，例如 `create` 和 `edit`。為了方便，你可以使用 `apiResource` 方法來排除這兩個路由：

```
use App\Http\Controllers\PhotoController;

Route::apiResource('photos', PhotoController::class);
```

你也可以傳遞一個陣列給 `apiResources` 方法來同時註冊多個 API 資源 controller ：

```
use App\Http\Controllers\PhotoController;
use App\Http\Controllers\PostController;

Route::apiResources([
    'photos' => PhotoController::class,
    'posts' => PostController::class,
]);
```

要快速生成不包含 `create` 或 `edit` 方法的 API 資源 controller ，你可以在執行 `make:controller` 命令時使用 `--api` 參數：

```
php artisan make:controller PhotoController --api
```

8.4.3 巢狀資源

有時可能需要定義一個巢狀的資源型路由。例如，照片資源可能被新增了多個評論。那麼可以在路由中使用 `.` 符號來聲明資源型 controller ：

```
use App\Http\Controllers\PhotoCommentController;
```

```
Route::resource('photos.comments', PhotoCommentController::class);
```

該路由會註冊一個巢狀資源，可以使用如下 URI 訪問：

```
/photos/{photo}/comments/{comment}
```

8.4.3.1 限定巢狀資源的範圍

Laravel 的 隱式模型繫結 特性可以自動限定巢狀繫結的範圍，以便確認已解析的子模型會自動屬於父模型。定義巢狀路由時，使用 scoped 方法，可以開啟自動範圍限定，也可以指定 Laravel 應該按照哪個欄位檢索子模型資源，有關如何完成此操作的更多資訊，請參見有關 範圍資源路由 的文件。

8.4.3.2 淺層巢狀

通常，並不是在所有情況下都需要在 URI 中同時擁有父 ID 和子 ID，因為子 ID 已經是唯一的識別碼。當使用唯一識別碼（如自動遞增的主鍵）來標識 URL 中的模型時，可以選擇使用「淺巢狀」的方式定義路由：

```
use App\Http\Controllers\CommentController;

Route::resource('photos.comments', CommentController::class)->shallow();
```

上面的路由定義方式會定義以下路由：

請求方式	請求 URI	行為	路由名稱
GET	/photos/{photo}/comments	index	photos.comments.index
GET	/photos/{photo}/comments/create	create	photos.comments.create
POST	/photos/{photo}/comments	store	photos.comments.store
GET	/comments/{comment}	show	comments.show
GET	/comments/{comment}/edit	edit	comments.edit
PUT/PATCH	/comments/{comment}	update	comments.update
DELETE	/comments/{comment}	destroy	comments.destroy

8.4.4 命名資源路由

默認情況下，所有的資源 controller 行為都有一個路由名稱。你可以傳入 names 陣列來覆蓋這些名稱：

```
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class)->names([
    'create' => 'photos.build'
]);
```

8.4.5 命名資源路由參數

默認情況下，Route::resource 會根據資源名稱的「單數」形式建立資源路由的路由參數。你可以使用 parameters 方法來輕鬆地覆蓋資源路由名稱。傳入 parameters 方法應該是資源名稱和參數名稱的關聯陣列：

```
use App\Http\Controllers\AdminUserController;

Route::resource('users', AdminUserController::class)->parameters([
    'users' => 'admin_user'
]);
```

上面的示例將會為資源的 show 路由生成以下的 URL：

```
/users/{admin_user}
```

8.4.6 限定範圍的資源路由

Laravel 的 範疇隱式模型繫結 功能可以自動確定巢狀繫結的範圍，以便確認已解析的子模型屬於父模型。通過在定義巢狀資源時使用 scoped 方法，你可以啟用自動範圍界定，並指示 Laravel 應該通過以下方式來檢索子資源的哪個欄位：

```
use App\Http\Controllers\PhotoCommentController;

Route::resource('photos.comments', PhotoCommentController::class)->scoped([
    'comment' => 'slug',
]);
```

此路由將註冊一個有範圍的巢狀資源，該資源可以通過以下 URI 進行訪問：

```
/photos/{photo}/comments/{comment:slug}
```

當使用一個自訂鍵的隱式繫結作為巢狀路由參數時，Laravel 會自動限定查詢範圍，按照約定的命名方式去父類中尋找關聯方法，然後檢索到對應的巢狀模型。在這種情況下，將假定 Photo 模型有一個叫 comments （路由參數名的複數）的關聯方法，通過這個方法可以檢索到 Comment 模型。

8.4.7 本地化資源 URIs

默認情況下，Route::resource 將會用英文動詞建立資源 URIs。如果需要自訂 create 和 edit 行為的動名詞，你可以在 App\Providers\RouteServiceProvider 的 boot 方法中使用 Route::resourceVerbs 方法實現：

```
/**
 * 定義你的路由模型繫結，模式過濾器等
 */
public function boot(): void
{
    Route::resourceVerbs([
        'create' => 'crear',
        'edit' => 'editar',
    ]);

    // ...
}
```

Laravel 的複數器支援組態幾種不同的語言。自訂動詞和複數語言後，諸如 Route::resource('publicacion', PublicacionController::class) 之類的資源路由註冊將生成以下 URI：

```
/publicacion/crear

/publicacion/{publicaciones}/editar
```

8.4.8 補充資源 controller

如果你需要向資源 controller 新增超出默認資源路由集的其他路由，則應在呼叫 Route::resource 方法之前定義這些路由；否則，由 resource 方法定義的路由可能會無意中優先於您的補充路由： 單例資源 use App;

```
Route::get('/photos/popular', [PhotoController::class, 'popular']);
Route::resource('photos', PhotoController::class);
```

> 技巧：請記住讓你的 controller 保持集中。如果你發現自己經常需要典型資源操作集之外的方法，請考慮將 controller 拆分為兩個更小的 controller 。

8.4.9 單例資源 controller

有時候，應用中的資源可能只有一個實例。比如，使用者「個人資料」可被編輯或更新，但是一個使用者只會有一份「個人資料」。同樣，一張圖片也只有一個「縮圖」。這些資源就是所謂「單例資源」，這意味著該資源有且只能有一個實例存在。這種情況下，你可以註冊成單例(signleton)資源 controller :

```
use App\Http\Controllers\ProfileController;
use Illuminate\Support\Facades\Route;

Route::singleton('profile', ProfileController::class);
```

上例中定義的單例資源會註冊如下所示的路由。如你所見，單例資源中「新建」路由沒有被註冊；並且註冊的路由不接收路由參數，因為該資源中只有一個實例存在：

請求方式	請求 URI	行為	路由名稱
GET	`/profile`	show	profile.show
GET	`/profile/edit`	edit	profile.edit
PUT/PATCH	`/profile`	update	profile.update

單例資源也可以在標準資源內巢狀使用：

```
Route::singleton('photos.thumbnail', ThumbnailController::class);
```

上例中， `photo` 資源將接收所有的標準資源路由；不過，`thumbnail` 資源將會是個單例資源，它的路由如下所示：

請求方式	請求 URI	行為	路由名稱
GET	`/photos/{photo}/thumbnail`	show	photos.thumbnail.show
GET	`/photos/{photo}/thumbnail/edit`	edit	photos.thumbnail.edit
PUT/PATCH	`/photos/{photo}/thumbnail`	update	photos.thumbnail.update

8.4.9.1 Creatable 單例資源

有時，你可能需要為單例資源定義 create 和 storage 路由。要實現這一功能，你可以在註冊單例資源路由時，呼叫 `creatable` 方法：

```
Route::singleton('photos.thumbnail', ThumbnailController::class)->creatable();
```

如下所示，將註冊以下路由。還為可建立的單例資源註冊 `DELETE` 路由：

Verb	URI	Action	Route Name
GET	`/photos/{photo}/thumbnail/create`	create	photos.thumbnail.create
POST	`/photos/{photo}/thumbnail`	store	photos.thumbnail.store
GET	`/photos/{photo}/thumbnail`	show	photos.thumbnail.show
GET	`/photos/{photo}/thumbnail/edit`	edit	photos.thumbnail.edit
PUT/PATCH	`/photos/{photo}/thumbnail`	update	photos.thumbnail.update
DELETE	`/photos/{photo}/thumbnail`	destroy	photos.thumbnail.destroy

如果希望 Laravel 為單個資源註冊 `DELETE` 路由，但不註冊建立或儲存路由，則可以使用 `destroyable` 方法：

```
Route::singleton(...)->destroyable();
```

8.4.9.2 API 單例資源

`apiSingleton` 方法可用於註冊將通過 API 操作的單例資源，從而不需要 `create` 和 `edit` 路由：

```
Route::apiSingleton('profile', ProfileController::class);
```

當然， API 單例資源也可以是 可建立的 ，它將註冊 `store` 和 `destroy` 資源路由：

```
Route::apiSingleton('photos.thumbnail', ProfileController::class)->creatable();
```

8.5 依賴注入和 controller

8.5.1.1 建構函式注入

Laravel 服務容器 用於解析所有 Laravel controller 。因此，可以在其建構函式中對 controller 可能需要的任何依賴項進行類型提示。聲明的依賴項將自動解析並注入到 controller 實例中：

```
<?php

namespace App\Http\Controllers;

use App\Repositories\UserRepository;

class UserController extends Controller
{
    /**
     * 建立新 controller 實例。
     */
    public function __construct(
        protected UserRepository $users,
    ) {}
}
```

8.5.1.2 方法注入

除了建構函式注入，還可以在 controller 的方法上鍵入提示依賴項。方法注入的一個常見用例是將 `Illuminate\Http\Request` 實例注入到 controller 方法中：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * 儲存新使用者。
     */
    public function store(Request $request): RedirectResponse
    {
        $name = $request->name;

        // 儲存使用者。。。

        return redirect('/users');
    }
}
```

如果 controller 方法也需要路由參數，那就在其他依賴項之後列出路由參數。例如，路由是這樣定義的：

```
use App\Http\Controllers\UserController;

Route::put('/user/{id}', [UserController::class, 'update']);
```

如下所示，你依然可以類型提示 `Illuminate\Http\Request` 並通過定義您的 controller 方法訪問 `id` 參數：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
```

```
class UserController extends Controller
{
    /**
     * 更新給定使用者。
     */
    public function update(Request $request, string $id): RedirectResponse
    {
        // 更新使用者。。。

        return redirect('/users');
    }
}
```

9 HTTP 請求

9.1 介紹

Laravel 的 `Illuminate\Http\Request` 類提供了一種物件導向的方式來與當前由應用程式處理的 HTTP 請求進行互動，並檢索提交請求的輸入內容、Cookie 和檔案。

9.2 與請求互動

9.2.1 訪問請求

要通過依賴注入獲取當前的 HTTP 請求實例，您應該在路由閉包或 controller 方法中匯入 `Illuminate\Http\Request` 類。傳入的請求實例將由 Laravel 服務容器 自動注入：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class UserController extends Controller
{
     /**
      * 儲存新使用者。
      */
     public function store(Request $request): RedirectResponse
     {
          $name = $request->input('name');

          // 儲存使用者……

          return redirect('/users');
     }
}
```

如上所述，您也可以在路由閉包上匯入 `Illuminate\Http\Request` 類。服務容器將在執行時自動將傳入請求注入閉包中：

```
use Illuminate\Http\Request;

Route::get('/', function (Request $request) {
     // ...
});
```

9.2.1.1 依賴注入和路由參數

如果您的 controller 方法還需要從路由參數中獲取輸入，則應該在其他依賴項之後列出路由參數。例如，如果您的路由定義如下：

```
use App\Http\Controllers\UserController;

Route::put('/user/{id}', [UserController::class, 'update']);
```

您仍然可以在 controller 方法中使用類型提示的 `Illuminate\Http\Request` 並通過以下方式訪問您的 `id` 路

由參數來定義您的 controller 方法：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Update the specified user.
     */
    public function update(Request $request, string $id): RedirectResponse
    {
        // 更新使用者...

        return redirect('/users');
    }
}
```

9.2.2 請求路徑、主機和方法

`Illuminate\Http\Request` 實例提供各種方法來檢查傳入的 HTTP 請求，並擴展了 `Symfony\Component\HttpFoundation\Request` 類。下面我們將討論一些最重要的方法。

9.2.2.1 獲取請求路徑

`path` 方法返回請求的路徑資訊。因此，如果傳入的請求針對 `http://example.com/foo/bar`，則 `path` 方法將返回 `foo/bar`：

```
$uri = $request->path();
```

9.2.2.2 檢查請求路徑/路由資訊

`is` 方法允許您驗證傳入請求路徑是否與給定的模式匹配。當使用此方法時，您可以使用 * 字元作為萬用字元：

```
if ($request->is('admin/*')) {
    // ...
}
```

使用 `routeIs` 方法，您可以確定傳入的請求是否與 命名路由 匹配：

```
if ($request->routeIs('admin.*')) {
    // ...
}
```

9.2.2.3 獲取請求 URL

要獲取傳入請求的完整 URL，您可以使用 `url` 或 `fullUrl` 方法。`url` 方法將返回不帶查詢字串的 URL，而 `fullUrl` 方法將包括查詢字串：

```
$url = $request->url();

$urlWithQueryString = $request->fullUrl();
```

如果您想將查詢字串資料附加到當前 URL，請呼叫 `fullUrlWithQuery` 方法。此方法將給定的查詢字串變數陣列與當前查詢字串合併：

```
$request->fullUrlWithQuery(['type' => 'phone']);
```

9.2.2.4 獲取請求 Host

您可以通過 host、httpHost 和 schemeAndHttpHost 方法獲取傳入請求的 「host」：

```
$request->host();
$request->httpHost();
$request->schemeAndHttpHost();
```

9.2.2.5 獲取請求方法

method 方法將返回請求的 HTTP 動詞。您可以使用 isMethod 方法來驗證 HTTP 動詞是否與給定的字串匹配：

```
$method = $request->method();

if ($request->isMethod('post')) {
    // ...
}
```

9.2.3 要求標頭

您可以使用 header 方法從 Illuminate\Http\Request 實例中檢索請求標頭。如果請求中沒有該標頭，則返回 null。但是，header 方法接受兩個可選參數，如果該標頭在請求中不存在，則返回第二個參數：

```
$value = $request->header('X-Header-Name');

$value = $request->header('X-Header-Name', 'default');
```

hasHeader 方法可用於確定請求是否包含給定的標頭：

```
if ($request->hasHeader('X-Header-Name')) {
    // ...
}
```

為了方便起見，bearerToken 方法可用於從 Authorization 標頭檢索授權標記。如果不存在此類標頭，將返回一個空字串：

```
$token = $request->bearerToken();
```

9.2.4 請求 IP 地址

ip 方法可用於檢索向您的應用程式發出請求的客戶端的 IP 地址：

```
$ipAddress = $request->ip();
```

9.2.5 內容協商

Laravel 提供了幾種方法，通過 Accept 標頭檢查傳入請求的請求內容類型。首
先，getAcceptableContentTypes 方法將返回包含請求接受的所有內容類型的陣列：

```
$contentTypes = $request->getAcceptableContentTypes();
```

accepts 方法接受一個內容類型陣列，並在請求接受任何內容類型時返回 true。否則，將返回 false：

```
if ($request->accepts(['text/html', 'application/json'])) {
    // ...
}
```

您可以使用 prefers 方法確定給定內容類型陣列中的哪種內容類型由請求最具優勢。如果請求未接受任何提供的內容類型，則返回 null：

```
$preferred = $request->prefers(['text/html', 'application/json']);
```

由於許多應用程式僅提供 HTML 或 JSON，因此您可以使用 `expectsJson` 方法快速確定傳入請求是否期望獲得 JSON 響應：

```
if ($request->expectsJson()) {
    // ...
}
```

9.2.6 PSR-7 請求

PSR-7 標準 指定了 HTTP 消息的介面，包括請求和響應。如果您想要獲取 PSR-7 請求的實例而不是 Laravel 請求，您首先需要安裝一些庫。Laravel 使用 *Symfony HTTP Message Bridge* 元件將典型的 Laravel 請求和響應轉換為 PSR-7 相容的實現：

```
composer require symfony/psr-http-message-bridge
composer require nyholm/psr7
```

安裝這些庫之後，您可以通過在路由閉包或 controller 方法上的請求介面進行類型提示來獲取 PSR-7 請求：

```
use Psr\Http\Message\ServerRequestInterface;

Route::get('/', function (ServerRequestInterface $request) {
    // ...
});
```

> **注意** 如果您從路由或 controller 返回 PSR-7 響應實例，它將自動轉換回 Laravel 響應實例，並由框架顯示。

9.3 輸入

9.3.1 檢索輸入

9.3.1.1 檢索所有輸入資料

您可以使用 `all` 方法將所有傳入請求的輸入資料作為 `array` 檢索。無論傳入請求是否來自 HTML 表單或 XHR 請求，都可以使用此方法：

```
$input = $request->all();
```

使用 `collect` 方法，您可以將所有傳入請求的輸入資料作為 集合 檢索：

```
$input = $request->collect();
```

`collect` 方法還允許您將傳入請求的子集作為集合檢索：

```
$request->collect('users')->each(function (string $user) {
    // ...
});
```

9.3.1.2 檢索輸入值

使用幾個簡單的方法，無論請求使用了哪種 HTTP 動詞，都可以從您的 `Illuminate\Http\Request` 實例訪問所有使用者輸入。`input` 方法可用於檢索使用者輸入：

```
$name = $request->input('name');
```

您可以將預設值作為第二個參數傳遞給 `input` 方法。如果請求中不存在所請求的輸入值，則返回此值：

```
$name = $request->input('name', 'Sally');
```

處理包含陣列輸入的表單時，請使用「.」符號訪問陣列：

```
$name = $request->input('products.0.name');

$names = $request->input('products.*.name');
```

您可以呼叫不帶任何參數的 input 方法，以將所有輸入值作為關聯陣列檢索出來：

```
$input = $request->input();
```

9.3.1.3 從查詢字串檢索輸入

雖然 input 方法從整個請求消息載荷（包括查詢字串）檢索值，但 query 方法僅從查詢字串檢索值：

```
$name = $request->query('name');
```

如果請求的查詢字串值資料不存在，則將返回此方法的第二個參數：

```
$name = $request->query('name', 'Helen');
```

您可以呼叫不帶任何參數的 query 方法，以將所有查詢字串值作為關聯陣列檢索出來：

```
$query = $request->query();
```

9.3.1.4 檢索 JSON 輸入值

當向您的應用程式傳送 JSON 請求時，只要請求的 Content-Type 標頭正確設定為 application/json，您就可以通過 input 方法訪問 JSON 資料。您甚至可以使用「.」語法來檢索巢狀在 JSON 陣列/對象中的值：

```
$name = $request->input('user.name');
```

9.3.1.5 檢索可字串化的輸入值

您可以使用 string 方法將請求的輸入資料檢索為 Illuminate\Support\Stringable 的實例，而不是將其作為基本 string 檢索：

```
$name = $request->string('name')->trim();
```

9.3.1.6 檢索布林值輸入

處理類似複選框的 HTML 元素時，您的應用程式可能會接收到實際上是字串的「true」。例如，「true」或「on」。為了方便起見，您可以使用 boolean 方法將這些值作為布林值檢索。boolean 方法對於 1，「1」，true，「true」，「on」和「yes」，返回 true。所有其他值將返回 false：

```
$archived = $request->boolean('archived');
```

9.3.1.7 檢索日期輸入值

為了方便起見，包含日期 / 時間的輸入值可以使用 date 方法檢索為 Carbon 實例。如果請求中不包含給定名稱的輸入值，則返回 null：

```
$birthday = $request->date('birthday');
```

date 方法可接受的第二個和第三個參數可用於分別指定日期的格式和時區：

```
$elapsed = $request->date('elapsed', '!H:i', 'Europe/Madrid');
```

如果輸入值存在但格式無效，則會拋出一個 InvalidArgumentException 異常；因此，在呼叫 date 方法之前建議對輸入進行驗證。

9.3.1.8 檢索列舉輸入值

還可以從請求中檢索對應於 PHP 列舉 的輸入值。如果請求中不包含給定名稱的輸入值或列舉沒有與輸入值匹配的備份值，則返回 null。enum 方法接受輸入值的名稱和列舉類作為其第一個和第二個參數：

```
use App\Enums\Status;

$status = $request->enum('status', Status::class);
```

9.3.1.9 通過動態屬性檢索輸入

您也可以使用 `Illuminate\Http\Request` 實例上的動態屬性訪問使用者輸入。例如，如果您的應用程式的表單之一包含一個 `name` 欄位，則可以像這樣訪問該欄位的值：

```
$name = $request->name;
```

使用動態屬性時，Laravel 首先會在請求負載中尋找參數的值，如果不存在，則會在匹配路由的參數中搜尋該欄位。

9.3.1.10 檢索輸入資料的一部分

如果您需要檢索輸入資料的子集，則可以使用 `only` 和 `except` 方法。這兩個方法都接受一個單一的 `array` 或動態參數列表：

```
$input = $request->only(['username', 'password']);

$input = $request->only('username', 'password');

$input = $request->except(['credit_card']);

$input = $request->except('credit_card');
```

> **警告** `only` 方法返回您請求的所有鍵 / 值對；但是，它不會返回請求中不存在的鍵 / 值對。

9.3.2 判斷輸入是否存在

您可以使用 `has` 方法來確定請求中是否存在某個值。如果請求中存在該值則 `has` 方法返回 `true`：

```
if ($request->has('name')) {
    // ...
}
```

當給定一個陣列時，`has` 方法將確定所有指定的值是否都存在：

```
if ($request->has(['name', 'email'])) {
    // ...
}
```

`whenHas` 方法將在請求中存在一個值時執行給定的閉包：

```
$request->whenHas('name', function (string $input) {
    // ...
});
```

可以通過向 `whenHas` 方法傳遞第二個閉包來執行，在請求中沒有指定值的情況下：

```
$request->whenHas('name', function (string $input) {
    // "name" 值存在...
}, function () {
    // "name" 值不存在...
});
```

`hasAny` 方法返回 `true`，如果任一指定的值存在，則它返回 `true`：

```
if ($request->hasAny(['name', 'email'])) {
    // ...
}
```

如果您想要確定請求中是否存在一個值且不是一個空字串，則可以使用 `filled` 方法：

```
if ($request->filled('name')) {
```

```
    // ...
}
```

whenFilled 方法將在請求中存在一個值且不是空字串時執行給定的閉包：

```
$request->whenFilled('name', function (string $input) {
    // ...
});
```

可以通過向 whenFilled 方法傳遞第二個閉包來執行，在請求中沒有指定值的情況下：

```
$request->whenFilled('name', function (string $input) {
    // "name" 值已填寫...
}, function () {
    // "name" 值未填寫...
});
```

要確定給定的鍵是否不存在於請求中，可以使用 missing 和 whenMissing 方法：

```
if ($request->missing('name')) {
    // ...
}

$request->whenMissing('name', function (array $input) {
    // "name" 值缺失...
}, function () {
    // "name" 值存在...
});
```

9.3.3 合併其他輸入

有時，您可能需要手動將其他輸入合併到請求的現有輸入資料中。為此，可以使用 merge 方法。如果給定的輸入鍵已經存在於請求中，它將被提供給 merge 方法的資料所覆蓋：

```
$request->merge(['votes' => 0]);
```

如果請求的輸入資料中不存在相應的鍵，則可以使用 mergeIfMissing 方法將輸入合併到請求中：

```
$request->mergeIfMissing(['votes' => 0]);
```

9.3.4 舊輸入

Laravel 允許你在兩次請求之間保留資料。這個特性在檢測到驗證錯誤後重新填充表單時特別有用。但是，如果您使用 Laravel 的包含的 表單驗證，不需要自己手動呼叫這些方法，因為 Laravel 的一些內建驗證功能將自動呼叫它們。

9.3.4.1 快閃記憶體輸入到 Session

在 Illuminate\Http\Request 類上的 flash 方法將當前輸入快閃記憶體到 session，以便在下一次使用者請求應用程式時使用：

```
$request->flash();
```

您還可以使用 flashOnly 和 flashExcept 方法快閃記憶體一部分請求資料到 Session。這些方法對於將敏感資訊（如密碼）排除在 Session 外的情況下非常有用：

```
$request->flashOnly(['username', 'email']);

$request->flashExcept('password');
```

9.3.4.2 快閃記憶體輸入後重新導向

由於您通常希望快閃記憶體輸入到 Session，然後重新導向到以前的頁面，因此您可以使用 withInput 方法輕

鬆地將輸入快閃記憶體到重新導向中：

```
return redirect('form')->withInput();

return redirect()->route('user.create')->withInput();

return redirect('form')->withInput(
    $request->except('password')
);
```

9.3.4.3 檢索舊輸入值

若要獲取上一次請求所保存的舊輸入資料，可以在 `Illuminate\Http\Request` 的實例上呼叫 `old` 方法。`old` 方法會從 session 中檢索先前快閃記憶體的輸入資料：

```
$username = $request->old('username');
```

此外，Laravel 還提供了一個全域輔助函數 `old`。如果您在 Blade 範本 中顯示舊的輸入，則更方便使用 `old` 輔助函數重新填充表單。如果給定欄位沒有舊輸入，則會返回 `null`：

```
<input type="text" name="username" value="{{ old('username') }}">
```

9.3.5 Cookies

9.3.5.1 檢索請求中的 Cookies

Laravel 框架建立的所有 cookies 都經過加密並簽名，這意味著如果客戶端更改了 cookie 值，則這些 cookie 將被視為無效。要從請求中檢索 cookie 值，請在 `Illuminate\Http\Request` 實例上使用 `cookie` 方法：

```
$value = $request->cookie('name');
```

9.4 輸入過濾和規範化

默認情況下，Laravel 在應用程式的全域中介軟體棧中包含 `App\Http\Middleware\TrimStrings` 和 `Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull` 中介軟體。這些中介軟體在 `App\Http\Kernel` 類的全域中介軟體棧中列出。這些中介軟體將自動修剪請求中的所有字串欄位，並將任何空字串欄位轉換為 `null`。這使您不必在路由和 controller 中擔心這些規範化問題。

9.4.1.1 停用輸入規範化

如果要停用所有請求的該行為，可以從 `App\Http\Kernel` 類的 `$middleware` 屬性中刪除這兩個中介軟體，從而將它們從應用程式的中介軟體棧中刪除。

如果您想要停用應用程式的一部分請求的字串修剪和空字串轉換，可以使用中介軟體提供的 `skipWhen` 方法。該方法接受一個閉包，該閉包應返回 `true` 或 `false`，以指示是否應跳過輸入規範化。通常情況下，需要在應用程式的 `AppServiceProvider` 的 `boot` 方法中呼叫 `skipWhen` 方法。

```
use App\Http\Middleware\TrimStrings;
use Illuminate\Http\Request;
use Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    TrimStrings::skipWhen(function (Request $request) {
        return $request->is('admin/*');
    });
```

```
    ConvertEmptyStringsToNull::skipWhen(function (Request $request) {
        // ...
    });
}
```

9.5 檔案

9.5.1 檢索上傳的檔案

您可以使用 `file` 方法或動態屬性從 `Illuminate\Http\Request` 實例中檢索已上傳的檔案。`file` 方法返回 `Illuminate\Http\UploadedFile` 類的實例，該類擴展了 PHP 的 `SplFileInfo` 類，並提供了各種用於與檔案互動的方法：

```
$file = $request->file('photo');

$file = $request->photo;
```

您可以使用 `hasFile` 方法檢查請求中是否存在檔案：

```
if ($request->hasFile('photo')) {
    // ...
}
```

9.5.1.1 驗證成功上傳的檔案

除了檢查檔案是否存在之外，您還可以通過 `isValid` 方法驗證上傳檔案時是否存在問題：

```
if ($request->file('photo')->isValid()) {
    // ...
}
```

9.5.1.2 檔案路徑和擴展名

`UploadedFile` 類還包含訪問檔案的完全限定路徑及其擴展名的方法。`extension` 方法將嘗試基於其內容猜測檔案的擴展名。此擴展名可能與客戶端提供的擴展名不同：

```
$path = $request->photo->path();

$extension = $request->photo->extension();
```

9.5.1.3 其他檔案方法

`UploadedFile` 實例有許多其他可用的方法。有關這些方法的更多資訊，請查看該類的 API 文件 。

9.5.2 儲存上傳的檔案

要儲存已上傳的檔案，通常會使用您組態的一個檔案系統 。`UploadedFile` 類具有一個 `store` 方法，該方法將上傳的檔案移動到您的磁碟中的一個位置，該位置可以是本地檔案系統上的位置，也可以是像 Amazon S3 這樣的雲端儲存位置。

`store` 方法接受儲存檔案的路徑，該路徑相對於檔案系統的組態根目錄。此路徑不應包含檔案名稱，因為將自動生成唯一的 ID 作為檔案名稱。

`store` 方法還接受一個可選的第二個參數，用於指定應用於儲存檔案的磁碟的名稱。該方法將返回相對於磁碟根目錄的檔案路徑：

```
$path = $request->photo->store('images');

$path = $request->photo->store('images', 's3');
```

如果您不希望自動生成檔案名稱，則可以使用 `storeAs` 方法，該方法接受路徑、檔案名稱和磁碟名稱作為其參數：

```
$path = $request->photo->storeAs('images', 'filename.jpg');

$path = $request->photo->storeAs('images', 'filename.jpg', 's3');
```

> **注意** 有關在 Laravel 中儲存檔案的更多資訊，請查看完整的 檔案儲存文件 。

9.6 組態受信任的代理

在終止 TLS / SSL 證書的負載平衡器後面運行應用程式時，您可能會注意到，使用 `url` 幫助程序時，應用程式有時不會生成 HTTPS 連結。通常，這是因為正在從連接埠 `80` 上的負載平衡器轉發應用程式的流量，並且不知道它應該生成安全連結。

為瞭解決這個問題，您可以使用 `App\Http\Middleware\TrustProxies` 中介軟體，這個中介軟體已經包含在 Laravel 應用程式中，它允許您快速定製應用程式應信任的負載平衡器或代理。您信任的代理應該被列在此中介軟體的 `$proxies` 屬性上的陣列中。除了組態受信任的代理之外，您還可以組態應該信任的代理 `$headers`：

```
<?php

namespace App\Http\Middleware;

use Illuminate\Http\Middleware\TrustProxies as Middleware;
use Illuminate\Http\Request;

class TrustProxies extends Middleware
{
    /**
     * 此應用程式的受信任代理。
     *
     * @var string|array
     */
    protected $proxies = [
        '192.168.1.1',
        '192.168.1.2',
    ];

    /**
     * 應用於檢測代理的標頭。
     *
     * @var int
     */
    protected $headers = Request::HEADER_X_FORWARDED_FOR |
Request::HEADER_X_FORWARDED_HOST | Request::HEADER_X_FORWARDED_PORT |
Request::HEADER_X_FORWARDED_PROTO;
}
```

> **注意** 如果您正在使用 AWS 彈性負載平衡，請將 `$headers` 值設定為 `Request::HEADER_X_FORWARDED_AWS_ELB`。有關可在 `$headers` 屬性中使用的常數的更多資訊，請查看 Symfony 關於 信任代理 的文件。

9.6.1.1 信任所有代理

如果您使用的是 Amazon AWS 或其他「雲」負載平衡器提供商，則可能不知道實際負載平衡器的 IP 地址。在這種情況下，您可以使用 * 來信任所有代理：

```
/**
 * 應用所信任的代理。
 *
 * @var string|array
 */
protected $proxies = '*';
```

9.7 組態可信任的 Host

默認情況下，Laravel 將響應它接收到的所有請求，而不管 HTTP 請求的 `Host` 標頭的內容是什麼。此外，在 web 請求期間生成應用程式的絕對 URL 時，將使用 `Host` 頭的值。

通常情況下，您應該組態您的 Web 伺服器（如 Nginx 或 Apache）僅向匹配給定主機名的應用程式傳送請求。然而，如果您沒有直接自訂您的 Web 伺服器的能力，需要指示 Laravel 僅響應特定主機名的請求，您可以為您的應用程式啟用 `App\Http\Middleware\TrustHosts` 中介軟體。

`TrustHosts` 中介軟體已經包含在應用程式的 `$middleware` 堆疊中；但是，您應該將其取消註釋以使其生效。在此中介軟體的 `hosts` 方法中，您可以指定您的應用程式應該響應的主機名。具有其他 `Host` 值標頭的傳入請求將被拒絕：

```
/**
 * 獲取應被信任的主機模式。
 *
 * @return array<int, string>
 */
public function hosts(): array
{
    return [
        'laravel.test',
        $this->allSubdomainsOfApplicationUrl(),
    ];
}
```

`allSubdomainsOfApplicationUrl` 幫助程序方法將返回與您的應用程式 `app.url` 組態值的所有子域相匹配的正規表示式。在建構利用萬用字元子域的應用程式時，這個幫助程序提供了一種方便的方法來允許所有應用程式的子域。

10 HTTP 響應

10.1 建立響應

10.1.1.1 字串 & 陣列

所有路由和 controller 處理完業務邏輯之後都會返迴響應到使用者的瀏覽器，Laravel 提供了多種不同的響應方式，其中最基本就是從路由或 controller 返回一個簡單的字串，框架會自動將這個字串轉化為一個完整的 HTTP 響應：

```
Route::get('/', function () {
    return 'Hello World';
});
```

除了從路由和 controller 返回字串之外，你還可以返回陣列。 框架會自動將陣列轉換為 JSON 響應：

```
Route::get('/', function () {
    return [1, 2, 3];
});
```

> **技巧**
> 你知道從路由或 controller 還可以返回 Eloquent 集合嗎？他們也會自動轉化為 JSON 響應！

10.1.1.2 Response 對象

通常情況下會只返回簡單的字串或陣列，大多數時候，需要返回一個完整的 `Illuminate\Http\Response` 實例或是檢視.

返回一個完整的 `Response` 實例允許你自訂返回的 HTTP 狀態碼和返回頭資訊。`Response` 實例繼承自 `Symfony\Component\HttpFoundation\Response` 類，該類提供了各種建構 HTTP 響應的方法：

```
Route::get('/home', function () {
    return response('Hello World', 200)
                  ->header('Content-Type', 'text/plain');
});
```

10.1.1.3 Eloquent 模型 和 集合

你也可以直接從你的路由和 controller 返回 Eloquent ORM 模型和集合。當你這樣做時，Laravel 將自動將模型和集合轉換為 JSON 響應，同時遵循模型的 隱藏屬性:

```
use App\Models\User;

Route::get('/user/{user}', function (User $user) {
    return $user;
});
```

10.1.2 在響應中附加 Header 資訊

請記住，大多數響應方法都是可以鏈式呼叫的，它允許你流暢地建構響應實例。例如，在將響應傳送回使用者之前，可以使用 `header` 方法將一系列頭新增到響應中：

```
return response($content)
             ->header('Content-Type', $type)
             ->header('X-Header-One', 'Header Value')
```

```
            ->header('X-Header-Two', 'Header Value');
```

或者，你可以使用 `withHeaders` 方法指定要新增到響應的標頭陣列：

```
return response($content)
            ->withHeaders([
                'Content-Type' => $type,
                'X-Header-One' => 'Header Value',
                'X-Header-Two' => 'Header Value',
            ]);
```

10.1.2.1 快取控制中介軟體

Laravel 包含一個 `cache.headers` 中介軟體，可用於快速設定一組路由的 `Cache-Control` 標頭。指令應使用相應快取控制指令的 蛇形命名法 等效項提供，並應以分號分隔。如果在指令列表中指定了 `etag` ，則響應內容的 MD5 雜湊將自動設定為 ETag 識別碼：

```
Route::middleware('cache.headers:public;max_age=2628000;etag')->group(function () {
    Route::get('/privacy', function () {
        // ...
    });

    Route::get('/terms', function () {
        // ...
    });
});
```

10.1.3 在響應中附加 Cookie 資訊

可以使用 `cookie` 方法將 cookie 附加到傳出的 `illumize\Http\Response` 實例。你應將 cookie 的名稱、值和有效分鐘數傳遞給此方法：

```
return response('Hello World')->cookie(
    'name', 'value', $minutes
);
```

`cookie` 方法還接受一些使用頻率較低的參數。通常，這些參數的目的和意義與 PHP 的原生 setcookie 的參數相同

```
return response('Hello World')->cookie(
    'name', 'value', $minutes, $path, $domain, $secure, $httpOnly
);
```

如果你希望確保 cookie 與傳出響應一起傳送，但你還沒有該響應的實例，則可以使用 `Cookie` facade 將 cookie 加入佇列，以便在傳送響應時附加到響應中。`queue` 方法接受建立 cookie 實例所需的參數。在傳送到瀏覽器之前，這些 cookies 將附加到傳出的響應中：

```
use Illuminate\Support\Facades\Cookie;

Cookie::queue('name', 'value', $minutes);
```

10.1.3.1 生成 Cookie 實例

如果要生成一個 `Symfony\Component\HttpFoundation\Cookie` 實例，打算稍後附加到響應實例中，你可以使用全域 `cookie` 助手函數。此 cookie 將不會傳送回客戶端，除非它被附加到響應實例中：

```
$cookie = cookie('name', 'value', $minutes);

return response('Hello World')->cookie($cookie);
```

10.1.3.2 提前過期 Cookies

你可以通過響應中的 `withoutCookie` 方法使 cookie 過期，用於刪除 cookie ：

```
return response('Hello World')->withoutCookie('name');
```

如果尚未有建立響應的實例，則可以使用 `Cookie` facade 中的 `expire` 方法使 Cookie 過期：

```
Cookie::expire('name');
```

10.1.4 Cookies 和 加密

默認情況下，由 Laravel 生成的所有 cookie 都經過了加密和簽名，因此客戶端無法篡改或讀取它們。如果要對應用程式生成的部分 cookie 停用加密，可以使用 `App\Http\Middleware\EncryptCookies` 中介軟體的 `$except` 屬性，該屬性位於 `app/Http/Middleware` 目錄中：

```
/**
 * 這個名字的 Cookie 將不會加密。
 *
 * @var array
 */
protected $except = [
    'cookie_name',
];
```

10.2 重新導向

重新導向響應是 `Illuminate\Http\RedirectResponse` 類的實例，包含將使用者重新導向到另一個 URL 所需的適當 HTTP 頭。Laravel 有幾種方法可以生成 `RedirectResponse` 實例。最簡單的方法是使用全域 `redirect` 助手函數：

```
Route::get('/dashboard', function () {
    return redirect('home/dashboard');
});
```

有時你可能希望將使用者重新導向到以前的位置，例如當提交的表單無效時。你可以使用全域 back 助手函數來執行此操作。由於此功能使用 session，請確保呼叫 `back` 函數的路由使用的是 `web` 中介軟體組：

```
Route::post('/user/profile', function () {
    // 驗證請求參數

    return back()->withInput();
});
```

10.2.1 重新導向到指定名稱的路由

當你在沒有傳遞參數的情況下呼叫 `redirect` 助手函數時，將返回 `Illuminate\Routing\Redirector` 的實例，允許你呼叫 `Redirector` 實例上的任何方法。例如，要對命名路由生成 `RedirectResponse` ，可以使用 `route` 方法：

```
return redirect()->route('login');
```

如果路由中有參數，可以將其作為第二個參數傳遞給 `route` 方法：

```
// 對於具有以下 URI 的路由: /profile/{id}

return redirect()->route('profile', ['id' => 1]);
```

10.2.1.1 通過 Eloquent 模型填充參數

如果你要重新導向到使用從 Eloquent 模型填充 「ID」 參數的路由，可以直接傳遞模型本身。ID 將會被自動提取：

```
// 對於具有以下 URI 的路由: /profile/{id}

return redirect()->route('profile', [$user]);
```

如果你想要自訂路由參數，你可以指定路由參數 (`/profile/{id:slug}`) 或者重寫 Eloquent 模型上的 `getRouteKey` 方法：

```
/**
 * 獲取模型的路由鍵值。
 */
public function getRouteKey(): mixed
{
    return $this->slug;
}
```

10.2.2 重新導向到 controller 行為

也可以生成重新導向到 controller actions。只要把 controller 和 action 的名稱傳遞給 `action` 方法：

```
use App\Http\Controllers\UserController;

return redirect()->action([UserController::class, 'index']);
```

如果 controller 路由有參數，可以將其作為第二個參數傳遞給 `action` 方法：

```
return redirect()->action(
    [UserController::class, 'profile'], ['id' => 1]
);
```

10.2.3 重新導向到外部域名

有時候你需要重新導向到應用外的域名。可以通過呼叫 `away` 方法，它會建立一個不帶有任何額外的 URL 編碼、有效性總和檢查碼檢查 `RedirectResponse` 實例：

```
return redirect()->away('https://www.google.com');
```

10.2.4 重新導向並使用快閃記憶體的 Session 資料

重新導向到新的 URL 的同時傳送資料給 seesion 是很常見的。 通常這是在你將消息傳送到 session 後成功執行操作後完成的。為了方便，你可以建立一個 `RedirectResponse` 實例並在鏈式方法呼叫中將資料傳送給 session：

```
Route::post('/user/profile', function () {
    // ...

    return redirect('dashboard')->with('status', 'Profile updated!');
});
```

在使用者重新導向後，你可以顯示 session。例如，你可以使用 Blade 範本語法：

```
@if (session('status'))
    <div class="alert alert-success">
        {{ session('status') }}
    </div>
@endif
```

10.2.4.1 使用輸入重新導向

你可以使用 RedirectResponse 實例提供的 withInput 方法將當前請求輸入的資料傳送到 session ，然後再將使用者重新導向到新位置。當使用者遇到驗證錯誤時，通常會執行此操作。每當輸入資料被傳送到 session , 你可以很簡單的在下一次重新提交的表單請求中取回它：

```
return back()->withInput();
```

10.3 其他響應類型

response 助手可用於生成其他類型的響應實例。當不帶參數呼叫 response 助手時，會返回 Illuminate\Contracts\Routing\ResponseFactory contract 的實現。 該契約提供了幾種有用的方法來生成響應。

10.3.1 響應檢視

如果你需要控制響應的狀態和標頭，但還需要返回 view 作為響應的內容，你應該使用 view 方法：

```
return response()
              ->view('hello', $data, 200)
              ->header('Content-Type', $type);
```

當然，如果你不需要傳遞自訂 HTTP 狀態程式碼或自訂標頭，則可以使用全域 view 輔助函數。

10.3.2 JSON Responses

json 方法會自動將 Content-Type 標頭設定為 application/json，並使用 json_encode PHP 函數將給定的陣列轉換為 JSON：

```
return response()->json([
    'name' => 'Abigail',
    'state' => 'CA',
]);
```

如果你想建立一個 JSONP 響應，你可以結合使用 json 方法和 withCallback 方法：

```
return response()
              ->json(['name' => 'Abigail', 'state' => 'CA'])
              ->withCallback($request->input('callback'));
```

10.3.3 檔案下載

download 方法可用於生成強制使用者瀏覽器在給定路徑下載檔案的響應。download 方法接受檔案名稱作為該方法的第二個參數，這將確定下載檔案的使用者看到的檔案名稱。 最後，你可以將一組 HTTP 標頭作為該方法的第三個參數傳遞：

```
return response()->download($pathToFile);

return response()->download($pathToFile, $name, $headers);
```

> 注意：管理檔案下載的 Symfony HttpFoundation 要求正在下載的檔案具有 ASCII 檔案名稱。

10.3.3.1 流式下載

有時你可能希望將給定操作的字串響應轉換為可下載的響應，而不必將操作的內容寫入磁碟。在這種情況下，你可以使用 streamDownload 方法。此方法接受回呼、檔案名稱和可選的標頭陣列作為其參數：

```
use App\Services\GitHub;
```

```
return response()->streamDownload(function () {
    echo GitHub::api('repo')
                ->contents()
                ->readme('laravel', 'laravel')['contents'];
}, 'laravel-readme.md');
```

10.3.4 檔案響應

`file` 方法可用於直接在使用者的瀏覽器中顯示檔案，例如圖像或 PDF，而不是啟動下載。這個方法接受檔案的路徑作為它的第一個參數和一個頭陣列作為它的第二個參數：

```
return response()->file($pathToFile);

return response()->file($pathToFile, $headers);
```

10.4 響應宏

如果你想定義一個可以在各種路由和 controller 中重複使用的自訂響應，你可以使用 `Response` facade 上的 `macro` 方法。通常，你應該從應用程式的服務提供者，如 `App\Providers\AppServiceProvider` 服務提供程序的 `boot` 方法呼叫此方法：

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Response;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * 啟動一個應用的服務
     */
    public function boot(): void
    {
        Response::macro('caps', function (string $value) {
            return Response::make(strtoupper($value));
        });
    }
}
```

`macro` 函數接受名稱作為其第一個參數，並接受閉包作為其第二個參數。當從 `ResponseFactory` 實現或 `response` 助手函數呼叫宏名稱時，將執行宏的閉包：

```
return response()->caps('foo');
```

11 檢視

11.1 介紹

當然，直接從路由和 controller 返回整個 HTML 文件字串是不切實際的。值得慶幸的是，檢視提供了一種方便的方式來將我們所有的 HTML 放在單獨的檔案中。

檢視將你的 controller / 應用程式邏輯與你的表示邏輯分開並儲存在 `resources/views` 目錄中。一個簡單的檢視可能看起來像這樣：使用 Laravel 時，檢視範本通常使用 Blade 範本語言 編寫。一個簡單的檢視如下所示：

```
<!-- 檢視儲存在 `resources/views/greeting.blade.php` -->

<html>
    <body>
        <h1>Hello, {{ $name }}</h1>
    </body>
</html>
```

將上述程式碼儲存到 `resources/views/greeting.blade.php` 後，我們可以使用全域輔助函數 `view` 將其返回，例如：

```
Route::get('/', function () {
    return view('greeting', ['name' => 'James']);
});
```

> 技巧：如果你想瞭解更多關於如何編寫 Blade 範本的更多資訊？查看完整的 Blade 文件 將是最好的開始。

11.1.1 在 React / Vue 中編寫檢視

許多開發人員已經開始傾向於使用 React 或 Vue 編寫範本，而不是通過 Blade 在 PHP 中編寫前端範本。Laravel 讓這件事不痛不癢，這要歸功於 慣性，這是一個庫，可以輕鬆地將 React / Vue 前端連接到 Laravel 後端，而無需建構 SPA 的典型複雜性。

我們的 Breeze 和 Jetstream starter kits 為你提供了一個很好的起點，用 Inertia 驅動你的下一個 Laravel 應用程式。此外，Laravel Bootcamp 提供了一個完整的演示，展示如何建構一個由 Inertia 驅動的 Laravel 應用程式，包括 Vue 和 React 的示例。

11.2 建立和渲染檢視

你可以通過在應用程式 `resources/views` 目錄中放置具有 `.blade.php` 擴展名的檔案來建立檢視。該 `.blade.php` 擴展通知框架該檔案包含一個 Blade 範本。Blade 範本包含 HTML 和 Blade 指令，允許你輕鬆地回顯值、建立「if」語句、迭代資料等。

建立檢視後，可以使用全域 `view` 從應用程式的某個路由或 controller 返回檢視：

```
Route::get('/', function () {
    return view('greeting', ['name' => 'James']);
});
```

也可以使用 `View` 檢視門面（Facade）：

```
use Illuminate\Support\Facades\View;
```

```
return View::make('greeting', ['name' => 'James']);
```

如上所示，傳遞給 view 的第一個參數對應於 resources/views 目錄中檢視檔案的名稱。第二個參數是應該對檢視可用的資料陣列。在這種情況下，我們傳遞 name 變數，它使用 Blade 語法顯示在檢視中。

11.2.1 巢狀檢視目錄

檢視也可以巢狀在目錄 resources/views 的子目錄中。「.」符號可用於引用巢狀檢視。例如，如果檢視儲存在 resources/views/admin/profile.blade.php ，你可以從應用程式的路由或 controller 中返回它，如下所示：

```
return view('admin.profile', $data);
```

> 注意：查看目錄名稱不應包含該 . 字元。

11.2.2 建立第一個可用檢視

使用 View 門面的 first 方法，你可以建立給定陣列檢視中第一個存在的檢視。如果你的應用程式或開發的第三方包允許定製或覆蓋檢視，這會非常有用：

```
use Illuminate\Support\Facades\View;

return View::first(['custom.admin', 'admin'], $data);
```

11.2.3 判斷檢視檔案是否存在

如果需要判斷檢視檔案是否存在，可以使用 View 門面。如果檢視存在， exists 方法會返回 true：

```
use Illuminate\Support\Facades\View;

if (View::exists('emails.customer')) {
    // ...
}
```

11.3 向檢視傳遞資料

正如您在前面的示例中看到的，您可以將資料陣列傳遞給檢視，以使該資料可用於檢視：

```
return view('greetings', ['name' => 'Victoria']);
```

以這種方式傳遞資訊時，資料應該是帶有鍵 / 值對的陣列。向檢視提供資料後，您可以使用資料的鍵訪問檢視中的每個值，例如 <?php echo $name; ?>。

作為將完整的資料陣列傳遞給 view 輔助函數的替代方法，你可以使用該 with 方法將單個資料新增到檢視中。該 with 方法返回檢視對象的實例，以便你可以在返回檢視之前繼續連結方法：

```
return view('greeting')
             ->with('name', 'Victoria')
             ->with('occupation', 'Astronaut');
```

11.3.1 與所有檢視共享資料

有時，你可能需要與應用程式呈現的所有檢視共享資料，可以使用 View 門面的 share 。你可以在服務提供器的 boot 方法中呼叫檢視門面（Facade）的 share 。例如，可以將它們新增到 App\Providers\AppServiceProvider 或者為它們生成一個單獨的服務提供器：

```
<?php
```

```
namespace App\Providers;

use Illuminate\Support\Facades\View;

class AppServiceProvider extends ServiceProvider
{
    /**
     * 註冊應用服務.
     */
    public function register(): void
    {
        // ...
    }

    /**
     * 引導任何應用程式服務。
     */
    public function boot(): void
    {
        View::share('key', 'value');
    }
}
```

11.4 查看合成器

檢視合成器是在呈現檢視時呼叫的回呼或類方法。如果每次渲染檢視時都希望將資料繫結到檢視，則檢視合成器可以幫助你將邏輯組織到單個位置。如果同一檢視由應用程式中的多個路由或 controller 返回，並且始終需要特定的資料，檢視合成器或許會特別有用。

通常，檢視合成器將在應用程式的一個 服務提供者 中註冊。在本例中，我們假設我們已經建立了一個新的 `App\Providers\ViewServiceProvider` 來容納此邏輯。

我們將使用 `View` 門面的 `composer` 方法來註冊檢視合成器。 Laravel 不包含基於類的檢視合成器的默認目錄，因此你可以隨意組織它們。例如，可以建立一個 `app/View/Composers` 目錄來存放應用程式的所有檢視合成器：

```
<?php

namespace App\Providers;

use App\View\Composers\ProfileComposer;
use Illuminate\Support\Facades;
use Illuminate\Support\ServiceProvider;
use Illuminate\View\View;

class ViewServiceProvider extends ServiceProvider
{
    /**
     * 註冊任何應用程式服務。
     */
    public function register(): void
    {
        // ...
    }

    /**
     * 引導任何應用程式服務。
     */
    public function boot(): void
    {
        // 使用基於類的合成器。。。
        Facades\View::composer('profile', ProfileComposer::class);
```

```
        // 使用基於閉包的合成器。。。
        Facades\View::composer('welcome', function (View $view) {
            // ...
        });

        Facades\View::composer('dashboard', function (View $view) {
            // ...
        });
    }
}
```

> 注意：請記住，如果建立一個新的服務提供程序來包含檢視合成器註冊，則需要將服務提供程序新增到 `config/app.php` 組態檔案中的 `providers` 陣列中。

現在我們註冊了檢視合成器，每次渲染 `profile` 檢視時都會執行 `App\View\Composers\ProfileComposer` 類的 `compose` 方法。接下來看一個檢視合成器類的例子：

```
<?php

namespace App\View\Composers;

use App\Repositories\UserRepository;
use Illuminate\View\View;

class ProfileComposer
{
    /**
     * 建立新的組態檔案合成器。
     */
    public function __construct(
        protected UserRepository $users,
    ) {}

    /**
     * 將資料繫結到檢視。
     */
    public function compose(View $view): void
    {
        $view->with('count', $this->users->count());
    }
}
```

如上所示，所有的檢視合成器都會通過 服務容器進行解析，所以你可以在檢視合成器的建構函式中類型提示需要注入的依賴項。

11.4.1.1 將檢視合成器新增到多個檢視

你可以通過將檢視陣列作為第一個參數傳遞給 `composer` 方法，可以一次新增多個檢視到檢視合成器中：

```
use App\Views\Composers\MultiComposer;
use Illuminate\Support\Facades\View;

View::composer(
    ['profile', 'dashboard'],
    MultiComposer::class
);
```

該 `composer` 方法同時也接受萬用字元 `*` ，表示將所有檢視新增到檢視合成器中：

```
use Illuminate\Support\Facades;
use Illuminate\View\View;

Facades\View::composer('*', function (View $view) {
    // ...
```

```
});
```

11.4.2 檢視構造器

檢視構造器「creators」和檢視合成器非常相似。唯一不同之處在於檢視構造器在檢視實例化之後執行，而檢視合成器在檢視即將渲染時執行。使用 `creator` 方法註冊檢視構造器：

```
use App\View\Creators\ProfileCreator;
use Illuminate\Support\Facades\View;

View::creator('profile', ProfileCreator::class);
```

11.5 最佳化檢視

默認情況下，Blade 範本檢視是按需編譯的。當執行渲染檢視的請求時，Laravel 將確定檢視的編譯版本是否存在。如果檔案存在，Laravel 將比較未編譯的檢視和已編譯的檢視是否有修改。如果編譯後的檢視不存在，或者未編譯的檢視已被修改，Laravel 將重新編譯該檢視。

在請求期間編譯檢視可能會對性能產生小的負面影響，因此 Laravel 提供了 `view:cache` Artisan 命令來預編譯應用程式使用的所有檢視。為了提高性能，你可能希望在部署過程中運行此命令：

```
php artisan view:cache
```

你可以使用 `view:clear` 命令清除檢視快取：

```
php artisan view:clear
```

12 Blade 範本

12.1 簡介

Blade 是 Laravel 提供的一個簡單而又強大的範本引擎。 和其他流行的 PHP 範本引擎不同，Blade 並不限制你在檢視中使用原生 PHP 程式碼。實際上，所有 Blade 檢視檔案都將被編譯成原生的 PHP 程式碼並快取起來，除非它被修改，否則不會重新編譯，這就意味著 Blade 基本上不會給你的應用增加任何負擔。Blade 範本檔案使用 `.blade.php` 作為副檔名，被存放在 `resources/views` 目錄。

Blade 檢視可以使用全域 `view` 函數從 Route 或 controller 返回。當然，正如有關 views 的文件中所描述的，可以使用 `view` 函數的第二個參數將資料傳遞到 Blade 檢視：

```
Route::get('/', function () {
    return view('greeting', ['name' => 'Finn']);
});
```

12.1.1 用 Livewire 為 Blade 賦能

想讓你的 Blade 範本更上一層樓，輕鬆建構動態介面嗎？看看 Laravel Livewire。Livewire 允許你編寫 Blade 元件，這些元件具有動態功能，通常只能通過 React 或 Vue 等前端框架來實現，這提供了一個很好的方法來建構現代，沒有複雜前端對應，基於客戶端渲染，無須很多的建構步驟的 JavaScript 框架。

12.2 顯示資料

你可以把變數置於花括號中以在檢視中顯示資料。例如，給定下方的路由：

```
Route::get('/', function () {
    return view('welcome', ['name' => 'Samantha']);
});
```

你可以像如下這樣顯示 `name` 變數的內容：

```
Hello, {{ $name }}.
```

> **技巧**：Blade 的 `{{ }}` 語句將被 PHP 的 `htmlspecialchars` 函數自動轉義以防範 XSS 攻擊。

你不僅限於顯示傳遞給檢視的變數的內容。你也可以回顯任何 PHP 函數的結果。實際上，你可以將所需的任何 PHP 程式碼放入 Blade echo 語句中：

```
The current UNIX timestamp is {{ time() }}.
```

12.2.1 HTML 實體編碼

默認情況下，Blade（和 Laravel `e` 助手）將對 HTML 實體進行雙重編碼。如果你想停用雙重編碼，請從 `AppServiceProvider` 的 `boot` 方法呼叫 `Blade::withoutDoubleEncoding` 方法：

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
```

```
{
    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        Blade::withoutDoubleEncoding();
    }
}
```

12.2.1.1 展示非轉義資料

默認情況下， Blade `{{ }}` 語句將被 PHP 的 `htmlspecialchars` 函數自動轉義以防範 XSS 攻擊。如果不想你的資料被轉義，那麼你可使用如下的語法：

```
Hello, {!! $name !!}.
```

> **注意：**在應用中顯示使用者提供的資料時請格外小心，請儘可能的使用轉義和雙引號語法來防範 XSS 攻擊。

12.2.2 Blade & JavaScript 框架

由於許多 JavaScript 框架也使用「花括號」來標識將顯示在瀏覽器中的表示式，因此，你可以使用 @ 符號來表示 Blade 渲染引擎應當保持不變。例如：

```
<h1>Laravel</h1>

Hello, @{{ name }}.
```

在這個例子中， @ 符號將被 Blade 移除；當然，Blade 將不會修改 `{{ name }}` 表示式，取而代之的是 JavaScript 範本來對其進行渲染。

@ 符號也用於轉義 Blade 指令：

```
{{-- Blade template --}}
@@if()

<!-- HTML output -->
@if()
```

12.2.2.1 渲染 JSON

有時，你可能會將陣列傳遞給檢視，以將其呈現為 JSON，以便初始化 JavaScript 變數。 例如：

```
<script>
    var app = <?php echo json_encode($array); ?>;
</script>
```

或者，你可以使用 `Illuminate\Support\Js::from` 方法指令，而不是手動呼叫 `json_encode`。 `from` 方法接受與 PHP 的 `json_encode` 函數相同的參數；但是，它將確保正確轉義生成的 JSON 以包含在 HTML 引號中。 `from` 方法將返回一個字串 `JSON.parse` JavaScript 語句，它將給定對象或陣列轉換為有效的 JavaScript 對象：

```
<script>
    var app = {{ Illuminate\Support\Js::from($array) }};
</script>
```

Laravel 框架的最新版本包括一個 `Js` 門面，它提供了在 Blade 範本中方便地訪問此功能：

```
<script>
    var app = {{ Js::from($array) }};
</script>
```

注意：你應該只使用 `Js::from` 渲染已經存在的變數為 JSON。 Blade 範本基於正規表示式，如果嘗試將複雜表示式傳遞給 `Js::from` 可能會導致無法預測的錯誤。

12.2.2.2 `@verbatim` 指令

如果你在範本中顯示很大一部分 JavaScript 變數，你可以將 HTML 嵌入到 `@verbatim` 指令中，這樣，你就不需要在每一個 Blade 回顯語句前新增 @ 符號：

```
@verbatim
     <div class="container">
          Hello, {{ name }}.
     </div>
@endverbatim
```

12.3 Blade 指令

除了範本繼承和顯示資料以外， Blade 還為常見的 PHP 控制結構提供了便捷的快捷方式，例如條件語句和循環。這些快捷方式為 PHP 控制結構提供了一個非常清晰、簡潔的書寫方式，同時，還與 PHP 中的控制結構保持了相似的語法特性。

12.3.1 If 語句

你可以使用 `@if` ， `@elseif` ， `@else` 和 `@endif` 指令構造 `if` 語句。這些指令功能與它們所對應的 PHP 語句完全一致：

```
@if (count($records) === 1)
     有一條記錄
@elseif (count($records) > 1)
     有多條記錄
@else
     沒有記錄
@endif
```

為了方便， Blade 還提供了一個 `@unless` 指令：

```
@unless (Auth::check())
     你還沒有登錄
@endunless
```

譯註：相當於 `@if (! Auth::check()) @endif`

除了上面所說條件指令外， `@isset` 和 `@empty` 指令亦可作為它們所對應的 PHP 函數的快捷方式：

```
@isset($records)
     // $records 已經被定義且不為 null ……
@endisset

@empty($records)
     // $records 為「空」……
@endempty
```

12.3.1.1 授權指令

`@auth` 和 `@guest` 指令可用於快速判斷當前使用者是否已經獲得 授權 或是遊客：

```
@auth
     // 使用者已經通過認證……
@endauth

@guest
```

```
    // 使用者沒有通過認證……
@endguest
```

如有需要，你亦可在使用 @auth 和 @guest 指令時指定 認證守衛：

```
@auth('admin')
    // 使用者已經通過認證...
@endauth

@guest('admin')
    // 使用者沒有通過認證...
@endguest
```

12.3.1.2 環境指令

你可以使用 @production 指令來判斷應用是否處於生產環境：

```
@production
    // 生產環境特定內容...
@endproduction
```

或者，你可以使用 @env 指令來判斷應用是否運行於指定的環境：

```
@env('staging')
    //  應用運行於「staging」環境...
@endenv

@env(['staging', 'production'])
    // 應用運行於 「staging」或 [生產] 環境...
@endenv
```

12.3.1.3 區塊指令

你可以使用 @hasSection 指令來判斷區塊是否有內容：

```
@hasSection('navigation')
    <div class="pull-right">
        @yield('navigation')
    </div>

    <div class="clearfix"></div>
@endif
```

你可以使用 sectionMissing 指令來判斷區塊是否沒有內容：

```
@sectionMissing('navigation')
    <div class="pull-right">
        @include('default-navigation')
    </div>
@endif
```

12.3.2 Switch 語句

你可使用 @switch ， @case ， @break ， @default 和 @endswitch 語句來構造 Switch 語句：

```
@switch($i)
    @case(1)
        First case...
        @break

    @case(2)
        Second case...
        @break

    @default
        Default case...
```

```
@endswitch
```

12.3.3 循環

除了條件語句， Blade 還提供了與 PHP 循環結構功能相同的指令。同樣，這些語句的功能和它們所對應的 PHP 語法一致：

```
@for ($i = 0; $i < 10; $i++)
    The current value is {{ $i }}
@endfor

@foreach ($users as $user)
    <p>This is user {{ $user->id }}</p>
@endforeach

@forelse ($users as $user)
    <li>{{ $user->name }}</li>
@empty
    <p>No users</p>
@endforelse

@while (true)
    <p>I'm looping forever.</p>
@endwhile
```

> **技巧：**在遍歷 `foreach` 循環時，你可以使用 循環變數 去獲取有關循環的有價值的資訊，例如，你處於循環的第一個迭代亦或是處於最後一個迭代。

使用循環時，還可以使用 `@continue` 和 `@break` 循環或跳過當前迭代：

```
@foreach ($users as $user)
    @if ($user->type == 1)
        @continue
    @endif

    <li>{{ $user->name }}</li>

    @if ($user->number == 5)
        @break
    @endif
@endforeach
```

你還可以在指令聲明中包含繼續或中斷條件：

```
@foreach ($users as $user)
    @continue($user->type == 1)

    <li>{{ $user->name }}</li>

    @break($user->number == 5)
@endforeach
```

12.3.4 Loop 變數

在遍歷 `foreach` 循環時，循環內部可以使用 `$loop` 變數。該變數提供了訪問一些諸如當前的循環索引和此次迭代是首次或是末次這樣的資訊的方式：

```
@foreach ($users as $user)
    @if ($loop->first)
        This is the first iteration.
    @endif

    @if ($loop->last)
        This is the last iteration.
```

```
    @endif

    <p>This is user {{ $user->id }}</p>
@endforeach
```

如果你處於巢狀循環中，你可以使用循環的 `$loop` 變數的 `parent` 屬性訪問父級循環：

```
@foreach ($users as $user)
    @foreach ($user->posts as $post)
        @if ($loop->parent->first)
            This is the first iteration of the parent loop.
        @endif
    @endforeach
@endforeach
```

該 `$loop` 變數還包含各種各樣有用的屬性：

屬性	描述
`$loop->index`	當前迭代的索引（從 0 開始）。
`$loop->iteration`	當前循環的迭代次數（從 1 開始）。
`$loop->remaining`	循環剩餘的迭代次數。
`$loop->count`	被迭代的陣列的元素個數。
`$loop->first`	當前迭代是否是循環的首次迭代。
`$loop->last`	當前迭代是否是循環的末次迭代。
`$loop->even`	當前循環的迭代次數是否是偶數。
`$loop->odd`	當前循環的迭代次數是否是奇數。
`$loop->depth`	當前循環的巢狀深度。
`$loop->parent`	巢狀循環中的父級循環。

12.3.5 有條件地編譯 class 樣式

該 `@class` 指令有條件地編譯 CSS class 樣式。該指令接收一個陣列，其中陣列的鍵包含你希望新增的一個或多個樣式的類名，而值是一個布林值表示式。如果陣列元素有一個數值的鍵，它將始終包含在呈現的 class 列表中：

```
@php
    $isActive = false;
    $hasError = true;
@endphp

<span @class([
    'p-4',
    'font-bold' => $isActive,
    'text-gray-500' => ! $isActive,
    'bg-red' => $hasError,
])></span>

<span class="p-4 text-gray-500 bg-red"></span>
```

同樣，`@style` 指令可用於有條件地將內聯 CSS 樣式新增到一個 HTML 元素中。

```
@php
    $isActive = true;
@endphp

<span @style([
    'background-color: red',
    'font-weight: bold' => $isActive,
])></span>

<span style="background-color: red; font-weight: bold;"></span>
```

12.3.6 附加屬性

為方便起見，你可以使用該 `@checked` 指令輕鬆判斷給定的 HTML 複選框輸入是否被「選中（checked）」。如果提供的條件判斷為 `true` ，則此指令將回顯 `checked`：

```
<input type="checkbox"
         name="active"
         value="active"
         @checked(old('active', $user->active)) />
```

同樣，該 `@selected` 指令可用於判斷給定的選項是否被「選中（selected）」：

```
<select name="version">
    @foreach ($product->versions as $version)
         <option value="{{ $version }}" @selected(old('version') == $version)>
             {{ $version }}
         </option>
    @endforeach
</select>
```

此外，該 `@disabled` 指令可用於判斷給定元素是否為「停用（disabled）」:

```
<button type="submit" @disabled($errors->isNotEmpty())>Submit</button>
```

此外，`@readonly` 指令可以用來指示某個元素是否應該是「唯讀 （readonly）」的。

```
<input type="email"
         name="email"
         value="email@laravel.com"
         @readonly($user->isNotAdmin()) />
```

此外，`@required` 指令可以用來指示一個給定的元素是否應該是「必需的（required）」。

```
<input type="text"
         name="title"
         value="title"
         @required($user->isAdmin()) />
```

12.3.7 包含子檢視

> **技巧：**雖然你可以自由使用該 `@include` 指令，但是 Blade 元件 提供了類似的功能，並提供了優於該 `@include` 指令的功能，如資料和屬性繫結。

Blade 的 `@include` 指令允許你從一個檢視中包含另外一個 Blade 檢視。父檢視中的所有變數在子檢視中都可以使用：

```
<div>
    @include('shared.errors')

    <form>
         <!-- Form Contents -->
    </form>
</div>
```

儘管子檢視可以繼承父檢視中所有可以使用的資料，但是你也可以傳遞一個額外的陣列，這個陣列在子檢視中也可以使用:

```
@include('view.name', ['status' => 'complete'])
```

如果你想要使用 `@include` 包含一個不存在的檢視，Laravel 將會拋出一個錯誤。如果你想要包含一個可能存在也可能不存在的檢視，那麼你應該使用 `@includeIf` 指令:

```
@includeIf('view.name', ['status' => 'complete'])
```

如果想要使用 `@include` 包含一個給定值為 `true` 或 `false` 的布林值表示式的檢視，那麼你可以使用

@includeWhen 和 @includeUnless 指令:

```
@includeWhen($boolean, 'view.name', ['status' => 'complete'])

@includeUnless($boolean, 'view.name', ['status' => 'complete'])
```

如果想要包含一個檢視陣列中第一個存在的檢視，你可以使用 includeFirst 指令:

```
@includeFirst(['custom.admin', 'admin'], ['status' => 'complete'])
```

> **注意：**在檢視中，你應該避免使用 __DIR__ 和 __FILE__ 這些常數，因為他們將引用已快取的和已編譯的檢視。

12.3.7.1 為集合渲染檢視

你可以使用 Blade 的 @each 指令將循環合併在一行內：

```
@each('view.name', $jobs, 'job')
```

該 @each 指令的第一個參數是陣列或集合中的元素的要渲染的檢視片段。第二個參數是你想要迭代的陣列或集合，當第三個參數是一個表示當前迭代的檢視的變數名。因此，如果你遍歷一個名為 jobs 的陣列，通常會在檢視片段中使用 job 變數來訪問每一個 job （jobs 陣列的元素）。在你的檢視片段中，可以使用 key 變數來訪問當前迭代的鍵。

你亦可傳遞第四個參數給 @each 指令。當給定的陣列為空時，將會渲染該參數所對應的檢視。

```
@each('view.name', $jobs, 'job', 'view.empty')
```

> **注意：**通過 @each 指令渲染的檢視不會繼承父檢視的變數。如果子檢視需要使用這些變數，你可以使用 @foreach 和 @include 來代替它。

12.3.8 @once 指令

該 @once 指令允許你定義範本的一部分內容，這部分內容在每一個渲染週期中只會被計算一次。該指令在使用 堆疊 推送一段特定的 JavaScript 程式碼到頁面的頭部環境下是很有用的。例如，如果你想要在循環中渲染一個特定的 元件 ，你可能希望僅在元件渲染的首次推送 JavaScript 程式碼到頭部：

```
@once
    @push('scripts')
        <script>
            // 你自訂的 JavaScript 程式碼...
        </script>
    @endpush
@endonce
```

由於該 @once 指令經常與 @push 或 @prepend 指令一起使用，為了使用方便，我們提供了 @pushOnce 和 @prependOnce 指令：

```
@pushOnce('scripts')
    <script>
        // 你自訂的 JavaScript 程式碼...
    </script>
@endPushOnce
```

12.3.9 原始 PHP 語法

在許多情況下，嵌入 PHP 程式碼到你的檢視中是很有用的。你可以在範本中使用 Blade 的 @php 指令執行原生的 PHP 程式碼塊：

```
@php
    $counter = 1;
```

```
@endphp
```

如果只需要寫一條 PHP 語句，可以在 @php 指令中包含該語句。

```
@php($counter = 1)
```

12.3.10 註釋

Blade 也允許你在檢視中定義註釋。但是，和 HTML 註釋不同， Blade 註釋不會被包含在應用返回的 HTML 中：

```
{{-- 這個註釋將不會出現在渲染的 HTML 中。 --}}
```

12.4 元件

元件和插槽的作用與區塊和佈局的作用一致；不過，有些人可能覺著元件和插槽更易於理解。有兩種書寫元件的方法：基於類的元件和匿名元件。

你可以使用 make:component Artisan 命令來建立一個基於類的元件。我們將會建立一個簡單的 Alert 元件用於說明如何使用元件。該 make:component 命令將會把元件置於 App\View\Components 目錄中：

```
php artisan make:component Alert
```

該 make:component 命令將會為元件建立一個檢視範本。建立的檢視被置於 resources/views/components 目錄中。在為自己的應用程式編寫元件時，會在 app/View/Components 目錄和 resources/views/components 目錄中自動發現元件，因此通常不需要進一步的元件註冊。

你還可以在子目錄中建立元件：

```
php artisan make:component Forms/Input
```

上面的命令將在目錄中建立一個 Input 元件， App\View\Components\Forms 檢視將放置在 resources/views/components/forms 目錄中。

如果你想建立一個匿名元件（一個只有 Blade 範本並且沒有類的元件），你可以在呼叫命令 make:component 使用該 --view 標誌：

```
php artisan make:component forms.input --view
```

上面的命令將在 resources/views/components/forms/input.blade.php 建立一個 Blade 檔案，該檔案中可以通過 <x-forms.input /> 作為元件呈現。

12.4.1.1 手動註冊包元件

當為你自己的應用編寫元件的時候，Laravel 將會自動發現位於 app/View/Components 目錄和 resources/views/components 目錄中的元件。

當然，如果你使用 Blade 元件編譯一個包，你可能需要手動註冊元件類及其 HTML 標籤別名。你應該在包的服務提供者的 boot 方法中註冊你的元件：

```
use Illuminate\Support\Facades\Blade;

/**
 * 註冊你的包的服務
 */
public function boot(): void
{
    Blade::component('package-alert', Alert::class);
}
```

當元件註冊完成後，便可使用標籤別名來對其進行渲染。

```
<x-package-alert/>
```

或者，你可以使用該 `componentNamespace` 方法按照約定自動載入元件類。例如，一個 `Nightshade` 包可能有 `Calendar` 和 `ColorPicker` 元件駐留在 `Package\Views\Components` 命名空間中：

```
use Illuminate\Support\Facades\Blade;

/**
 * 註冊你的包的服務
 */
public function boot(): void
{
    Blade::componentNamespace('Nightshade\\Views\\Components', 'nightshade');
}
```

這將允許他們的供應商命名空間使用包元件，使用以下 `package-name::` 語法：

```
<x-nightshade::calendar />
<x-nightshade::color-picker />
```

Blade 將自動檢測連結到該元件的類，通過對元件名稱進行帕斯卡大小寫。使用「點」表示法也支援子目錄。

12.4.2 顯示元件

要顯示一個元件，你可以在 Blade 範本中使用 Blade 元件標籤。 Blade 元件以 `x-` 字串開始，其後緊接元件類 kebab case 形式的名稱（即單詞與單詞之間使用短橫線 `-` 進行連接）：

```
<x-alert/>

<x-user-profile/>
```

如果元件位於 `App\View\Components` 目錄的子目錄中，你可以使用 `.` 字元來指定目錄層級。例如，假設我們有一個元件位於 `App\View\Components\Inputs\Button.php`，那麼我們可以像這樣渲染它：

```
<x-inputs.button/>
```

如果你想有條件地渲染你的元件，你可以在你的元件類上定義一個 `shouldRender` 方法。如果 `shouldRender` 方法返回 `false`，該元件將不會被渲染。

```
use Illuminate\Support\Str;

/**
 * 該元件是否應該被渲染
 */
public function shouldRender(): bool
{
    return Str::length($this->message) > 0;
}
```

12.4.3 傳遞資料到元件中

你可以使用 HTML 屬性傳遞資料到 Blade 元件中。普通的值可以通過簡單的 HTML 屬性來傳遞給元件。PHP 表示式和變數應該通過以 `:` 字元作為前綴的變數來進行傳遞：

```
<x-alert type="error" :message="$message"/>
```

你應該在類的構造器中定義元件的必要資料。在元件的檢視中，元件的所有 public 類型的屬性都是可用的。不必通過元件類的 `render` 方法傳遞：

```
<?php

namespace App\View\Components;

use Illuminate\View\Component;
use Illuminate\View\View;
```

```
class Alert extends Component
{
    /**
     * 建立元件實例。
     */
    public function __construct(
        public string $type,
        public string $message,
    ) {}

    /**
     * 獲取代表該元件的檢視/內容
     */
    public function render(): View
    {
        return view('components.alert');
    }
}
```

渲染元件時，你可以回顯變數名來顯示元件的 public 變數的內容：

```
<div class="alert alert-{{ $type }}">
    {{ $message }}
</div>
```

12.4.3.1 命名方式（Casing）

元件的構造器的參數應該使用 駝峰式 類型，在 HTML 屬性中引用參數名時應該使用 短橫線隔開式 kebab-case ：單詞與單詞之間使用短橫線 - 進行連接） 。例如，給定如下的元件構造器：

```
/**
 * 建立一個元件實例
 */
public function __construct(
    public string $alertType,
) {}
```

$alertType 參數可以像這樣使用：

```
<x-alert alert-type="danger" />
```

12.4.3.2 短屬性語法/省略屬性語法

當向元件傳遞屬性時，你也可以使用「短屬性語法/省略屬性語法」（省略屬性書寫）。這通常很方便，因為屬性名稱經常與它們對應的變數名稱相匹配。

```
{{-- 短屬性語法/省略屬性語法... --}}
<x-profile :$userId :$name />

{{-- 等價於... --}}
<x-profile :user-id="$userId" :name="$name" />
```

12.4.3.3 轉義屬性渲染

因為一些 JavaScript 框架，例如 Alpine.js 還可以使用冒號前綴屬性，你可以使用雙冒號 (::) 前綴通知 Blade 屬性不是 PHP 表示式。例如，給定以下元件：

```
<x-button ::class="{ danger: isDeleting }">
    Submit
</x-button>
```

Blade 將渲染出以下 HTML 內容：

```
<button :class="{ danger: isDeleting }">
    Submit
```

```
</button>
```

12.4.3.4 元件方法

除了元件範本可用的公共變數外，還可以呼叫元件上的任何公共方法。例如，假設一個元件有一個 `isSelected` 方法：

```
/**
 * 確定給定選項是否為當前選定的選項。
 */
public function isSelected(string $option): bool
{
    return $option === $this->selected;
}
```

你可以通過呼叫與方法名稱匹配的變數，從元件範本執行此方法：

```
<option {{ $isSelected($value) ? 'selected' : '' }} value="{{ $value }}">
    {{ $label }}
</option>
```

12.4.3.5 訪問元件類中的屬性和插槽

Blade 元件還允許你訪問類的 render 方法中的元件名稱、屬性和插槽。但是，為了訪問這些資料，應該從元件的 `render` 方法返回閉包。閉包將接收一個 `$data` 陣列作為它的唯一參數。此陣列將包含幾個元素，這些元素提供有關元件的資訊：

```
use Closure;

/**
 * 獲取表示元件的檢視 / 內容
 */
public function render(): Closure
{
    return function (array $data) {
        // $data['componentName'];
        // $data['attributes'];
        // $data['slot'];

        return '<div>Components content</div>';
    };
}
```

`componentName` 等於 `x-` 前綴後面的 HTML 標記中使用的名稱。所以 `<x-alert />` 的 `componentName` 將是 `alert` 。 `attributes` 元素將包含 HTML 標記上的所有屬性。 `slot` 元素是一個 `Illuminate\Support\HtmlString` 實例，包含元件的插槽內容。

閉包應該返回一個字串。如果返回的字串與現有檢視相對應，則將呈現該檢視；否則，返回的字串將作為內聯 Blade 檢視進行計算。

12.4.3.6 附加依賴項

如果你的元件需要引入來自 Laravel 的 服務容器的依賴項，你可以在元件的任何資料屬性之前列出這些依賴項，這些依賴項將由容器自動注入：

```
use App\Services\AlertCreator;

/**
 * 建立元件實例
 */
public function __construct(
    public AlertCreator $creator,
    public string $type,
```

```
    public string $message,
) {}
```

12.4.3.7 隱藏屬性/方法

如果要防止某些公共方法或屬性作為變數公開給元件範本，可以將它們新增到元件的 $except 陣列屬性中：

```
<?php

namespace App\View\Components;

use Illuminate\View\Component;

class Alert extends Component
{
    /**
     * 不應向元件範本公開的屬性/方法。
     *
     * @var array
     */
    protected $except = ['type'];

    /**
     * Create the component instance.
     */
    public function __construct(
        public string $type,
    ) {}
}
```

12.4.4 元件屬性

我們已經研究了如何將資料屬性傳遞給元件；但是，有時你可能需要指定額外的 HTML 屬性，例如 class，這些屬性不是元件運行所需的資料的一部分。通常，你希望將這些附加屬性向下傳遞到元件範本的根元素。例如，假設我們要呈現一個 alert 元件，如下所示：

```
<x-alert type="error" :message="$message" class="mt-4"/>
```

所有不屬於元件的構造器的屬性都將被自動新增到元件的「屬性包」中。該屬性包將通過 $attributes 變數自動傳遞給元件。你可以通過回顯這個變數來渲染所有的屬性：

```
<div {{ $attributes }}>
    <!-- 元件內容 -->
</div>
```

> **注意：**此時不支援在元件中使用諸如 @env 這樣的指令。例如，<x-alert :live="@env('production')"/> 不會被編譯。

12.4.4.1 默認 / 合併屬性

某些時候，你可能需要指定屬性的預設值，或將其他值合併到元件的某些屬性中。為此，你可以使用屬性包的 merge 方法。 此方法對於定義一組應始終應用於元件的默認 CSS 類特別有用：

```
<div {{ $attributes->merge(['class' => 'alert alert-'.$type]) }}>
    {{ $message }}
</div>
```

假設我們如下方所示使用該元件：

```
<x-alert type="error" :message="$message" class="mb-4"/>
```

最終呈現的元件 HTML 將如下所示：

```
<div class="alert alert-error mb-4">
```

```
    <!-- Contents of the $message variable -->
</div>
```

12.4.4.2 有條件地合併類

有時你可能希望在給定條件為 `true` 時合併類。 你可以通過該 `class` 方法完成此操作，該方法接受一個類陣列，其中陣列鍵包含你希望新增的一個或多個類，而值是一個布林值表示式。如果陣列元素有一個數字鍵，它將始終包含在呈現的類列表中：

```
<div {{ $attributes->class(['p-4', 'bg-red' => $hasError]) }}>
    {{ $message }}
</div>
```

如果需要將其他屬性合併到元件中，可以將 `merge` 方法連結到 `class` 方法中：

```
<button {{ $attributes->class(['p-4'])->merge(['type' => 'button']) }}>
    {{ $slot }}
</button>
```

> **技巧：**如果你需要有條件地編譯不應接收合併屬性的其他 HTML 元素上的類，你可以使用 @class 指令。

12.4.4.3 非 class 屬性的合併

當合並非 `class` 屬性的屬性時，提供給 `merge` 方法的值將被視為該屬性的「default」值。但是，與 `class` 屬性不同，這些屬性不會與注入的屬性值合併。相反，它們將被覆蓋。例如， `button` 元件的實現可能如下所示：

```
<button {{ $attributes->merge(['type' => 'button']) }}>
    {{ $slot }}
</button>
```

若要使用自訂 `type` 呈現按鈕元件，可以在使用該元件時指定它。如果未指定 `type`，則將使用 `button` 作為 type 值：

```
<x-button type="submit">
    Submit
</x-button>
```

本例中 `button` 元件渲染的 HTML 為：

```
<button type="submit">
    Submit
</button>
```

如果希望 `class` 以外的屬性將其預設值和注入值連接在一起，可以使用 `prepends` 方法。在本例中， `data-controller` 屬性始終以 `profile-controller` 開頭，並且任何其他注入 `data-controller` 的值都將放在該預設值之後：

```
<div {{ $attributes->merge(['data-controller' => $attributes->prepends('profile-
controller')]) }}>
    {{ $slot }}
</div>
```

12.4.4.4 保留屬性 / 過濾屬性

可以使用 `filter` 方法篩選屬性。如果希望在屬性包中保留屬性，此方法接受應返回 `true` 的閉包：

```
{{ $attributes->filter(fn (string $value, string $key) => $key == 'foo') }}
```

為了方便起見，你可以使用 `whereStartsWith` 方法檢索其鍵以給定字串開頭的所有屬性：

```
{{ $attributes->whereStartsWith('wire:model') }}
```

相反，該 `whereDoesntStartWith` 方法可用於排除鍵以給定字串開頭的所有屬性：

```
{{ $attributes->whereDoesntStartWith('wire:model') }}
```

使用 `first` 方法，可以呈現給定屬性包中的第一個屬性：

```
{{ $attributes->whereStartsWith('wire:model')->first() }}
```

如果要檢查元件上是否存在屬性，可以使用 `has` 方法。此方法接受屬性名稱作為其唯一參數，並返回一個布林值，指示該屬性是否存在：

```
@if ($attributes->has('class'))
    <div>Class attribute is present</div>
@endif
```

你可以使用 `get` 方法檢索特定屬性的值：

```
{{ $attributes->get('class') }}
```

12.4.5 保留關鍵字

默認情況下，為了渲染元件，會保留一些關鍵字供 Blade 內部使用。以下關鍵字不能定義為元件中的公共屬性或方法名稱：

- `data`
- `render`
- `resolveView`
- `shouldRender`
- `view`
- `withAttributes`
- `withName`

12.4.6 插槽

你通常需要通過「插槽」將其他內容傳遞給元件。通過回顯 `$slot` 變數來呈現元件插槽。為了探索這個概念，我們假設 `alert` 元件具有以下內容：

```
<!-- /resources/views/components/alert.blade.php -->

<div class="alert alert-danger">
    {{ $slot }}
</div>
```

我們可以通過向元件中注入內容將內容傳遞到 `slot` ：

```
<x-alert>
    <strong>Whoops!</strong> Something went wrong!
</x-alert>
```

有時候一個元件可能需要在它內部的不同位置放置多個不同的插槽。我們來修改一下 alert 元件，使其允許注入「title」：

```
<!-- /resources/views/components/alert.blade.php -->

<span class="alert-title">{{ $title }}</span>

<div class="alert alert-danger">
    {{ $slot }}
</div>
```

你可以使用 `x-slot` 標籤來定義命名插槽的內容。任何沒有在 `x-slot` 標籤中的內容都將傳遞給 `$slot` 變數中的元件：

```
<x-alert>
    <x-slot:title>
        Server Error
    </x-slot>
```

```
    <strong>Whoops!</strong> Something went wrong!
</x-alert>
```

12.4.6.1 範疇插槽

如果你使用諸如 Vue 這樣的 JavaScript 框架，那麼你應該很熟悉「範疇插槽」，它允許你從插槽中的元件訪問資料或者方法。 Laravel 中也有類似的用法，只需在你的元件中定義 public 方法或屬性，並且使用 `$component` 變數來訪問插槽中的元件。在此示例中，我們將假設元件在其元件類上定義了 `x-alert` 一個公共方法：`formatAlert`

```
<x-alert>
    <x-slot:title>
        {{ $component->formatAlert('Server Error') }}
    </x-slot>

    <strong>Whoops!</strong> Something went wrong!
</x-alert>
```

12.4.6.2 插槽屬性

像 Blade 元件一樣，你可以為插槽分配額外的 屬性 ，例如 CSS 類名：

```
<x-card class="shadow-sm">
    <x-slot:heading class="font-bold">
        Heading
    </x-slot>

    Content

    <x-slot:footer class="text-sm">
        Footer
    </x-slot>
</x-card>
```

要與插槽屬性互動，你可以訪問 `attributes` 插槽變數的屬性。有關如何與屬性互動的更多資訊，請參閱有關 元件屬性 的文件：

```
@props([
    'heading',
    'footer',
])

<div {{ $attributes->class(['border']) }}>
    <h1 {{ $heading->attributes->class(['text-lg']) }}>
        {{ $heading }}
    </h1>

    {{ $slot }}

    <footer {{ $footer->attributes->class(['text-gray-700']) }}>
        {{ $footer }}
    </footer>
</div>
```

12.4.7 內聯元件檢視

對於小型元件而言，管理元件類和元件檢視範本可能會很麻煩。因此，你可以從 `render` 方法中返回元件的內容：

```
/**
 * 獲取元件的檢視 / 內容。
 */
public function render(): string
```

```
{
    return <<<'blade'
        <div class="alert alert-danger">
            {{ $slot }}
        </div>
    blade;
}
```

12.4.7.1 生成內聯檢視元件

要建立一個渲染內聯檢視的元件，你可以在運行 `make:component` 命令時使用 `inline` ：

```
php artisan make:component Alert --inline
```

12.4.8 動態元件

有時你可能需要渲染一個元件，但直到執行階段才知道應該渲染哪個元件。在這種情況下, 你可以使用 Laravel 內建的 `dynamic-component` 元件, 根據執行階段的值或變數來渲染元件:

```
<x-dynamic-component :component="$componentName" class="mt-4" />
```

12.4.9 手動註冊元件

> **注意：**以下關於手動註冊元件的文件主要適用於那些正在編寫包含檢視元件的 Laravel 包的使用者。如果你不是在寫包，這一部分的元件文件可能與你無關。

當為自己的應用程式編寫元件時，元件會在 `app/View/Components` 目錄和 `resources/views/components` 目錄下被自動發現。

但是，如果你正在建立一個利用 Blade 元件的包，或者將元件放在非傳統的目錄中，你將需要手動註冊你的元件類和它的 HTML 標籤別名，以便 Laravel 知道在哪裡可以找到這個元件。你通常應該在你的包的服務提供者的 `boot` 方法中註冊你的元件：

```
use Illuminate\Support\Facades\Blade;
use VendorPackage\View\Components\AlertComponent;

/**
 * 註冊你的包的服務。
 */
public function boot(): void
{
    Blade::component('package-alert', AlertComponent::class);
}
```

一旦你的元件被註冊，它就可以使用它的標籤別名進行渲染。

```
<x-package-alert/>
```

12.4.9.1 自動載入包元件

另外，你可以使用 `componentNamespace` 方法來自動載入元件類。例如，一個 `Nightshade` 包可能有 `Calendar` 和 `ColorPicker` 元件，它們位於 `PackageViews\Components` 命名空間中。

```
use Illuminate\Support\Facades\Blade;

/**
 * 註冊你的包的服務。
 */
public function boot(): void
{
    Blade::componentNamespace('Nightshade\\Views\\Components', 'nightshade');
```

```
}
```

這將允許使用 package-name::語法的供應商名稱空間來使用包的元件。

```
<x-nightshade::calendar />
<x-nightshade::color-picker />
```

Blade 將通過元件名稱的駝峰式大小寫 (pascal-casing) 自動檢測與該元件連結的類。也支援使用 “點”符號的子目錄。

12.4.10 匿名元件

與行內元件相同，匿名元件提供了一個通過單個檔案管理元件的機制。然而，匿名元件使用的是一個沒有關聯類的單一檢視檔案。要定義一個匿名元件，你只需將 Blade 範本置於 resources/views/components 目錄下。例如，假設你在 resources/views/components/alert.blade.php 中定義了一個元件：

```
<x-alert/>
```

如果元件在 components 目錄的子目錄中，你可以使用 . 字元來指定其路徑。例如，假設元件被定義在 resources/views/components/inputs/button.blade.php 中，你可以像這樣渲染它：

```
<x-inputs.button/>
```

12.4.10.1 匿名索引元件

有時，當一個元件由許多 Blade 範本組成時，你可能希望將給定元件的範本分組到一個目錄中。例如，想像一個具有以下目錄結構的「可摺疊」元件：

```
/resources/views/components/accordion.blade.php
/resources/views/components/accordion/item.blade.php
```

此目錄結構允許你像這樣呈現元件及其項目：

```
<x-accordion>
    <x-accordion.item>
        ...
    </x-accordion.item>
</x-accordion>
```

然而，為了通過 x-accordion 渲染元件， 我們被迫將「索引」元件範本放置在 resources/views/components 目錄中，而不是與其他相關的範本巢狀在 accordion 目錄中。

幸運的是，Blade 允許你 index.blade.php 在元件的範本目錄中放置檔案。當 index.blade.php 元件存在範本時，它將被呈現為元件的「根」節點。因此，我們可以繼續使用上面示例中給出的相同 Blade 語法；但是，我們將像這樣調整目錄結構：

```
/resources/views/components/accordion/index.blade.php
/resources/views/components/accordion/item.blade.php
```

12.4.10.2 資料 / 屬性

由於匿名元件沒有任何關聯類，你可能想要區分哪些資料應該被作為變數傳遞給元件，而哪些屬性應該被存放於 屬性包中。

你可以在元件的 Blade 範本的頂層使用 @props 指令來指定哪些屬性應該作為資料變數。元件中的其他屬性都將通過屬性包的形式提供。如果你想要為某個資料變數指定一個預設值，你可以將屬性名作為陣列鍵，預設值作為陣列值來實現：

```
<!-- /resources/views/components/alert.blade.php -->

@props(['type' => 'info', 'message'])

<div {{ $attributes->merge(['class' => 'alert alert-'.$type]) }}>
```

```
    {{ $message }}
</div>
```

給定上面的元件定義，我們可以像這樣渲染元件：

```
<x-alert type="error" :message="$message" class="mb-4"/>
```

12.4.10.3 訪問父元件資料

有時你可能希望從子元件中的父元件訪問資料。在這些情況下，你可以使用該 `@aware` 指令。例如，假設我們正在建構一個由父 `<x-menu>` 和 子組成的複雜菜單元件 `<x-menu.item>`：

```
<x-menu color="purple">
    <x-menu.item>...</x-menu.item>
    <x-menu.item>...</x-menu.item>
</x-menu>
```

該 `<x-menu>` 元件可能具有如下實現：

```
<!-- /resources/views/components/menu/index.blade.php -->

@props(['color' => 'gray'])

<ul {{ $attributes->merge(['class' => 'bg-'.$color.'-200']) }}>
    {{ $slot }}
</ul>
```

因為 `color` 只被傳遞到父級 (`<x-menu>`)中，所以 `<x-menu.item>` 在內部是不可用的。但是，如果我們使用該 `@aware` 指令，我們也可以使其在內部可用 `<x-menu.item>` ：

```
<!-- /resources/views/components/menu/item.blade.php -->

@aware(['color' => 'gray'])

<li {{ $attributes->merge(['class' => 'text-'.$color.'-800']) }}>
    {{ $slot }}
</li>
```

> **注意：**該 `@aware` 指令無法訪問未通過 HTML 屬性顯式傳遞給父元件的父資料。`@aware` 指令 不能訪問未顯式傳遞給父元件的預設值 `@props` 。

12.4.11 匿名元件路徑

如前所述，匿名元件通常是通過在你的 `resources/views/components` 目錄下放置一個 Blade 範本來定義的。然而，你可能偶爾想在 Laravel 註冊其他匿名元件的路徑，除了默認路徑。

`anonymousComponentPath` 方法接受匿名元件位置的「路徑」作為它的第一個參數，並接受一個可選的「命名空間」作為它的第二個參數，元件應該被放在這個命名空間下。通常，這個方法應該從你的應用程式的一個服務提供者 的 `boot` 方法中呼叫。

```
/**
 * 引導任何應用服務。
 */
public function boot(): void
{
    Blade::anonymousComponentPath(__DIR__.'/../components');
}
```

當元件路徑被註冊而沒有指定前綴時，就像上面的例子一樣，它們在你的 Blade 元件中可能也沒有相應的前綴。例如，如果一個 `panel.blade.php` 元件存在於上面註冊的路徑中，它可能會被呈現為這樣。

```
<x-panel />
```

前綴「命名空間」可以作為第二個參數提供給 `anonymousComponentPath` 方法。

```
Blade::anonymousComponentPath(__DIR__.'/../components', 'dashboard');
```

當提供一個前綴時，在該「命名空間」內的元件可以在渲染時將該元件的命名空間前綴到該元件的名稱。

```
<x-dashboard::panel />
```

12.5 建構佈局

12.5.1 使用元件佈局

大多數 web 應用程式在不同的頁面上有相同的總體佈局。如果我們必須在建立的每個檢視中重複整個佈局 HTML，那麼維護我們的應用程式將變得非常麻煩和困難。謝天謝地，將此佈局定義為單個 Blade 元件 並在整個應用程式中非常方便地使用它。

12.5.1.1 定義佈局元件

例如，假設我們正在建構一個「todo list」應用程式。我們可以定義如下所示的 `layout` 元件：

```
<!-- resources/views/components/layout.blade.php -->

<html>
    <head>
        <title>{{ $title ?? 'Todo Manager' }}</title>
    </head>
    <body>
        <h1>Todos</h1>
        <hr/>
        {{ $slot }}
    </body>
</html>
```

12.5.1.2 應用佈局元件

一旦定義了 `layout` 元件，我們就可以建立一個使用該元件的 Blade 檢視。在本例中，我們將定義一個顯示任務列表的簡單檢視：

```
<!-- resources/views/tasks.blade.php -->

<x-layout>
    @foreach ($tasks as $task)
        {{ $task }}
    @endforeach
</x-layout>
```

請記住，注入到元件中的內容將提供給 `layout` 元件中的默認 `$slot` 變數。正如你可能已經注意到的，如果提供了 `$title` 插槽，那麼我們的 `layout` 也會尊從該插槽；否則，將顯示默認的標題。我們可以使用元件文件中討論的標準槽語法從任務列表檢視中插入自訂標題。 我們可以使用元件文件中討論的標準插槽語法從任務列表檢視中注入自訂標題：

```
<!-- resources/views/tasks.blade.php -->

<x-layout>
    <x-slot:title>
        Custom Title
    </x-slot>

    @foreach ($tasks as $task)
        {{ $task }}
    @endforeach
</x-layout>
```

現在我們已經定義了佈局和任務列表檢視，我們只需要從路由中返回 task 檢視即可：

```
use App\Models\Task;

Route::get('/tasks', function () {
    return view('tasks', ['tasks' => Task::all()]);
});
```

12.5.2 使用範本繼承進行佈局

12.5.2.1 定義一個佈局

佈局也可以通過 「範本繼承」 建立。在引入 元件 之前，這是建構應用程式的主要方法。

讓我們看一個簡單的例子做開頭。首先，我們將檢查頁面佈局。由於大多數 web 應用程式在不同的頁面上保持相同的總體佈局，因此將此佈局定義為單一檢視非常方便：

```
<!-- resources/views/layouts/app.blade.php -->

<html>
    <head>
        <title>App Name - @yield('title')</title>
    </head>
    <body>
        @section('sidebar')
            這是一個主要的側邊欄
        @show

        <div class="container">
            @yield('content')
        </div>
    </body>
</html>
```

如你所見，此檔案包含經典的 HTML 標記。但是，請注意 @section 和 @yield 指令。顧名思義，@section 指令定義內容的一部分，而 @yield 指令用於顯示給定部分的內容。

現在我們已經為應用程式定義了一個佈局，讓我們定義一個繼承該佈局的子頁面。

12.5.2.2 繼承佈局

定義子檢視時，請使用 @extends Blade 指令指定子檢視應「繼承」的佈局。擴展 Blade 佈局的檢視可以使用 @section 指令將內容注入佈局的節點中。請記住，如上面的示例所示，這些部分的內容將使用 @yield 顯示在佈局中：

```
<!-- resources/views/child.blade.php -->

@extends('layouts.app')

@section('title', 'Page Title')

@section('sidebar')
    @parent

    <p>This is appended to the master sidebar.</p>
@endsection

@section('content')
    <p>This is my body content.</p>
@endsection
```

在本例中，sidebar 部分使用 @parent 指令將內容追加（而不是覆蓋）到局部的側欄位置。在呈現檢視時，

@parent 指令將被佈局的內容替換。

> **技巧：**與前面的示例相反，本 sidebar 節以 @endsection 結束，而不是以 @show 結束。@endsection 指令將只定義一個節，@show 將定義並 **立即 yield** 該節。

該 @yield 指令還接受預設值作為其第二個參數。如果要生成的節點未定義，則將呈現此內容：

```
@yield('content', 'Default content')
```

12.6 表單

12.6.1 CSRF 欄位

無論何時在應用程式中定義 HTML 表單，都應該在表單中包含一個隱藏的 CSRF 令牌欄位，以便 CSRF 保護中介軟體 可以驗證請求。你可以使用 @csrf Blade 指令生成令牌欄位：

```
<form method="POST" action="/profile">
    @csrf

    ...
</form>
```

12.6.2 Method 欄位

由於 HTML 表單不能發出 PUT、PATCH 或 DELETE 請求，因此需要新增一個隱藏的 _method 欄位來欺騙這些 HTTP 動詞。 @method Blade 指令可以為你建立此欄位：

```
<form action="/foo/bar" method="POST">
    @method('PUT')

    ...
</form>
```

12.6.3 表單校驗錯誤

該 @error 指令可用於快速檢查給定屬性是否存在 驗證錯誤消息 。在 @error 指令中，可以回顯 $message 變數以顯示錯誤消息：

```
<!-- /resources/views/post/create.blade.php -->

<label for="title">Post Title</label>

<input id="title"
    type="text"
    class="@error('title') is-invalid @enderror">

@error('title')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

由於該 @error 指令編譯為「if」語句，因此你可以在 @else 屬性沒有錯誤時使用該指令來呈現內容：

```
<!-- /resources/views/auth.blade.php -->

<label for="email">Email address</label>

<input id="email"
    type="email"
    class="@error('email') is-invalid @else is-valid @enderror">
```

你可以將 特定錯誤包的名稱 作為第二個參數傳遞給 `@error` 指令，以便在包含多個表單的頁面上檢索驗證錯誤消息：

```
<!-- /resources/views/auth.blade.php -->

<label for="email">Email address</label>

<input id="email"
    type="email"
    class="@error('email', 'login') is-invalid @enderror">

@error('email', 'login')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

12.7 堆疊

Blade 允許你推送到可以在其他檢視或佈局中的其他地方渲染的命名堆疊。這對於指定子檢視所需的任何 JavaScript 庫特別有用：

```
@push('scripts')
    <script src="/example.js"></script>
@endpush
```

如果你想在給定的布林值表示式評估為 `true` 時 `@push` 內容，你可以使用 `@pushIf` 指令。

```
@pushIf($shouldPush, 'scripts')
    <script src="/example.js"></script>
@endPushIf
```

你可以根據需要多次推入堆疊。要呈現完整的堆疊內容，請將堆疊的名稱傳遞給 `@stack` 指令：

```
<head>
    <!-- Head Contents -->

    @stack('scripts')
</head>
```

如果要將內容前置到堆疊的開頭，應使用 `@prepend` 指令：

```
@push('scripts')
    This will be second...
@endpush

// Later...

@prepend('scripts')
    This will be first...
@endprepend
```

12.8 服務注入

該 `@inject` 指令可用於從 Laravel 服務容器中檢索服務。傳遞給 `@inject` 的第一個參數是要將服務放入的變數的名稱，而第二個參數是要解析的服務的類或介面名稱：

```
@inject('metrics', 'App\Services\MetricsService')

<div>
    Monthly Revenue: {{ $metrics->monthlyRevenue() }}.
</div>
```

12.9 渲染內聯 Blade 範本

有時你可能需要將原始 Blade 範本字串轉換為有效的 HTML。你可以使用 `Blade` 門面提供的 `render` 方法來完成此操作。該 `render` 方法接受 Blade 範本字串和提供給範本的可選資料陣列：

```
use Illuminate\Support\Facades\Blade;

return Blade::render('Hello, {{ $name }}', ['name' => 'Julian Bashir']);
```

Laravel 通過將內聯 Blade 範本寫入 `storage/framework/views` 目錄來呈現它們。如果你希望 Laravel 在渲染 Blade 範本後刪除這些臨時檔案，你可以為 `deleteCachedView` 方法提供參數：

```
return Blade::render(
    'Hello, {{ $name }}',
    ['name' => 'Julian Bashir'],
    deleteCachedView: true
);
```

12.10 渲染 Blade 片段

當使用 Turbo 和 htmx 等前端框架時，你可能偶爾需要在你的 HTTP 響應中只返回 Blade 範本的一個部分。Blade「片段（fragment）」允許你這樣做。要開始，將你的 Blade 範本的一部分放在`@fragment` 和 `@endfragment` 指令中。

```
@fragment('user-list')
    <ul>
        @foreach ($users as $user)
            <li>{{ $user->name }}</li>
        @endforeach
    </ul>
@endfragment
```

然後，在渲染使用該範本的檢視時，你可以呼叫 `fragment` 方法來指定只有指定的片段應該被包含在傳出的 HTTP 響應中。

```
return view('dashboard', ['users' => $users])->fragment('user-list');
```

`fragmentIf` 方法允許你根據一個給定的條件有條件地返回一個檢視的片段。否則，整個檢視將被返回。

```
return view('dashboard', ['users' => $users])
    ->fragmentIf($request->hasHeader('HX-Request'), 'user-list');
```

`fragments` 和 `fragmentsIf` 方法允許你在響應中返回多個檢視片段。這些片段將被串聯起來。

```
view('dashboard', ['users' => $users])
    ->fragments(['user-list', 'comment-list']);

view('dashboard', ['users' => $users])
    ->fragmentsIf(
        $request->hasHeader('HX-Request'),
        ['user-list', 'comment-list']
    );
```

12.11 擴展 Blade

Blade 允許你使用 `directive` 方法定義自己的自訂指令。當 Blade 編譯器遇到自訂指令時，它將使用該指令包含的表示式呼叫提供的回呼。

下面的示例建立了一個 `@datetime($var)` 指令，該指令格式化給定的 `$var`，它應該是 `DateTime` 的一個實例：

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * 註冊應用的服務
     */
    public function register(): void
    {
        // ...
    }

    /**
     * Bootstrap any application services.
     */
    public function boot(): void
    {
        Blade::directive('datetime', function (string $expression) {
            return "<?php echo ($expression)->format('m/d/Y H:i'); ?>";
        });
    }
}
```

正如你所見，我們將 `format` 方法應用到傳遞給指令中的任何表示式上。因此，在本例中，此指令生成的最終 PHP 將是：

```
<?php echo ($var)->format('m/d/Y H:i'); ?>
```

> **注意：**更新 Blade 指令的邏輯後，需要刪除所有快取的 Blade 檢視。可以使用 `view:clear` Artisan 命令。

12.11.1 自訂回顯處理程序

如果你試圖使用 Blade 來「回顯」一個對象， 該對象的 `__toString` 方法將被呼叫。該`__toString` 方法是 PHP 內建的「魔術方法」之一。但是，有時你可能無法控制 `__toString` 給定類的方法，例如當你與之互動的類屬於第三方庫時。

在這些情況下，Blade 允許您為該特定類型的對象註冊自訂回顯處理程序。為此，您應該呼叫 Blade 的 `stringable` 方法。該 `stringable` 方法接受一個閉包。這個閉包類型應該提示它負責呈現的對象的類型。通常，應該在應用程式的 `AppServiceProvider` 類的 `boot` 方法中呼叫該 `stringable` 方法：

```
use Illuminate\Support\Facades\Blade;
use Money\Money;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Blade::stringable(function (Money $money) {
        return $money->formatTo('en_GB');
    });
}
```

定義自訂回顯處理程序後，您可以簡單地回顯 Blade 範本中的對象：

```
Cost: {{ $money }}
```

12.11.2 自訂 if 聲明

在定義簡單的自訂條件語句時，編寫自訂指令通常比較複雜。因此，Blade 提供了一個 Blade::if 方法，允許你使用閉包快速定義自訂條件指令。例如，讓我們定義一個自訂條件來檢查為應用程式組態的默認 「儲存」 。我們可以在 AppServiceProvider 的 boot 方法中執行此操作：

```
use Illuminate\Support\Facades\Blade;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Blade::if('disk', function (string $value) {
        return config('filesystems.default') === $value;
    });
}
```

一旦定義了自訂條件，就可以在範本中使用它:

```
@disk('local')
    <!-- The application is using the local disk... -->
@elsedisk('s3')
    <!-- The application is using the s3 disk... -->
@else
    <!-- The application is using some other disk... -->
@enddisk

@unlessdisk('local')
    <!-- The application is not using the local disk... -->
@enddisk
```

13 Vite 編譯 Assets

13.1 介紹

Vite 是一款現代前端建構工具，提供極快的開發環境並將你的程式碼捆綁到生產準備的資源中。在使用 Laravel 建構應用程式時，通常會使用 Vite 將你的應用程式的 CSS 和 JavaScript 檔案繫結到生產環境的資源中。

Laravel 通過提供官方外掛和 Blade 指令，與 Vite 完美整合，以載入你的資源進行開發和生產。

> 注意：你正在運行 Laravel Mix 嗎？在新的 Laravel 安裝中，Vite 已經取代了 Laravel Mix 。有關 Mix 的文件，請訪問 Laravel Mix 網站。如果你想切換到 Vite，請參閱我們的 遷移指南。

13.1.1.1 選擇 Vite 還是 Laravel Mix

在轉向 Vite 之前，新的 Laravel 應用程式在打包資產時通常使用 Mix，它由 webpack 支援。Vite 專注於在建構豐富的 JavaScript 應用程式時提供更快、更高效的開發體驗。如果你正在開發單頁面應用程式（SPA），包括使用 Inertia 工具開發的應用程式，則 Vite 是完美選擇。

Vite 也適用於具有 JavaScript “sprinkles” 的傳統伺服器端渲染應用程式，包括使用 Livewire 的應用程式。但是，它缺少 Laravel Mix 支援的某些功能，例如將沒有直接在 JavaScript 應用程式中引用的任意資產複製到建構中的能力。

13.1.1.2 切換回 Mix

如果你使用我們的 Vite 腳手架建立了一個新的 Laravel 應用程式，但需要切換回 Laravel Mix 和 webpack，那麼也沒有問題。請參閱我們的從 Vite 切換到 Mix 的官方指南。

13.2 安裝和設定

> **注意** 以下文件討論如何手動安裝和組態 Laravel Vite 外掛。但是，Laravel 的起始套件已經包含了所有的腳手架，並且是使用 Laravel 和 Vite 開始最快的方式。

13.2.1 安裝 Node

在運行 Vite 和 Laravel 外掛之前，你必須確保已安裝 Node.js（16+）和 NPM：

```
node -v npm -v
```

你可以通過官方 Node 網站的簡單圖形安裝程序輕鬆安裝最新版本的 Node 和 NPM。或者，如果你使用的是 Laravel Sail，可以通過 Sail 呼叫 Node 和 NPM：

```
./vendor/bin/sail node -v ./vendor/bin/sail npm -v
```

13.2.2 安裝 Vite 和 Laravel 外掛

在 Laravel 的全新安裝中，你會在應用程式目錄結構的根目錄下找到一個 package.json 檔案。默認的 package.json 檔案已經包含了你開始使用 Vite 和 Laravel 外掛所需的一切。你可以通過 NPM 安裝應用程式的前端依賴：

```
npm install
```

13.2.3 組態 Vite

Vite 通過項目根目錄中的 `vite.config.js` 檔案進行組態。你可以根據自己的需要自訂此檔案，也可以安裝任何其他外掛，例如 `@vitejs/plugin-vue` 或 `@vitejs/plugin-react`。

Laravel Vite 外掛需要你指定應用程式的入口點。這些入口點可以是 JavaScript 或 CSS 檔案，并包括預處理語言，例如 TypeScript、JSX、TSX 和 Sass。

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
    plugins: [
        laravel([
            'resources/css/app.css',
            'resources/js/app.js',
        ]),
    ],
});
```

如果你正在建構一個單頁應用程式，包括使用 Inertia 建構的應用程式，則最好不要使用 CSS 入口點：

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
    plugins: [
        laravel([
            'resources/css/app.css', // [tl! remove]
            'resources/js/app.js',
        ]),
    ],
});
```

相反，你應該通過 JavaScript 匯入你的 CSS。通常，這將在應用程式的 resources/js/app.js 檔案中完成：

```
import './bootstrap';
import '../css/app.css'; // [tl! add]
```

Laravel 外掛還支援多個入口點和高級組態選項，例如 SSR 入口點。

13.2.3.1 使用安全的開發伺服器

如果你的本地開發 Web 伺服器通過 HTTPS 提供應用程式服務，則可能會遇到連接到 Vite 開發伺服器的問題。

如果你在本地開發中使用 Laravel Valet 並已針對你的應用程式運行 secure 命令，則可以組態 Vite 開發伺服器自動使用 Valet 生成的 TLS 證書：

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
    plugins: [
        laravel({
            // ...
            valetTls: 'my-app.test', // [tl! add]
        }),
    ],
});
```

當使用其他 Web 伺服器時，你應生成一個受信任的證書並手動組態 Vite 使用生成的證書：

```
// ...
```

```
import fs from 'fs'; // [tl! add]

const host = 'my-app.test'; // [tl! add]

export default defineConfig({
    // ...
    server: { // [tl! add]
        host, // [tl! add]
        hmr: { host }, // [tl! add]
        https: { // [tl! add]
            key: fs.readFileSync(`/path/to/${host}.key`), // [tl! add]
            cert: fs.readFileSync(`/path/to/${host}.crt`), // [tl! add]
        }, // [tl! add]
    }, // [tl! add]
});
```

如果你無法為系統生成可信證書，則可以安裝並組態 @vitejs/plugin-basic-ssl 外掛。使用不受信任的證書時，你需要通過在運行 `npm run dev` 命令時按照控制台中的“Local”連結接受 Vite 開發伺服器的證書警告。

13.2.4 載入你的指令碼和樣式

組態了 Vite 入口點後，你只需要在應用程式根範本的 `<head>` 中新增一個 `@vite()` Blade 指令引用它們即可：

```
<!doctype html>
<head>
    {{-- ... --}}

    @vite(['resources/css/app.css', 'resources/js/app.js'])
</head>
```

如果你通過 JavaScript 匯入你的 CSS 檔案，你只需要包含 JavaScript 的入口點：

```
<!doctype html>
<head>
    {{-- ... --}}

    @vite('resources/js/app.js')
</head>
```

`@vite` 指令會自動檢測 Vite 開發伺服器並注入 Vite 客戶端以啟用熱模組替換。在建構模式下，該指令將載入已編譯和版本化的資產，包括任何匯入的 CSS 檔案。

如果需要，在呼叫 `@vite` 指令時，你還可以指定已編譯資產的建構路徑：

```
<!doctype html>
<head>
    {{-- Given build path is relative to public path. --}}

    @vite('resources/js/app.js', 'vendor/courier/build')
</head>
```

13.3 運行 Vite

你可以通過兩種方式運行 Vite。你可以通過 `dev` 命令運行開發伺服器，在本地開發時非常有用。開發伺服器會自動檢測檔案的更改，並立即在任何打開的瀏覽器窗口中反映這些更改。

或者，運行 `build` 命令將版本化並打包應用程式的資產，並準備好部署到生產環境：

Or, running the `build` command will version and bundle your application's assets and get them ready for you to deploy to production:

```
# Run the Vite development server...
```

```
npm run dev

# Build and version the assets for production...
npm run build
```

13.4 使用 JavaScript

13.4.1 別名

默認情況下，Laravel 外掛提供一個常用的別名，以幫助你快速開始並方便地匯入你的應用程式的資產：

```
{
    '@' => '/resources/js'
}
```

你可以通過新增自己的別名到 `vite.config.js` 組態檔案中，覆蓋 `'@'` 別名：

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
    plugins: [
        laravel(['resources/ts/app.tsx']),
    ],
    resolve: {
        alias: {
            '@': '/resources/ts',
        },
    },
});
```

13.4.2 Vue

如果你想要使用 Vue 框架建構前端，那麼你還需要安裝 `@vitejs/plugin-vue` 外掛：

```
npm install --save-dev @vitejs/plugin-vue
```

然後你可以在 `vite.config.js` 組態檔案中包含該外掛。當使用 Laravel 和 Vue 外掛時，還需要一些附加選項：

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';
import vue from '@vitejs/plugin-vue';

export default defineConfig({
    plugins: [
        laravel(['resources/js/app.js']),
        vue({
            template: {
                transformAssetUrls: {
                    // Vue 外掛會重新編寫資產 URL，以便在單檔案元件中引用時，指向 Laravel
web 伺服器。
                    // 將其設定為 `null`，則 Laravel 外掛會將資產 URL 重新編寫為指向
Vite 伺服器。
                    base: null,

                    // Vue 外掛將解析絕對 URL 並將其視為磁碟上檔案的絕對路徑。
                    // 將其設定為 `false`，將保留絕對 URL 不變，以便可以像預期那樣引用公共
目錄中的資源。
                    includeAbsolute: false,
                },
            },
```

```
        }),
    ],
});
```

注意 Laravel 的 起步套件 已經包含了適當的 Laravel、Vue 和 Vite 組態。請查看 Laravel Breeze 以瞭解使用 Laravel、Vue 和 Vite 快速入門的最快方法。

13.4.3 React

如果你想要使用 React 框架建構前端，那麼你還需要安裝 `@vitejs/plugin-react` 外掛：

```
npm install --save-dev @vitejs/plugin-react
```

你可以在 `vite.config.js` 組態檔案中包含該外掛：

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';
import react from '@vitejs/plugin-react';

export default defineConfig({
    plugins: [
        laravel(['resources/js/app.jsx']),
        react(),
    ],
});
```

當使用 Vite 和 React 時，你將需要確保任何包含 JSX 的檔案都有一個 .jsx 和 .tsx 擴展，記住更新入口檔案，如果需要 如上所示。你還需要在現有的 `@vite` 指令旁邊包含額外的 `@viteReactRefresh` Blade 指令。

```
@viteReactRefresh
@vite('resources/js/app.jsx')
```

`@viteReactRefresh` 指令必須在 `@vite` 指令之前呼叫 。

> **注意**
> Laravel 的 起步套件 已經包含了適合的 Laravel、React 和 Vite 組態。查看 Laravel Breeze 以最快的方式開始學習 Laravel、React 和 Vite。

13.4.4 Inertia

Laravel Vite 外掛提供了一個方便的 `resolvePageComponent` 函數，幫助你解決 Inertia 頁面元件。以下是使用 Vue 3 的助手示例；然而，你也可以在其他框架中使用該函數，例如 React：

```
import { createApp, h } from 'vue';
import { createInertiaApp } from '@inertiajs/vue3';
import { resolvePageComponent } from 'laravel-vite-plugin/inertia-helpers';

createInertiaApp({
  resolve: (name) => resolvePageComponent(`./Pages/${name}.vue`,
import.meta.glob('./Pages/**/*.vue')),
  setup({ el, App, props, plugin }) {
    return createApp({ render: () => h(App, props) })
      .use(plugin)
      .mount(el)
  },
});
```

> **注意**
> Laravel 的 起步套件 已經包含了適合的 Laravel、Inertia 和 Vite 組態。查看 Laravel Breeze 以最快的方式開始學習 Laravel、Inertia 和 Vite。

13.4.5 URL 處理

當使用 Vite 並在你的應用程式的 HTML，CSS 和 JS 中引用資源時，有幾件事情需要考慮。首先，如果你使用絕對路徑引用資源，Vite 將不會在建構中包含資源；因此，你需要確認資源在你的公共目錄中是可用的。

在引用相對路徑的資源時，你應該記住這些路徑是相對於它們被引用的檔案的路徑。通過相對路徑引用的所有資源都將被 Vite 重寫、版本化和打包。

參考以下項目結構：

```
public/
  taylor.png
resources/
  js/
    Pages/
      Welcome.vue
  images/
    abigail.png
```

以下示例演示了 Vite 如何處理相對路徑和絕對 URL：

```
<!-- 這個資源不被 Vite 處理，不會被包含在建構中 -->
<img src="/taylor.png">

<!-- 這個資源將由 Vite 重寫、版本化和打包 -->
<img src="../../images/abigail.png">`
```

13.5 使用樣式表

你可以在 Vite 文件 中瞭解有關 Vite 的 CSS 支援更多的資訊。如果你使用 PostCSS 外掛，如 Tailwind，你可以在項目的根目錄中建立一個 `postcss.config.js` 檔案，Vite 會自動應用它：

```
module.exports = {
    plugins: {
        tailwindcss: {},
        autoprefixer: {},
    },
};
```

> **注意** Laravel 的 起始套件 已經包含了正確的 Tailwind、PostCSS 和 Vite 組態。或者，如果你想在不使用我們的起始套件的情況下使用 Tailwind 和 Laravel，請查看 Tailwind 的 Laravel 安裝指南。

13.6 使用 Blade 和 路由

13.6.1 通過 Vite 處理靜態資源

在你的 JavaScript 或 CSS 中引用資源時，Vite 會自動處理和版本化它們。此外，在建構基於 Blade 的應用程式時，Vite 還可以處理和版本化你僅在 Blade 範本中引用的靜態資源。

然而，要實現這一點，你需要通過將靜態資源匯入到應用程式的入口點來讓 Vite 瞭解你的資源。例如，如果你想處理和版本化儲存在 `resources/images` 中的所有圖像和儲存在 `resources/fonts` 中的所有字型，你應該在應用程式的 `resources/js/app.js` 入口點中新增以下內容：

```
import.meta.glob([
  '../images/**',
  '../fonts/**',
]);
```

這些資源將在運行 `npm run build` 時由 Vite 處理。然後，你可以在 Blade 範本中使用 `Vite::asset` 方法引用這些資源，該方法將返回給定資源的版本化 URL：

```
<img src="{{ Vite::asset('resources/images/logo.png') }}">
```

13.6.2 保存刷新

當你的應用程式使用傳統的伺服器端渲染 Blade 建構時，Vite 可以通過在你的應用程式中更改檢視檔案時自動刷新瀏覽器來提高你的開發工作流程。要開始，你可以簡單地將 `refresh` 選項指定為 `true`。

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
    plugins: [
        laravel({
            // ...
            refresh: true,
        }),
    ],
});
```

當 `refresh` 選項為 `true` 時，保存以下目錄中的檔案將在你運行 `npm run dev` 時觸發瀏覽器進行全面的頁面刷新：

- `app/View/Components/**`
- `lang/**`
- `resources/lang/**`
- `resources/views/**`
- `routes/**`

監聽 `routes/**` 目錄對於在應用程式前端中利用 Ziggy 生成路由連結非常有用。

如果這些默認路徑不符合你的需求，你可以指定自己的路徑列表：

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
    plugins: [
        laravel({
            // ...
            refresh: ['resources/views/**'],
        }),
    ],
});
```

在後台，Laravel Vite 外掛使用了 `vite-plugin-full-reload` 包，該包提供了一些高級組態選項，可微調此功能的行為。如果你需要這種等級的自訂，可以提供一個 `config` 定義：

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
    plugins: [
        laravel({
            // ...
            refresh: [{
                paths: ['path/to/watch/**'],
                config: { delay: 300 }
            }],
        }),
    ],
});
```

13.6.3 別名

在 JavaScript 應用程式中建立別名來引用常用目錄是很常見的。但是，你也可以通過在 `Illuminate\Support\Facades\Vite` 類上使用 `macro` 方法來建立在 Blade 中使用的別名。通常，“宏”應在 服務提供商 的 `boot` 方法中定義：

```
/**
 * Bootstrap any application services.
 */
public function boot(): void {
    Vite::macro('image', fn (string $asset) =>
$this->asset("resources/images/{$asset}"));
}
```

定義了宏之後，可以在範本中呼叫它。例如，我們可以使用上面定義的 `image` 宏來引用位於 `resources/images/logo.png` 的資源：

```
<img src="{{ Vite::image('logo.png') }}" alt="Laravel Logo">
```

13.6.4 自訂 base URL

如果你的 Vite 編譯的資產部署到與應用程式不同的域（例如通過 CDN），必須在應用程式的 `.env` 檔案中指定 `ASSET_URL` 環境變數：

```
ASSET_URL=https://cdn.example.com
```

在組態了資源 URL 之後，所有重寫的 URL 都將以組態的值為前綴：

```
https://cdn.example.com/build/assets/app.9dce8d17.js
```

請記住，絕對路徑的 URL 不會被 Vite 重新編寫，因此它們不會被新增前綴。

13.6.5 環境變數

你可以在應用程式的 `.env` 檔案中以 `VITE_` 為前綴注入環境變數以在 JavaScript 中使用：

```
VITE_SENTRY_DSN_PUBLIC=http://example.com
```

你可以通過 `import.meta.env` 對象訪問注入的環境變數：

```
import.meta.env.VITE_SENTRY_DSN_PUBLIC
```

13.6.6 在測試中停用 Vite

Laravel 的 Vite 整合將在運行測試時嘗試解析你的資產，這需要你運行 Vite 開發伺服器或建構你的資產。

如果你希望在測試中模擬 Vite，你可以呼叫 `withoutVite` 方法，該方法對任何擴展 Laravel 的 `TestCase` 類的測試都可用：

```
use Tests\TestCase;

class ExampleTest extends TestCase {
    public function test_without_vite_example(): void {
        $this->withoutVite();

        // ...
    }
}
```

如果你想在所有測試中停用 Vite，可以在基本的 `TestCase` 類上的 `setUp` 方法中呼叫 `withoutVite` 方法：

```
<?php
```

```
namespace Tests;

use Illuminate\Foundation\Testing\TestCase as BaseTestCase;

abstract class TestCase extends BaseTestCase {
    use CreatesApplication;

    protected function setUp(): void// [tl! add:start] {
        parent::setUp();

        $this->withoutVite();
    }// [tl! add:end]
}
```

13.6.7 伺服器端渲染

Laravel Vite 外掛可以輕鬆地設定與 Vite 的伺服器端渲染。要開始使用，請在 `resources/js` 中建立一個 SSR（Server-Side Rendering）入口點，並通過將組態選項傳遞給 Laravel 外掛來指定入口點：

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
    plugins: [
        laravel({
            input: 'resources/js/app.js',
            ssr: 'resources/js/ssr.js',
        }),
    ],
});
```

為確保不遺漏重建 SSR 入口點，我們建議增加應用程式的 `package.json` 中的 「build」 指令碼來建立 SSR 建構：

```
"scripts": {
     "dev": "vite",
     "build": "vite build" // [tl! remove]
     "build": "vite build && vite build --ssr" // [tl! add]
}
```

然後，要建構和啟動 SSR 伺服器，你可以運行以下命令：

```
npm run build
node bootstrap/ssr/ssr.mjs
```

> **注意** Laravel 的 starter kits 已經包含了適當的 Laravel、Inertia SSR 和 Vite 組態。查看 Laravel Breeze ，以獲得使用 Laravel、Inertia SSR 和 Vite 的最快速的方法。

13.7 Script & Style 標籤的屬性

13.7.1 Content Security Policy (CSP) Nonce

如果你希望在你的指令碼和樣式標籤中包含 nonce 屬性，作為你的 Content Security Policy 的一部分，你可以使用自訂 middleware 中的 `useCspNonce` 方法生成或指定一個 nonce：

Copy code

```
<?php

namespace App\Http\Middleware;

use Closure;
```

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Vite;
use Symfony\Component\HttpFoundation\Response;

class AddContentSecurityPolicyHeaders {
    /**
     * Handle an incoming request.
     *
     * @param  \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\
Response)  $next
     */
    public function handle(Request $request, Closure $next, string $role): Response {
        Vite::useCspNonce();

        return $next($request)->withHeaders([
            'Content-Security-Policy' => "script-src 'nonce-".Vite::cspNonce()."'",
        ]);
    }
}
```

呼叫了 `useCspNonce` 方法後，Laravel 將自動在所有生成的指令碼和樣式標籤上包含 `nonce` 屬性。

如果你需要在其他地方指定 nonce，包括 Laravel 的 starter kits 中帶有的 Ziggy `@route` directive 指令，你可以使用 `cspNonce` 方法來檢索它：

```
@routes(nonce: Vite::cspNonce())
```

如果你已經有了一個 nonce，想要告訴 Laravel 使用它，你可以通過將 nonce 傳遞給 `useCspNonce` 方法來實現：

```
Vite::useCspNonce($nonce);
```

13.7.2 子資源完整性 (SRI)

如果你的 `Vite manifest` 中包括資源的完整性雜湊，則 Laravel 將自動向其生成的任何指令碼和樣式標籤中新增 `integrity` 屬性，以執行 子資源完整性。默認情況下，Vite 不包括其清單中的 `integrity` 雜湊，但是你可以通過安裝 `vite-plugin-manifest-sri` NPM 外掛來啟用它：

```
npm install --save-dev vite-plugin-manifest-sri
```

然後，在你的 `vite.config.js` 檔案中啟用此外掛：

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';
import manifestSRI from 'vite-plugin-manifest-sri';// [tl! add]

export default defineConfig({
    plugins: [
        laravel({
            // ...
        }),
        manifestSRI(),// [tl! add]
    ],
});
```

如果需要，你也可以自訂清單中的完整性雜湊鍵：

```
use Illuminate\Support\Facades\Vite;

Vite::useIntegrityKey('custom-integrity-key');
```

如果你想完全停用這個自動檢測，你可以將 `false` 傳遞給 `useIntegrityKey` 方法：

```
Vite::useIntegrityKey(false);
```

13.7.3 任意屬性

如果你需要在指令碼和樣式標籤中包含其他屬性，例如 `data-turbo-track` 屬性，你可以通過 `useScriptTagAttributes` 和 `useStyleTagAttributes` 方法指定它們。通常，這些方法應從一個服務提供程序中呼叫：

```
use Illuminate\Support\Facades\Vite;

Vite::useScriptTagAttributes([
    'data-turbo-track' => 'reload', // 為屬性指定一個值...
    'async' => true, // 在不使用值的情況下指定屬性...
    'integrity' => false, // 排除一個將被包含的屬性...
]);

Vite::useStyleTagAttributes([
    'data-turbo-track' => 'reload',
]);
```

如果你需要有條件地新增屬性，你可以傳遞一個回呼函數，它將接收到資產源路徑、它的 URL、它的清單塊和整個清單：

```
use Illuminate\Support\Facades\Vite;

Vite::useScriptTagAttributes(fn (string $src, string $url, array|null $chunk, array|null
$manifest) => [
    'data-turbo-track' => $src === 'resources/js/app.js' ? 'reload' : false,
]);

Vite::useStyleTagAttributes(fn (string $src, string $url, array|null $chunk, array|null
$manifest) => [
    'data-turbo-track' => $chunk && $chunk['isEntry'] ? 'reload' : false,
]);
```

> **警告** 在 Vite 開發伺服器執行階段，`$chunk` 和 `$manifest` 參數將為 `null`。

13.8 高級定製

默認情況下，Laravel 的 Vite 外掛使用合理的約定，適用於大多數應用，但是有時你可能需要自訂 Vite 的行為。為了啟用額外的自訂選項，我們提供了以下方法和選項，可以用於替代 `@vite` Blade 指令：

```
<!doctype html>
<head>
    {{-- ... --}}

    {{
        Vite::useHotFile(storage_path('vite.hot')) // 自訂 "hot" 檔案...
            ->useBuildDirectory('bundle') // 自訂建構目錄...
            ->useManifestFilename('assets.json') // 自訂清單檔案名稱...
            ->withEntryPoints(['resources/js/app.js']) // 指定入口點...
    }}
</head>
```

然後，在 `vite.config.js` 檔案中，你應該指定相同的組態：

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
    plugins: [
        laravel({
            hotFile: 'storage/vite.hot', // 自訂 "hot" 檔案...
            buildDirectory: 'bundle', // 自訂建構目錄...
            input: ['resources/js/app.js'], // 指定入口點...
```

```
        }),
    ],
    build: {
      manifest: 'assets.json', // 自訂清單檔案名稱...
    },
});
```

13.8.1 修正開發伺服器 URL

Vite 生態系統中的某些外掛默認假設以正斜槓開頭的 URL 始終指向 Vite 開發伺服器。然而，由於 Laravel 整合的性質，實際情況並非如此。

例如，`vite-imagetools` 外掛在 Vite 服務時，你的資產時會輸出以下類似的 URL：

```
<img src="/@imagetools/f0b2f404b13f052c604e632f2fb60381bf61a520">
```

`vite-imagetools` 外掛期望輸出 URL 將被 Vite 攔截，並且外掛可以處理所有以 `/@imagetools` 開頭的 URL。如果你正在使用期望此行為的外掛，則需要手動糾正 URL。你可以在 `vite.config.js` 檔案中使用 `transformOnServe` 選項來實現。

在這個例子中，我們將在生成的程式碼中的所有 `/@imagetools` 錢加上開發伺服器的 URL：

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';
import { imagetools } from 'vite-imagetools';

export default defineConfig({
    plugins: [
        laravel({
            // ...
            transformOnServe: (code, devServerUrl) => code.replaceAll('/@imagetools',
devServerUrl+'/@imagetools'),
        }),
        imagetools(),
    ],
});
```

現在，在 Vite 提供資產服務時，它會輸出 URL 指向 Vite 開發伺服器：

```
- <img src="/@imagetools/f0b2f404b13f052c604e632f2fb60381bf61a520"><!-- [tl! remove] -->
+ <img src="http://[::1]:5173/@imagetools/f0b2f404b13f052c604e632f2fb60381bf61a520"><!--
[tl! add] -->
```

14 生成 URL

14.1 簡介

Laravel 提供了幾個輔助函數來為應用程式生成 URL。主要用於在範本和 API 響應中建構 URL 或者在應用程式的其它部分生成重新導向響應。

14.2 基礎

14.2.1 生成基礎 URLs

輔助函數 `url` 可以用於應用的任何一個 URL。生成的 URL 將自動使用當前請求中的方案 (HTTP 或 HTTPS) 和主機：

```
$post = App\Models\Post::find(1);

echo url("/posts/{$post->id}");

// http://example.com/posts/1
```

14.2.2 訪問當前 URL

如果沒有給輔助函數 `url` 提供路徑，則會返回一個 `Illuminate\Routing\UrlGenerator` 實例，來允許你訪問有關當前 URL 的資訊：

```
// 獲取當前 URL 沒有 query string...
echo url()->current();

// 獲取當前 URL 包括 query string...
echo url()->full();

// 獲取上個請求 URL
echo url()->previous();
```

上面的這些方法都可以通過 URL facade 訪問:

```
use Illuminate\Support\Facades\URL;

echo URL::current();
```

14.3 命名路由的 URLs

輔助函數 `route` 可以用於生成指定 命名路由 的 URLs。 命名路由生成的 URLs 不與路由上定義的 URL 相耦合。因此，就算路由的 URL 有任何改變，都不需要對 `route` 函數呼叫進行任何更改。例如，假設你的應用程式包含以下路由：

```
Route::get('/post/{post}', function (Post $post) {
    // ...
})->name('post.show');
```

要生成此路由的 URL ，可以像這樣使用輔助函數 `route` ：

```
echo route('post.show', ['post' => 1]);
```

```
// http://example.com/post/1
```

當然，輔助函數 `route` 也可以用於為具有多個參數的路由生成 URL：

```
Route::get('/post/{post}/comment/{comment}', function (Post $post, Comment $comment) {
    // ...
})->name('comment.show');

echo route('comment.show', ['post' => 1, 'comment' => 3]);

// http://example.com/post/1/comment/3
```

任何與路由定義參數對應不上的附加陣列元素都將新增到 URL 的查詢字串中：

```
echo route('post.show', ['post' => 1, 'search' => 'rocket']);

// http://example.com/post/1?search=rocket
```

14.3.1.1 Eloquent Models

你通常使用 Eloquent 模型 的主鍵生成 URL。因此，您可以將 Eloquent 模型作為參數值傳遞。 `route` 輔助函數將自動提取模型的主鍵：

```
echo route('post.show', ['post' => $post]);
```

14.3.2 簽名 URLs

Laravel 允許你輕鬆地為命名路徑建立「簽名」 URLs，這些 URLs 在查詢字串後附加了「簽名」雜湊，允許 Laravel 驗證 URL 自建立以來未被修改過。 簽名 URLs 對於可公開訪問但需要一層防止 URL 操作的路由特別有用。

例如，你可以使用簽名 URLs 來實現通過電子郵件傳送給客戶的公共「取消訂閱」連結。要建立指向路徑的簽名 URL ，請使用 `URL` facade 的 `signedRoute` 方法：

```
use Illuminate\Support\Facades\URL;

return URL::signedRoute('unsubscribe', ['user' => 1]);
```

如果要生成具有有效期的臨時簽名路由 URL，可以使用以下 `temporarySignedRoute` 方法，當 Laravel 驗證一個臨時的簽名路由 URL 時，它會確保編碼到簽名 URL 中的過期時間戳沒有過期：

```
use Illuminate\Support\Facades\URL;

return URL::temporarySignedRoute(
    'unsubscribe', now()->addMinutes(30), ['user' => 1]
);
```

14.3.2.1 驗證簽名路由請求

要驗證傳入請求是否具有有效簽名，你應該對傳入的 `Illuminate\Http\Request` 實例中呼叫 `hasValidSignature` 方法：

```
use Illuminate\Http\Request;

Route::get('/unsubscribe/{user}', function (Request $request) {
    if (! $request->hasValidSignature()) {
        abort(401);
    }

    // ...
})->name('unsubscribe');
```

有時，你可能需要允許你的應用程式前端將資料附加到簽名 URL，例如在執行客戶端分頁時。因此，你可以指定在使用 `hasValidSignatureWhileIgnoring` 方法驗證簽名 URL 時應忽略的請求查詢參數。請記住，忽

略參數將允許任何人根據請求修改這些參數：

```
if (! $request->hasValidSignatureWhileIgnoring(['page', 'order'])) {
    abort(401);
}
```

或者，你可以將 `Illuminate\Routing\Middleware\ValidateSignature` 中介軟體 分配給路由。如果它不存在，則應該在 HTTP 核心的 `$middlewareAliases` 陣列中為此中介軟體分配一個鍵：

```
/**
 * The application's middleware aliases.
 *
 * Aliases may be used to conveniently assign middleware to routes and groups.
 *
 * @var array<string, class-string|string>
 */
protected $middlewareAliases = [
    'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,
];
```

一旦在核心中註冊了中介軟體，就可以將其附加到路由。如果傳入的請求沒有有效的簽名，中介軟體將自動返回 `403` HTTP 響應：

```
Route::post('/unsubscribe/{user}', function (Request $request) {
    // ...
})->name('unsubscribe')->middleware('signed');
```

14.3.2.2 響應無效的簽名路由

當有人訪問已過期的簽名 URL 時，他們將收到一個通用的錯誤頁面，顯示 `403` HTTP 狀態程式碼。然而，你可以通過在異常處理程序中為 `InvalidSignatureException` 異常定義自訂 “可渲染” 閉包來自訂此行為。這個閉包應該返回一個 HTTP 響應：

```
use Illuminate\Routing\Exceptions\InvalidSignatureException;

/**
 * 為應用程式註冊異常處理回呼
 */
public function register(): void
{
    $this->renderable(function (InvalidSignatureException $e) {
        return response()->view('error.link-expired', [], 403);
    });
}
```

14.4 controller 行為的 URL

`action` 功能可以為給定的 controller 行為生成 URL。

```
use App\Http\Controllers\HomeController;

$url = action([HomeController::class, 'index']);
```

如果 controller 方法接收路由參數，你可以通過第二個參數傳遞：

```
$url = action([UserController::class, 'profile'], ['id' => 1]);
```

14.5 預設值

對於某些應用程式，你可能希望為某些 URL 參數的請求範圍指定預設值。例如，假設有些路由定義了 `{locale}` 參數：

```
Route::get('/{locale}/posts', function () {
```

```
    // ...
})->name('post.index');
```

每次都通過 `locale` 來呼叫輔助函數 `route` 也是一件很麻煩的事情。因此，使用 `URL::defaults` 方法定義這個參數的預設值，可以讓該參數始終存在當前請求中。然後就能從 路由中介軟體 呼叫此方法來訪問當前請求：

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\URL;
use Symfony\Component\HttpFoundation\Response;

class SetDefaultLocaleForUrls
{
    /**
     * 處理傳入的請求
     *
     * @param  \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\
Response)  $next
     */
    public function handle(Request $request, Closure $next): Response
    {
        URL::defaults(['locale' => $request->user()->locale]);

        return $next($request);
    }
}
```

一旦設定了 `locale` 參數的預設值，你就不再需要通過輔助函數 `route` 生成 URL 時傳遞它的值。

14.5.1.1 默認 URL & 中介軟體優先順序

設定 URL 的預設值會影響 Laravel 對隱式模型繫結的處理。因此，你應該通過設定中介軟體優先順序來確保在 Laravel 自己的 `SubstituteBindings` 中介軟體執行之前設定 URL 的預設值。你可以通過在你的應用的 HTTP kernel 檔案中的 `$middlewarePriority` 屬性裡把你的中介軟體放在 `SubstituteBindings` 中介軟體之前。

`$middlewarePriority` 這個屬性在 `Illuminate\Foundation\Http\Kernel` 這個基類裡。你可以複製一份到你的應用程式的 HTTP kernel 檔案中以便做修改:

```
/**
 * 根據優先順序排序的中介軟體列表
 *
 * 這將保證非全域中介軟體按照既定順序排序
 *
 * @var array
 */
protected $middlewarePriority = [
    // ...
    \App\Http\Middleware\SetDefaultLocaleForUrls::class,
    \Illuminate\Routing\Middleware\SubstituteBindings::class,
    // ...
];
```

15 HTTP Session

15.1 簡介

由於 HTTP 驅動的應用程式是無狀態的，Session 提供了一種在多個請求之間儲存有關使用者資訊的方法，這類資訊一般都儲存在後續請求可以訪問的持久儲存 / 後端中。

Laravel 通過同一個可讀性強的 API 處理各種自帶的後台驅動程式。支援諸如比較熱門的 Memcached、 Redis 和資料庫。

15.1.1 組態

Session 的組態檔案儲存在 `config/session.php` 檔案中。請務必查看此檔案中對於你而言可用的選項。默認情況下，Laravel 為絕大多數應用程式組態的 Session 驅動為 `file` 驅動，它適用於大多數程序。如果你的應用程式需要在多個 Web 伺服器之間進行負載平衡，你應該選擇一個所有伺服器都可以訪問的集中式儲存，例如 Redis 或資料庫。

Session`driver` 的組態預設了每個請求儲存 Session 資料的位置。Laravel 自帶了幾個不錯而且開箱即用的驅動：

- `file` - Sessions 儲存在 `storage/framework/sessions`。
- `cookie` - Sessions 被儲存在安全加密的 cookie 中。
- `database` - Sessions 被儲存在關係型資料庫中。
- `memcached` / `redis` - Sessions 被儲存在基於快取記憶體的儲存系統中。
- `dynamodb` - Sessions 被儲存在 AWS DynamoDB 中。
- `array` - Sessions 儲存在 PHP 陣列中，但不會被持久化。

  技巧
  陣列驅動一般用於測試並且防止儲存在 Session 中的資料被持久化。

15.1.2 驅動先決條件

15.1.2.1 資料庫

使用 `database`Session 驅動時，你需要建立一個記錄 Session 的表。下面是 `Schema` 的聲明示例：

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::create('sessions', function (Blueprint $table) {
    $table->string('id')->primary();
    $table->foreignId('user_id')->nullable()->index();
    $table->string('ip_address', 45)->nullable();
    $table->text('user_agent')->nullable();
    $table->text('payload');
    $table->integer('last_activity')->index();
});
```

你可以使用 Artisan 命令 `session:table` 生成這個遷移。瞭解更多資料庫遷移，請查看完整的文件遷移文件:

```
php artisan session:table

php artisan migrate
```

15.1.2.2 Redis

在 Laravel 使用 Redis Session 驅動前，你需要安裝 PhpRedis PHP 擴展，可以通過 PECL 或者 通過 Composer 安裝這個 `predis/predis` 包 (~1.0)。更多關於 Redis 組態資訊，查詢 Laravel 的 Redis 文件.

> **技巧**
> 在 `session` 組態檔案裡，`connection` 選項可以用來設定 Session 使用 Redis 連接方式。

15.2 使用 Session

15.2.1 獲取資料

在 Laravel 中有兩種基本的 Session 使用方式：全域 `session` 助手函數和通過 `Request` 實例。首先看下通過 `Request` 實例訪問 Session , 它可以隱式繫結路由閉包或者 controller 方法。記住，Laravel 會自動注入 controller 方法的依賴。服務容器：

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * 顯示指定使用者個人資料。
     */
    public function show(Request $request, string $id): View
    {
        $value = $request->session()->get('key');

        // ...

        $user = $this->users->find($id);

        return view('user.profile', ['user' => $user]);
    }
}
```

當你從 Session 獲取資料時，你也可以在 `get` 方法第二個參數里傳遞一個 default 預設值，如果 Session 裡不存在鍵值對 key 的資料結果，這個預設值就會返回。如果你傳遞給 `get` 方法一個閉包作為預設值，這個閉包會被執行並且返回結果：

```
$value = $request->session()->get('key', 'default');

$value = $request->session()->get('key', function () {
    return 'default';
});
```

15.2.1.1 全域 Session 助手函數

你也可以在 Session 裡使用 PHP 全域 `session` 函數獲取和儲存資料。當這個 `session` 函數以一個單獨的字串形式被呼叫時，它將會返回這個 Session 鍵值對的結果。當函數以 key / value 陣列形式被呼叫時，這些值會被儲存在 Session 裡：

```
Route::get('/home', function () {
    // 從 Session 獲取資料 ...
```

```
    $value = session('key');

    // 設定預設值...
    $value = session('key', 'default');

    // 在 Session 裡儲存一段資料 ...
    session(['key' => 'value']);
});
```

> **技巧**
> 通過 HTTP 請求實例與通過 `session` 助手函數方式使用 Session 之間沒有實際區別。兩種方式都是可的測試，你所有的測試用例中都可以通過 `assertSessionHas` 方法進行斷言。

15.2.1.2 獲取所有 Session 資料

如果你想要從 Session 裡獲取所有資料，你可以使用 `all` 方法：

```
$data = $request->session()->all();
```

15.2.1.3 判斷 Session 裡是否存在條目

判斷 Session 裡是否存在一個條目，你可以使用 `has` 方法。如果條目存在 `has`，方法返回 `true` 不存在則返回 `null`：

```
if ($request->session()->has('users')) {
    // ...
}
```

判斷 Session 裡是否存在一個即使結果值為 `null` 的條目，你可以使用 `exists` 方法：

```
if ($request->session()->exists('users')) {
    // ...
}
```

要確定某個條目是否在 session 中不存在，你可以使用 `missing` 方法。如果條目不存在，`missing` 方法返回 `true`：

```
if ($request->session()->missing('users')) {
    // ...
}
```

15.2.2 儲存資料

Session 裡儲存資料，你通常將使用 Request 實例中的 `put` 方法或者 `session` 助手函數：

```
// 通過 Request 實例儲存 ...
$request->session()->put('key', 'value');

// 通過全域 Session 助手函數儲存 ...
session(['key' => 'value']);
```

15.2.2.1 Session 儲存陣列

`push` 方法可以把一個新值推入到以陣列形式儲存的 session 值裡。例如：如果 `user.teams` 鍵值對有一個關於團隊名字的陣列，你可以推入一個新值到這個陣列裡：

```
$request->session()->push('user.teams', 'developers');
```

15.2.2.2 獲取 & 刪除條目

`pull` 方法會從 Session 裡獲取並且刪除一個條目，只需要一步如下：

```
$value = $request->session()->pull('key', 'default');
```

15.2.2.3 遞增 / 遞減 session 值

如果你的 Session 資料裡有整形你希望進行加減操作，可以使用 `increment` 和 `decrement` 方法：

```
$request->session()->increment('count');

$request->session()->increment('count', $incrementBy = 2);

$request->session()->decrement('count');

$request->session()->decrement('count', $decrementBy = 2);
```

15.2.3 快閃記憶體資料

有時你可能想在 Session 裡為下次請求儲存一些條目。你可以使用 `flash` 方法。使用這個方法，儲存在 Session 的資料將立即可用並且會保留到下一個 HTTP 請求期間，之後會被刪除。快閃記憶體資料主要用於短期的狀態消息：

```
$request->session()->flash('status', 'Task was successful!');
```

如果你需要為多次請求持久化快閃記憶體資料，可以使用 `reflash` 方法，它會為一個額外的請求保持住所有的快閃記憶體資料，如果你僅需要保持特定的快閃記憶體資料，可以使用 `keep` 方法：

```
$request->session()->reflash();

$request->session()->keep(['username', 'email']);
```

如果你僅為了當前的請求持久化快閃記憶體資料，可以使用 `now` 方法：

```
$request->session()->now('status', 'Task was successful!');
```

15.2.4 刪除資料

`forget` 方法會從 Session 刪除一些資料。如果你想刪除所有 Session 資料，可以使用 `flush` 方法：

```
// 刪除一個單獨的鍵值對 ...
$request->session()->forget('name');

// 刪除多個 鍵值對 ...
$request->session()->forget(['name', 'status']);

$request->session()->flush();
```

15.2.5 重新生成 Session ID

重新生成 Session ID 經常被用來阻止惡意使用者使用 session fixation 攻擊你的應用。

如果你正在使用入門套件或 Laravel Fortify 中的任意一種， Laravel 會在認證階段自動生成 Session ID；然而如果你需要手動重新生成 Session ID ，可以使用 `regenerate` 方法：

```
$request->session()->regenerate();
```

如果你需要重新生成 Session ID 並同時刪除所有 Session 裡的資料，可以使用 `invalidate` 方法：

```
$request->session()->invalidate();
```

15.3 Session 阻塞

注意

> 應用 Session 阻塞功能，你的應用必須使用一個支援原子鎖的快取驅動。目前，可用的快取驅動有 `memcached`、 `dynamodb`、 `redis` 和 `database` 等。另外，你可能不會使用 `cookie` Session 驅動。

默認情況下，Laravel 允許使用同一 Session 的請求並行地執行，舉例來說，如果你使用一個 JavaScript HTTP 庫向你的應用執行兩次 HTTP 請求，它們將同時執行。對多數應用這不是問題，然而 在一小部分應用中可能出現 Session 資料丟失，這些應用會向兩個不同的應用端並行請求，並同時寫入資料到 Session。

為瞭解決這個問題，Laravel 允許你限制指定 Session 的並行請求。首先，你可以在路由定義時使用 `block` 鏈式方法。在這個示例中，一個到`/profile` 的路由請求會拿到一把 Session 鎖。當它處在鎖定狀態時，任何使用相同 Session ID 的到`/profile` 或`/order` 的路由請求都必須等待，直到第一個請求處理完成後再繼續執行：

```
Route::post('/profile', function () {
    // ...
})->block($lockSeconds = 10, $waitSeconds = 10)

Route::post('/order', function () {
    // ...
})->block($lockSeconds = 10, $waitSeconds = 10)
```

`block` 方法接受兩個可選參數。`block` 方法接受的第一個參數是 Session 鎖釋放前應該持有的最大秒數。當然，如果請求在此時間之前完成執行，鎖將提前釋放。

`block` 方法接受的第二個參數是請求在試圖獲得 Session 鎖時應該等待的秒數。如果請求在給定的秒數內無法獲得 session 鎖，將拋出 `Illuminate\Contracts\Cache\LockTimeoutException` 異常。

如果不傳參，那麼 Session 鎖默認鎖定最大時間是 10 秒，請求鎖最大的等待時間也是 10 秒：

```
Route::post('/profile', function () {
    // ...
})->block()
```

15.4 新增自訂 Session 驅動

15.4.1.1 實現驅動

如果現存的 Session 驅動不能滿足你的需求，Laravel 允許你自訂 Session Handler。你的自訂驅動應實現 PHP 內建的 `SessionHandlerInterface`。這個介面僅包含幾個方法。以下是 MongoDB 驅動實現的程式碼片段：

```
<?php

namespace App\Extensions;

class MongoSessionHandler implements \SessionHandlerInterface
{
    public function open($savePath, $sessionName) {}
    public function close() {}
    public function read($sessionId) {}
    public function write($sessionId, $data) {}
    public function destroy($sessionId) {}
    public function gc($lifetime) {}
}
```

> **技巧**
> Laravel 沒有內建存放擴展的目錄，你可以放置在任意目錄下，這個示例裡，我們建立了一個 `Extensions` 目錄存放 `MongoSessionHandler`。

由於這些方法的含義並非通俗易懂，因此我們快速瀏覽下每個方法：

- `open` 方法通常用於基於檔案的 Session 儲存系統。因為 Laravel 附帶了一個 `file` Session 驅動。你無須在裡面寫任何程式碼。可以簡單地忽略掉。

- `close` 方法跟 `open` 方法很像，通常也可以忽略掉。對大多數驅動來說，它不是必須的。
- `read` 方法應返回與給定的`$sessionId` 關聯的 Session 資料的字串格式。在你的驅動中獲取或儲存 Session 資料時，無須作任何序列化和編碼的操作，Laravel 會自動為你執行序列化。
- `write` 方法將與`$sessionId` 關聯的給定的`$data` 字串寫入到一些持久化儲存系統，如 MongoDB 或者其他你選擇的儲存系統。再次，你無須進行任何序列化操作，Laravel 會自動為你處理。
- `destroy` 方法應可以從持久化儲存中刪除與`$sessionId` 相關聯的資料。
- `gc` 方法應可以銷毀給定的`$lifetime`（UNIX 時間戳格式）之前的所有 Session 資料。對於像 Memcached 和 Redis 這類擁有過期機制的系統來說，本方法可以置空。

15.4.1.2 註冊驅動

一旦你的驅動實現了，需要註冊到 Laravel 。在 Laravel 中新增額外的驅動到 Session 後端 ，你可以使用 `Session` Facade 提供的 `extend` 方法。你應該在服務提供者中的 `boot` 方法中呼叫 `extend` 方法。可以通過已有的 `App\Providers\AppServiceProvider` 或建立一個全新的服務提供者執行此操作：

```
<?php

namespace App\Providers;

use App\Extensions\MongoSessionHandler;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Support\Facades\Session;
use Illuminate\Support\ServiceProvider;

class SessionServiceProvider extends ServiceProvider
{
    /**
     * 註冊任意應用服務。
     */
    public function register(): void
    {
        // ...
    }

    /**
     * 啟動任意應用服務。
     */
    public function boot(): void
    {
        Session::extend('mongo', function (Application $app) {
            // 返回一個 SessionHandlerInterface 介面的實現 ...
            return new MongoSessionHandler;
        });
    }
}
```

一旦 Session 驅動註冊完成，就可以在 `config/session.php` 組態檔案選擇使用 `mongo` 驅動。

16 表單驗證

16.1 簡介

Laravel 提供了幾種不同的方法來驗證傳入應用程式的資料。最常見的做法是在所有傳入的 HTTP 請求中使用 `validate` 方法。同時，我們還將討論其他驗證方法。

Laravel 包含了各種方便的驗證規則，你可以將它們應用於資料，甚至可以驗證給定資料庫表中的值是否唯一。我們將詳細介紹每個驗證規則，以便你熟悉 Laravel 的所有驗證功能。

16.2 快速開始

為了瞭解 Laravel 強大的驗證功能，我們來看一個表單驗證並將錯誤消息展示給使用者的完整示例。通過閱讀概述，這將會對你如何使用 Laravel 驗證傳入的請求資料有一個很好的理解：

16.2.1 定義路由

首先，假設我們在 `routes/web.php` 路由檔案中定義了下面這些路由：

```
use App\Http\Controllers\PostController;
Route::get('/post/create', [PostController::class, 'create']);
Route::post('/post', [PostController::class, 'store']);
```

`GET` 路由會顯示一個供使用者建立新部落格文章的表單，而 `POST` 路由會將新的部落格文章儲存到資料庫中。

16.2.2 建立 controller

接下來，讓我們一起來看看處理這些路由的簡單 controller 。我們暫時留空了 store 方法：

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\View\View;

class PostController extends Controller
{
    /**
     * 部落格的表單檢視
     */
    public function create(): View
    {
        return view('post.create');
    }

    /**
     * 儲存部落格的 Action
     */
    public function store(Request $request): RedirectResponse
    {
        // 驗證並且執行儲存邏輯
```

```
        $post = /** ... */

        return to_route('post.show', ['post' => $post->id]);
    }
}
```

16.2.3 編寫驗證邏輯

現在我們開始在 `store` 方法中編寫用來驗證新的部落格文章的邏輯程式碼。為此，我們將使用 `Illuminate\Http\Request` 類提供的 `validate` 方法。如果驗證通過，你的程式碼會繼續正常運行。如果驗證失敗，則會拋出 `Illuminate\Validation\ValidationException` 異常，並自動將對應的錯誤響應返回給使用者。

如果在傳統 HTTP 請求期間驗證失敗，則會生成對先前 URL 的重新導向響應。如果傳入的請求是 XHR，將將返回包含驗證錯誤資訊的 JSON 響應。

為了深入理解 `validate` 方法，讓我們接著回到 `store` 方法中：

```
/**
 * 儲存一篇新的部落格文章。
 */
public function store(Request $request): RedirectResponse
{
    $validated = $request->validate([
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ]);

    // 部落格文章驗證通過...

    return redirect('/posts');
}
```

如你所見，驗證規則被傳遞到 `validate` 方法中。不用擔心——所有可用的驗證規則均已 存檔。 另外再提醒一次，如果驗證失敗，會自動生成一個對應的響應。如果驗證通過，那我們的 controller 會繼續正常運行。

另外，驗證規則可以使用陣列，而不是單個 `|` 分隔的字串：

```
$validatedData = $request->validate([
    'title' => ['required', 'unique:posts', 'max:255'],
    'body' => ['required'],
]);
```

此外，你可以使用 `validateWithBag` 方法來驗證請求，並將所有錯誤資訊儲存在一個 命名錯誤資訊包：

```
$validatedData = $request->validateWithBag('post', [
    'title' => ['required', 'unique:posts', 'max:255'],
    'body' => ['required'],
]);
```

16.2.3.1 在首次驗證失敗時停止運行

有時候我們希望某個欄位在第一次驗證失敗後就停止運行驗證規則，只需要將 `bail` 新增到規則中：

```
$request->validate([
    'title' => 'bail|required|unique:posts|max:255',
    'body' => 'required',
]);
```

在這個例子中，如果 `title` 欄位沒有通過 `unique` 規則，那麼不會繼續驗證 `max` 規則。規則會按照分配時的順序來驗證。

16.2.3.2 巢狀欄位的說明

如果傳入的 HTTP 請求包含「巢狀」參數，你可以在驗證規則中使用 . 語法來指定這些參數：

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'author.name' => 'required',
    'author.description' => 'required',
]);
```

另外，如果你的欄位名稱包含點，則可以通過使用反斜槓將點轉義，以防止將其解釋為 . 語法：

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'v1\.0' => 'required',
]);
```

16.2.4 顯示驗證錯誤資訊

那麼，如果傳入的請求欄位沒有通過驗證規則呢？如前所述，Laravel 會自動將使用者重新導向到之前的位置。此外，所有的驗證錯誤和請求輸入都會自動存入到快閃記憶體 session 中。

`Illuminate\View\Middleware\ShareErrorsFromSession` 中介軟體與應用程式的所有檢視共享一個 `$errors` 變數，該變數由 `web` 中介軟體組提供。當應用該中介軟體時，`$errors` 變數始終在檢視中可用，`$errors` 變數是 `Illuminate\Support\MessageBag` 的實例。更多有關使用該對象的資訊，查看文件

因此，在實例中，當驗證失敗時，使用者將重新導向到 controller `create` 方法，從而在檢視中顯示錯誤消息：

```
<!-- /resources/views/post/create.blade.php -->

<h1>Create Post</h1>

@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

<!-- Create Post Form -->
```

16.2.4.1 在語言檔案中指定自訂消息

Laravel 的內建驗證規則每個都對應一個錯誤消息，位於應用程式的 `lang/en/validation.php` 檔案中。在此檔案中，你將找到每個驗證規則的翻譯條目。你可以根據應用程式的需求隨意更改或修改這些消息。

此外，你可以將此檔案複製到另一個翻譯語言目錄中，以翻譯應用程式語言的消息。要瞭解有關 Laravel 本地化的更多資訊，請查看完整的本地化文件.

> **注意** 默認，Laravel 應用程式框架不包括 `lang` 目錄。如果你想自訂 Laravel 的語言檔案，你可以通過 `lang:publish` Artisan 命令發佈它們。

16.2.4.2 XHR 請求 & 驗證

在如下示例中，我們使用傳統形式將資料傳送到應用程式。但是，許多應用程式從 JavaScript 驅動的前端接收 XHR 請求。在 XHR 請求期間使用 `validate` 方法時，Laravel 將不會生成重新導向響應。相反，Laravel 生成一個包含所有驗證錯誤的 JSON 響應。該 JSON 響應將以 422 HTTP 狀態碼傳送。

16.2.4.3 `@error` 指令

你亦可使用 `@error` Blade 指令方便地檢查給定的屬性是否存在驗證錯誤資訊。在`@error` 指令中，你可以輸出`$message` 變數以顯示錯誤資訊：

```
<!-- /resources/views/post/create.blade.php -->

<label for="title">Post Title</label>

<input id="title"
     type="text"
     name="title"
     class="@error('title') is-invalid @enderror">

@error('title')
     <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

如果你使用命名錯誤包，你可以將錯誤包的名稱作為第二個參數傳遞給`@error` 指令：

```
<input ... class="@error('title', 'post') is-invalid @enderror">
```

16.2.5 回填表單

當 Laravel 由於驗證錯誤而生成重新導向響應時，框架將自動將所有請求的輸入快閃記憶體到 session 中。這樣做是為了方便你在下一個請求期間訪問輸入，並重新填充使用者嘗試提交的表單。

要從先前的請求中檢索快閃記憶體的輸入，請在 `Illuminate\Http\Request` 的實例上呼叫 `old` 方法。`old` 方法將從 session 中提取先前快閃記憶體的輸入資料：

```
$title = $request->old('title');
```

Laravel 還提供了一個全域性的 `old`。如果要在 Blade 範本, 中顯示舊輸入，則使用 `old` 來重新填充表單會更加方便。如果給定欄位不存在舊輸入，則將返回 `null`：

```
<input type="text" name="title" value="{{ old('title') }}">
```

16.2.6 關於可選欄位的注意事項

默認情況下， 在你的 Laravel 應用的全域中介軟體堆疊 `App\Http\Kernel` 類中包含了 `TrimStrings` 和 `ConvertEmptyStringsToNull` 中介軟體。因此，如果你不想讓 `null` 被驗證器標識為非法的話，你需要將「可選」欄位標誌為 `nullable`。例如：

```
$request->validate([
     'title' => 'required|unique:posts|max:255',
     'body' => 'required',
     'publish_at' => 'nullable|date',
]);
```

在此示例中，我們指定 `publish_at` 欄位可以為 `null` 或有效的日期表示。如果沒有將 `nullable` 修飾符新增到規則定義中，則驗證器會將 `null` 視為無效日期。

16.2.7 驗證錯誤響應格式

當您的應用程式拋出 `Illuminate\Validation\ValidationException` 異常，並且傳入的 HTTP 請求希望返回 JSON 響應時，Laravel 將自動為您格式化錯誤消息，並返回 `422 Unprocessable Entity` HTTP 響應。

下面是驗證錯誤的 JSON 響應格式示例。請注意，巢狀的錯誤鍵會被轉換為“點”符號格式：

```
{
```

```
    "message": "The team name must be a string. (and 4 more errors)",
    "errors": {
        "team_name": [
            "The team name must be a string.",
            "The team name must be at least 1 characters."
        ],
        "authorization.role": [
            "The selected authorization.role is invalid."
        ],
        "users.0.email": [
            "The users.0.email field is required."
        ],
        "users.2.email": [
            "The users.2.email must be a valid email address."
        ]
    }
}
```

16.3 表單請求驗證

16.3.1 建立表單請求

對於更複雜的驗證場景，您可能希望建立一個“表單請求”。表單請求是自訂請求類，封裝了自己的驗證和授權邏輯。要建立一個表單請求類，您可以使用 `make:request` Artisan CLI 命令：

```
php artisan make:request StorePostRequest
```

生成的表單請求類將被放置在 `app/Http/Requests` 目錄中。如果此目錄不存在，則在運行 `make:request` 命令時將建立該目錄。Laravel 生成的每個表單請求都有兩個方法：`authorize` 和 `rules`。

你可能已經猜到了，`authorize` 方法負責確定當前已認證使用者是否可以執行請求所代表的操作，而 `rules` 方法返回應用於請求資料的驗證規則：

```
/**
 * 獲取應用於請求的驗證規則。
 *
 * @return array<string, \Illuminate\Contracts\Validation\Rule|array|string>
 */
public function rules(): array
{
    return [
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ];
}
```

> **注意** 你可以在 `rules` 方法的簽名中指定任何你需要的依賴項類型提示。它們將通過 Laravel 的 服務容器 自動解析。

那麼，驗證規則是如何被評估的呢？你只需要在 controller 方法中對請求進行類型提示。在呼叫 controller 方法之前，傳入的表單請求將被驗證，這意味著你不需要在 controller 中新增任何驗證邏輯：

```
/**
 * 儲存新部落格文章。
 */
public function store(StorePostRequest $request): RedirectResponse
{
    // 傳入的請求有效...

    // 檢索已驗證的輸入資料...
    $validated = $request->validated();
```

```
    // Retrieve a portion of the validated input data...
    $validated = $request->safe()->only(['name', 'email']);
    $validated = $request->safe()->except(['name', 'email']);

    // 儲存部落格文章...

    return redirect('/posts');
}
```

如果驗證失敗，將生成重新導向響應以將使用者傳送回其先前的位置。錯誤也將被快閃記憶體到 session 中，以便進行顯示。如果請求是 XHR 請求，則會向使用者返回帶有 422 狀態程式碼的 HTTP 響應，其中包含 JSON 格式的驗證錯誤表示。

16.3.1.1 在表單請求後新增鉤子

如果您想在表單請求「之後」新增驗證鉤子，可以使用 `withValidator` 方法。這個方法接收一個完整的驗證構造器，允許你在驗證結果返回之前呼叫任何方法：

```
use Illuminate\Validation\Validator;

/**
 * 組態驗證實例。
 */
public function withValidator(Validator $validator): void
{
    $validator->after(function (Validator $validator) {
        if ($this->somethingElseIsInvalid()) {
            $validator->errors()->add('field', 'Something is wrong with this field!');
        }
    });
}
```

16.3.1.2 單個驗證規則失敗後停止

通過向您的請求類新增 `stopOnFirstFailure` 屬性，您可以通知驗證器一旦發生單個驗證失敗後，停止驗證所有規則。

```
/**
 * 表示驗證器是否應在第一個規則失敗時停止。
 *
 * @var bool
 */
protected $stopOnFirstFailure = true;
```

16.3.1.3 自訂重新導向

如前所述，當表單請求驗證失敗時，將會生成一個讓使用者返回到先前位置的重新導向響應。當然，您也可以自由定義此行為。如果您要這樣做，可以在表單請求中定義一個 `$redirect` 屬性：

```
/**
 * 如果驗證失敗，使用者應重新導向到的 URI。
 *
 * @var string
 */
protected $redirect = '/dashboard';
```

或者，如果你想將使用者重新導向到一個命名路由，你可以定義一個 `$redirectRoute` 屬性來代替：

```
/**
 * 如果驗證失敗，使用者應該重新導向到的路由。
 *
 * @var string
```

```
 */
protected $redirectRoute = 'dashboard';
```

16.3.2 表單請求授權驗證

表單請求類內也包含了 `authorize` 方法。在這個方法中，您可以檢查經過身份驗證的使用者確定其是否具有更新給定資源的權限。例如，您可以判斷使用者是否擁有更新文章評論的權限。最有可能的是，您將通過以下方法與你的 授權與策略 進行互動：

```
use App\Models\Comment;

/**
 * 確定使用者是否有請求權限。
 */
public function authorize(): bool
{
    $comment = Comment::find($this->route('comment'));

    return $comment && $this->user()->can('update', $comment);
}
```

由於所有的表單請求都是繼承了 Laravel 中的請求基類，所以我們可以使用 `user` 方法去獲取當前認證登錄的使用者。同時請注意上述例子中對 `route` 方法的呼叫。這個方法允許你在被呼叫的路由上獲取其定義的 URI 參數，譬如下面例子中的 `{comment}` 參數：

```
Route::post('/comment/{comment}');
```

因此，如果您的應用程式正在使用 路由模型繫結，則可以通過將解析的模型作為請求從而讓您的程式碼更加簡潔：

```
return $this->user()->can('update', $this->comment);
```

如果 `authorize` 方法返回 `false`，則會自動返回一個包含 403 狀態碼的 HTTP 響應，也不會運行 controller 的方法。

如果您打算在應用程式的其它部分處理請求的授權邏輯，只需從 `authorize` 方法返回 `true`：

```
/**
 * 判斷使用者是否有請求權限。
 */
public function authorize(): bool
{
    return true;
}
```

> **注意** 你可以向 `authorize` 方法傳入所需的任何依賴項。它們會自動被 Laravel 提供的 服務容器 自動解析。

16.3.3 自訂錯誤消息

你可以通過重寫表單請求的 `messages` 方法來自訂錯誤消息。此方法應返回屬性 / 規則對及其對應錯誤消息的陣列：

```
/**
 * 獲取已定義驗證規則的錯誤消息。
 *
 * @return array<string, string>
 */
public function messages(): array
{
    return [
        'title.required' => 'A title is required',
```

```
            'body.required' => 'A message is required',
        ];
    }
```

16.3.3.1 自訂驗證屬性

Laravel 的許多內建驗證規則錯誤消息都包含 `:attribute` 預留位置。如果您希望將驗證消息的 `:attribute` 部分取代為自訂屬性名稱，則可以重寫 `attributes` 方法來指定自訂名稱。此方法應返回屬性 / 名稱對的陣列：

```
    /**
     * 獲取驗證錯誤的自訂屬性
     *
     * @return array<string, string>
     */
    public function attributes(): array
    {
        return [
            'email' => 'email address',
        ];
    }
```

16.3.4 準備驗證輸入

如果您需要在應用驗證規則之前修改或清理請求中的任何資料，您可以使用 `prepareForValidation` 方法：

```
    use Illuminate\Support\Str;

    /**
     * 準備驗證資料。
     */
    protected function prepareForValidation(): void
    {
        $this->merge([
            'slug' => Str::slug($this->slug),
        ]);
    }
```

同樣地，如果您需要在驗證完成後對任何請求資料進行規範化，您可以使用 `passedValidation` 方法：

```
    use Illuminate\Support\Str;

    /**
     * Handle a passed validation attempt.
     */
    protected function passedValidation(): void
    {
        $this->replace(['name' => 'Taylor']);
    }
```

16.4 手動建立驗證器

如果您不想在請求上使用 `validate` 方法，可以使用 `Validator` 門面 手動建立一個驗證器實例。門面上的 `make` 方法會生成一個新的驗證器實例：

```
    <?php

    namespace App\Http\Controllers;

    use App\Http\Controllers\Controller;
    use Illuminate\Http\RedirectResponse;
```

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Validator;

class PostController extends Controller
{
    /**
     * 儲存新的部落格文章。
     */
    public function store(Request $request): RedirectResponse
    {
        $validator = Validator::make($request->all(), [
            'title' => 'required|unique:posts|max:255',
            'body' => 'required',
        ]);

        if ($validator->fails()) {
            return redirect('post/create')
                        ->withErrors($validator)
                        ->withInput();
        }

        // 獲取驗證後的輸入...
        $validated = $validator->validated();

        // 獲取驗證後輸入的一部分...
        $validated = $validator->safe()->only(['name', 'email']);
        $validated = $validator->safe()->except(['name', 'email']);

        // 儲存部落格文章...

        return redirect('/posts');
    }
}
```

第一個參數傳遞給 `make` 方法的是要驗證的資料。第二個參數是一個應用於資料的驗證規則的陣列。

在確定請求驗證是否失敗之後，您可以使用 `withErrors` 方法將錯誤消息快閃記憶體到 session 中。使用此方法後，`$errors` 變數將自動在重新導向後與您的檢視共享，從而可以輕鬆地將其顯示回使用者。`withErrors` 方法接受驗證器、`MessageBag` 或 PHP 陣列。

16.4.1.1 單個驗證規則失敗後停止

通過向您的請求類新增 `stopOnFirstFailure` 屬性，您可以通知驗證器一旦發生單個驗證失敗後，停止驗證所有規則。

```
if ($validator->stopOnFirstFailure()->fails()) {
    // ...
}
```

16.4.2 自動重新導向

如果您想手動建立驗證器實例，但仍要利用 HTTP 請求的 `validate` 方法提供的自動重新導向，可以在現有驗證器實例上呼叫 `validate` 方法。如果驗證失敗，則會自動重新導向使用者，或者在 XHR 請求的情況下，將返回一個 JSON 響應

```
Validator::make($request->all(), [
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
])->validate();
```

如果驗證失敗，您可以使用 `validateWithBag` 方法將錯誤消息儲存在命名錯誤包中：

```
Validator::make($request->all(), [
```

```
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
])->validateWithBag('post');
```

16.4.3 命名的錯誤包

如果您在同一頁上有多個表單，您可能希望為包含驗證錯誤的 MessageBag 命名，以便檢索特定表單的錯誤消息。為此，將名稱作為第二個參數傳遞給 withErrors：

```
return redirect('register')->withErrors($validator, 'login');
```

你可以通過 $errors 變數訪問命名後的 MessageBag 實例：

```
{{ $errors->login->first('email') }}
```

16.4.4 自訂錯誤消息

如果需要，你可以提供驗證程序實例使用的自訂錯誤消息，而不是 Laravel 提供的默認錯誤消息。有幾種指定自訂消息的方法。首先，您可以將自訂消息作為第三個參數傳遞給 Validator::make 方法：

```
$validator = Validator::make($input, $rules, $messages = [
    'required' => 'The :attribute field is required.',
]);
```

在此示例中，:attribute 預留位置將被驗證中的欄位的實際名稱替換。您也可以在驗證消息中使用其它預留位置。例如：

```
$messages = [
    'same' => 'The :attribute and :other must match.',
    'size' => 'The :attribute must be exactly :size.',
    'between' => 'The :attribute value :input is not between :min - :max.',
    'in' => 'The :attribute must be one of the following types: :values',
];
```

16.4.4.1 為給定屬性指定自訂消息

有時你可能希望只為特定屬性指定自訂錯誤消息。你可以使用 . 表示法。首先指定屬性名稱，然後指定規則：

```
$messages = [
    'email.required' => 'We need to know your email address!',
];
```

16.4.4.2 指定自訂屬性值

Laravel 的許多內建錯誤消息都包含一個 :attribute 預留位置，該預留位置已被驗證中的欄位或屬性的名稱替換。為了自訂用於替換特定欄位的這些預留位置的值，你可以將自訂屬性的陣列作為第四個參數傳遞給 Validator::make 方法：

```
$validator = Validator::make($input, $rules, $messages, [
    'email' => 'email address',
]);
```

16.4.5 驗證後鉤子

驗證器允許你在完成驗證操作後執行附加的回呼。以便你處理下一步的驗證，甚至是往資訊集合中新增更多的錯誤資訊。你可以在驗證器實例上使用 after 方法實現：

```
use Illuminate\Support\Facades;
use Illuminate\Validation\Validator;

$validator = Facades\Validator::make(/* ... */);
```

```
$validator->after(function (Validator $validator) {
    if ($this->somethingElseIsInvalid()) {
        $validator->errors()->add(
            'field', 'Something is wrong with this field!'
        );
    }
});

if ($validator->fails()) {
    // ...
}
```

16.5 處理驗證欄位

在使用表單請求或手動建立的驗證器實例驗證傳入請求資料後，你可能希望檢索經過驗證的請求資料。 這可以通過多種方式實現。 首先，你可以在表單請求或驗證器實例上呼叫 `validated` 方法。 此方法返回已驗證的資料陣列：

```
$validated = $request->validated();

$validated = $validator->validated();
```

或者，你可以在表單請求或驗證器實例上呼叫 `safe` 方法。 此方法返回一個 `Illuminate\Support\ValidatedInput` 的實例。 該實例對象包含 `only`、`except` 和 `all` 方法來檢索已驗證資料的子集或整個已驗證資料陣列：

```
$validated = $request->safe()->only(['name', 'email']);

$validated = $request->safe()->except(['name', 'email']);

$validated = $request->safe()->all();
```

此外， `Illuminate\Support\ValidatedInput` 實例可以像陣列一樣被迭代和訪問：

```
// 迭代驗證資料...
foreach ($request->safe() as $key => $value) {
    // ...
}

// 訪問驗證資料陣列...
$validated = $request->safe();

$email = $validated['email'];
```

`merge` 方法可以給驗證過的資料新增額外的欄位：

```
$validated = $request->safe()->merge(['name' => 'Taylor Otwell']);
```

`collect` 方法以 collection 實例的形式來檢索驗證的資料：

```
$collection = $request->safe()->collect();
```

16.6 使用錯誤消息

在呼叫 `Validator` 實例的 `errors` 方法後，會收到一個 `Illuminate\Support\MessageBag` 實例，用於處理錯誤資訊。自動提供給所有檢視的 `$errors` 變數也是 `MessageBag` 類的一個實例。

16.6.1.1 檢索欄位的第一條錯誤消息

`first` 方法返回給定欄位的第一條錯誤資訊：

```
$errors = $validator->errors();
```

```
echo $errors->first('email');
```

16.6.1.2 檢索一個欄位的所有錯誤資訊

get 方法用於檢索一個給定欄位的所有錯誤資訊，返回值類型為陣列：

```
foreach ($errors->get('email') as $message) {
    // ...
}
```

對於陣列表單欄位，可以使用 * 來檢索每個陣列元素的所有錯誤資訊：

```
foreach ($errors->get('attachments.*') as $message) {
    // ...
}
```

16.6.1.3 檢索所有欄位的所有錯誤資訊

all 方法用於檢索所有欄位的所有錯誤資訊，返回值類型為陣列：

```
foreach ($errors->all() as $message) {
    // ...
}
```

16.6.1.4 判斷欄位是否存在錯誤資訊

has 方法可用於確定一個給定欄位是否存在任何錯誤資訊：

```
if ($errors->has('email')) {
    // ...
}
```

16.6.2 在語言檔案中指定自訂消息

Laravel 內建的驗證規則都有一個錯誤資訊，位於應用程式的 lang/en/validation.php 檔案中。在這個檔案中, 你會發現每個驗證規則都有一個翻譯條目。可以根據你的應用程式的需要，自由地改變或修改這些資訊。

此外, 你可以把這個檔案複製到另一個語言目錄，為你的應用程式的語言翻譯資訊。要瞭解更多關於 Laravel 本地化的資訊, 請查看完整的 本地化。

> **Warning** 默認情況下, Laravel 應用程式的骨架不包括 lang 目錄. 如果你想定製 Laravel 的語言檔案, 可以通過 lang:publish Artisan 命令發佈它們。

16.6.2.1 針對特定屬性的自訂資訊

可以在應用程式的驗證語言檔案中自訂用於指定屬性和規則組合的錯誤資訊。將自訂資訊新增到應用程式的 lang/xx/validation.php 語言檔案的 custom 陣列中：

```
'custom' => [
    'email' => [
        'required' => 'We need to know your email address!',
        'max' => 'Your email address is too long!'
    ],
],
```

16.6.3 在語言檔案中指定屬性

Laravel 內建的錯誤資訊包括一個 `:attribute` 預留位置，它被取代為驗證中的欄位或屬性的名稱。如果你希望你的驗證資訊中的 `:attribute` 部分被替換成一個自訂的值, 可以在 `lang/xx/validation.php` 檔案的 `attributes` 陣列中指定自訂屬性名稱:

```
'attributes' => [
    'email' => 'email address',
],
```

> **Warning** 默認情況下, Laravel 應用程式的骨架不包括 `lang` 目錄. 如果你想定製 Laravel 的語言檔案, 可以通過 `lang:publish` Artisan 命令發佈它們。

16.6.4 指定語言檔案中的值

Laravel 內建的驗證規則錯誤資訊包含一個 `:value` 預留位置，它被替換成請求屬性的當前值。然而, 你可能偶爾需要在驗證資訊的 `:value` 部分替換成自訂的值。 例如，如果 `payment_type` 的值為 `cc` 則需要驗證信用卡號碼:

```
Validator::make($request->all(), [
    'credit_card_number' => 'required_if:payment_type,cc'
]);
```

如果這個驗證規則失敗了，它將產生以下錯誤資訊:

```
The credit card number field is required when payment type is cc.
```

你可以在 `lang/xx/validation.php` 語言檔案中通過定義一個 `values` 陣列來指定一個更友好的提示，而不是顯示 `cc` 作為支付類型值：

```
'values' => [
    'payment_type' => [
        'cc' => 'credit card'
    ],
],
```

> **Warning** 默認情況下, Laravel 應用程式的骨架不包括 `lang` 目錄. 如果你想定製 Laravel 的語言檔案, 你可以通過 `lang:publish` Artisan 命令發佈它們。

定義這個值後，驗證規則將產生以下錯誤資訊：

```
The credit card number field is required when payment type is credit card.
```

16.7 可用的驗證規則

下面是所有可用的驗證規則及其功能的列表:

省略不列印

16.8 有條件新增規則

16.8.1.1 當欄位具有特定值時跳過驗證

有時，您可能希望在給定欄位具有特定值時不驗證另一個欄位。您可以使用 `exclude_if` 驗證規則來實現這一點。在下面的示例中，如果 `has_appointment` 欄位的值為 `false`，則不會驗證 `appointment_date` 和 `doctor_name` 欄位：

```
use Illuminate\Support\Facades\Validator;

$validator = Validator::make($data, [
     'has_appointment' => 'required|boolean',
     'appointment_date' => 'exclude_if:has_appointment,false|required|date',
     'doctor_name' => 'exclude_if:has_appointment,false|required|string',
]);
```

或者，您可以使用 `exclude_unless` 規則，除非另一個欄位具有給定值，否則不驗證給定欄位：

```
$validator = Validator::make($data, [
     'has_appointment' => 'required|boolean',
     'appointment_date' => 'exclude_unless:has_appointment,true|required|date',
     'doctor_name' => 'exclude_unless:has_appointment,true|required|string',
]);
```

16.8.1.2 僅在欄位存在時驗證

在某些情況下，您可能希望僅在驗證資料中存在該欄位時才對該欄位運行驗證檢查。要快速實現此操作，請將 `sometimes` 規則新增到您的規則列表中：

```
$v = Validator::make($data, [
     'email' => 'sometimes|required|email',
]);
```

在上面的示例中，如果`$data` 陣列中存在 `email` 欄位，則僅對其進行驗證。

> **注意** 如果您嘗試驗證始終應存在但可能為空的欄位，請查看有關可選欄位的說明。

16.8.1.3 複雜條件驗證

有時，您可能希望根據更複雜的條件邏輯新增驗證規則。例如，您可能只希望在另一個欄位的值大於 100 時要求給定欄位。或者，只有在存在另一個欄位時，兩個欄位才需要具有給定值。新增這些驗證規則不必是痛苦的。首先，使用永不改變的靜態規則建立一個 `Validator` 實例：

```
use Illuminate\Support\Facades\Validator;

$validator = Validator::make($request->all(), [
     'email' => 'required|email',
     'games' => 'required|numeric',
]);
```

假設我們的 Web 應用是給遊戲收藏家使用的。如果一個遊戲收藏家在我們的應用上註冊，並且他們擁有超過 100 個遊戲，我們想要讓他們解釋為什麼擁有這麼多遊戲。例如，也許他們經營著一家遊戲轉售店，或者他們只是喜歡收集遊戲。為了有條件地新增這個要求，我們可以在 `Validator` 實例上使用 `sometimes` 方法。

```
use Illuminate\Support\Fluent;

$validator->sometimes('reason', 'required|max:500', function (Fluent $input) {
     return $input->games >= 100;
});
```

傳遞給 `sometimes` 方法的第一個參數是我們有條件驗證的欄位的名稱。第二個參數是我們想要新增的規則列表。如果傳遞作為第三個參數的閉包返回 `true`，這些規則將被新增。使用此方法可以輕鬆建構複雜的條件驗證。您甚至可以同時為多個欄位新增條件驗證：

```
$validator->sometimes(['reason', 'cost'], 'required', function (Fluent $input) {
     return $input->games >= 100;
});
```

> **注意** 傳遞給您的閉包的 `$input` 參數將是 `Illuminate\Support\Fluent` 的一個實例，可用於訪問您正在驗證的輸入和檔案。

16.8.1.4 複雜條件陣列驗證

有時，您可能想要基於同一巢狀陣列中的另一個欄位驗證一個欄位，而您不知道其索引。在這種情況下，您可以允許您的閉包接收第二個參數，該參數將是正在驗證的當前個體陣列項：

```
$input = [
    'channels' => [
        [
            'type' => 'email',
            'address' => 'abigail@example.com',
        ],
        [
            'type' => 'url',
            'address' => 'https://example.com',
        ],
    ],
];

$validator->sometimes('channels.*.address', 'email', function (Fluent $input, Fluent
$item) {
    return $item->type === 'email';
});

$validator->sometimes('channels.*.address', 'url', function (Fluent $input, Fluent
$item) {
    return $item->type !== 'email';
});
```

像傳遞給閉包的 `$input` 參數一樣，當屬性資料是陣列時，`$item` 參數是 `Illuminate\Support\Fluent` 的實例；否則，它是一個字串。

16.9 驗證陣列

正如在 `array` 驗證規則文件 中討論的那樣，`array` 規則接受允許的陣列鍵列表。如果陣列中存在任何額外的鍵，則驗證將失敗：

```
use Illuminate\Support\Facades\Validator;

$input = [
    'user' => [
        'name' => 'Taylor Otwell',
        'username' => 'taylorotwell',
        'admin' => true,
    ],
];

Validator::make($input, [
    'user' => 'array:username,locale',
]);
```

通常情況下，您應該始終指定允許出現在陣列中的鍵。否則，驗證器的 `validate` 和 `validated` 方法將返回所有經過驗證的資料，包括陣列及其所有鍵，即使這些鍵沒有通過其他巢狀陣列驗證規則進行驗證。

16.9.1 驗證巢狀陣列輸入

驗證基於巢狀陣列的表單輸入欄位並不需要很痛苦。您可以使用 “點符號” 來驗證陣列中的屬性。例如，如果傳入的 HTTP 請求包含一個 `photos[profile]` 欄位，您可以像這樣驗證它：

```
use Illuminate\Support\Facades\Validator;

$validator = Validator::make($request->all(), [
```

```
    'photos.profile' => 'required|image',
]);
```

您還可以驗證陣列中的每個元素。例如，要驗證給定陣列輸入欄位中的每個電子郵件是否唯一，可以執行以下操作：

```
$validator = Validator::make($request->all(), [
    'person.*.email' => 'email|unique:users',
    'person.*.first_name' => 'required_with:person.*.last_name',
]);
```

同樣，您可以在語言檔案中指定自訂驗證消息時使用 * 字元，使得針對基於陣列的欄位使用單個驗證消息變得非常簡單：

```
'custom' => [
    'person.*.email' => [
        'unique' => 'Each person must have a unique email address',
    ]
],
```

16.9.1.1 訪問巢狀陣列資料

有時，當為屬性分配驗證規則時，您可能需要訪問給定巢狀陣列元素的值。您可以使用 `Rule::forEach` 方法來實現此目的。`forEach` 方法接受一個閉包，在驗證陣列屬性的每次迭代中呼叫該閉包，並接收屬性的值和顯式的完全展開的屬性名稱。閉包應該返回要分配給陣列元素的規則陣列：

```
use App\Rules\HasPermission;
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

$validator = Validator::make($request->all(), [
    'companies.*.id' => Rule::forEach(function (string|null $value, string $attribute) {
        return [
            Rule::exists(Company::class, 'id'),
            new HasPermission('manage-company', $value),
        ];
    }),
]);
```

16.9.2 錯誤消息索引和位置

在驗證陣列時，您可能希望在應用程式顯示的錯誤消息中引用失敗驗證的特定項的索引或位置。為了實現這一點，您可以在自訂驗證消息中包含 `:index`（從 `0` 開始）和 `:position`（從 `1` 開始）預留位置：

```
use Illuminate\Support\Facades\Validator;

$input = [
    'photos' => [
        [
            'name' => 'BeachVacation.jpg',
            'description' => '我的海灘假期照片！',
        ],
        [
            'name' => 'GrandCanyon.jpg',
            'description' => '',
        ],
    ],
];

Validator::validate($input, [
    'photos.*.description' => 'required',
], [
    'photos.*.description.required' => '請描述第 :position 張照片。',
]);
```

上述示例將驗證失敗，並且使用者會看到以下錯誤：“請描述第 2 張照片。”

16.10 驗證檔案

Laravel 提供了多種上傳檔案的驗證規則，如 `mimes`、`image`、`min` 和 `max`。雖然你可以在驗證檔案時單獨指定這些規則，但 Laravel 還是提供了一個流暢的檔案驗證規則生成器，你可能會覺得更方便：

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rules\File;

Validator::validate($input, [
    'attachment' => [
        'required',
        File::types(['mp3', 'wav'])
            ->min(1024)
            ->max(12 * 1024),
    ],
]);
```

如果你的程序允許使用者上傳圖片，那麼可以使用 `File` 規則的 `image` 構造方法來指定上傳的檔案應該是圖片。另外， `dimensions` 規則可用於限制圖片的尺寸：

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rules\File;

Validator::validate($input, [
    'photo' => [
        'required',
        File::image()
            ->min(1024)
            ->max(12 * 1024)
            ->dimensions(Rule::dimensions()->maxWidth(1000)->maxHeight(500)),
    ],
]);
```

> **技巧** 更多驗證圖片尺寸的資訊，請參見尺寸規則文件。

16.10.1.1 檔案類型

儘管在呼叫 `types` 方法時只需要指定擴展名，但該方法實際上是通過讀取檔案的內容並猜測其 MIME 類型來驗證檔案的 MIME 類型的。MIME 類型及其相應擴展的完整列表可以在以下連結中找到：

https://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types

16.11 驗證密碼

為確保密碼具有足夠的複雜性，你可以使用 Laravel 的 `password` 規則對象：

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rules\Password;

$validator = Validator::make($request->all(), [
    'password' => ['required', 'confirmed', Password::min(8)],
]);
```

`Password` 規則對象允許你輕鬆自訂應用程式的密碼複雜性要求，例如指定密碼至少需要一個字母、數字、符號或混合大小寫的字元：

```
// 至少需要 8 個字元...
```

```
    Password::min(8)

    // 至少需要一個字母...
    Password::min(8)->letters()

    // 至少需要一個大寫字母和一個小寫字母...
    Password::min(8)->mixedCase()

    // 至少需要一個數字...
    Password::min(8)->numbers()

    // 至少需要一個符號...
    Password::min(8)->symbols()
```

此外，你可以使用 uncompromised 方法確保密碼沒有在公共密碼資料洩露事件中被洩露：

```
    Password::min(8)->uncompromised()
```

在內部，Password 規則對象使用 k-Anonymity 模型來確定密碼是否已通過 haveibeenpwned.com 服務而不犧牲使用者的隱私或安全。

默認情況下，如果密碼在資料洩露中至少出現一次，則會被視為已洩露。你可以使用 uncompromised 方法的第一個參數自訂此閾值

```
    // Ensure the password appears less than 3 times in the same data leak...
    Password::min(8)->uncompromised(3);
```

當然，你可以將上面示例中的所有方法連結起來：

```
    Password::min(8)
        ->letters()
        ->mixedCase()
        ->numbers()
        ->symbols()
        ->uncompromised()
```

16.11.1.1 定義默認密碼規則

你可能會發現在應用程式的單個位置指定密碼的默認驗證規則很方便。你可以使用接受閉包的 Password::defaults 方法輕鬆完成此操作。給 defaults 方法的閉包應該返回密碼規則的默認組態。通常，應該在應用程式服務提供者之一的 boot 方法中呼叫 defaults 規則：

```
use Illuminate\Validation\Rules\Password;

/**
 * 引導任何應用程式服務
 */
public function boot(): void
{
    Password::defaults(function () {
        $rule = Password::min(8);

        return $this->app->isProduction()
                    ? $rule->mixedCase()->uncompromised()
                    : $rule;
    });
}
```

然後，當你想將默認規則應用於正在驗證的特定密碼時，你可以呼叫不帶參數的 defaults 方法：

```
'password' => ['required', Password::defaults()],
```

有時，你可能希望將其他驗證規則附加到默認密碼驗證規則。 你可以使用 rules 方法來完成此操作：

```
use App\Rules\ZxcvbnRule;

Password::defaults(function () {
```

```
    $rule = Password::min(8)->rules([new ZxcvbnRule]);

    // ...
});
```

16.12 自訂驗證規則

16.12.1 使用規則對象

Laravel 提供了各種有用的驗證規則；但是，你可能希望指定一些你自己的。 註冊自訂驗證規則的一種方法是使用規則對象。 要生成新的規則對象，你可以使用 `make:rule` Artisan 命令。 讓我們使用這個命令生成一個規則來驗證字串是否為大寫。 Laravel 會將新規則放在 `app/Rules` 目錄中。 如果這個目錄不存在，Laravel 會在你執行 Artisan 命令建立規則時建立它：

```
php artisan make:rule Uppercase
```

一旦規則被建立，我們就可以定義其行為。一個規則對象包含一個單一的方法：`validate`。該方法接收屬性名、其值和一個回呼函數，如果驗證失敗應該呼叫該回呼函數並傳入驗證錯誤消息：

```
<?php

namespace App\Rules;

use Closure;
use Illuminate\Contracts\Validation\ValidationRule;

class Uppercase implements ValidationRule
{
    /**
     * Run the validation rule.
     */
    public function validate(string $attribute, mixed $value, Closure $fail): void
    {
        if (strtoupper($value) !== $value) {
            $fail('The :attribute must be uppercase.');
        }
    }
}
```

一旦定義了規則，您可以通過將規則對象的實例與其他驗證規則一起傳遞來將其附加到驗證器：

```
use App\Rules\Uppercase;

$request->validate([
    'name' => ['required', 'string', new Uppercase],
]);
```

16.12.1.1 驗證消息

您可以不提供 `$fail` 閉包的字面錯誤消息，而是提供一個翻譯字串鍵，並指示 Laravel 翻譯錯誤消息：

```
if (strtoupper($value) !== $value) {
    $fail('validation.uppercase')->translate();
}
```

如有必要，您可以通過第一個和第二個參數分別提供預留位置替換和首選語言來呼叫 `translate` 方法：

```
$fail('validation.location')->translate([
    'value' => $this->value,
], 'fr')
```

16.12.1.2 訪問額外資料

如果您的自訂驗證規則類需要訪問正在驗證的所有其他資料，則規則類可以實現 `Illuminate\Contracts\Validation\DataAwareRule` 介面。此介面要求您的類定義一個 `setData` 方法。Laravel 會自動呼叫此方法（在驗證繼續之前）並傳入所有正在驗證的資料：

```
<?php

namespace App\Rules;

use Illuminate\Contracts\Validation\DataAwareRule;
use Illuminate\Contracts\Validation\ValidationRule;

class Uppercase implements DataAwareRule, ValidationRule
{
    /**
     * 正在驗證的所有資料。
     *
     * @var array<string, mixed>
     */
    protected $data = [];

    // ...

    /**
     * 設定正在驗證的資料。
     *
     * @param  array<string, mixed>  $data
     */
    public function setData(array $data): static
    {
        $this->data = $data;

        return $this;
    }
}
```

或者，如果您的驗證規則需要訪問執行驗證的驗證器實例，則可以實現 `ValidatorAwareRule` 介面：

```
<?php

namespace App\Rules;

use Illuminate\Contracts\Validation\ValidationRule;
use Illuminate\Contracts\Validation\ValidatorAwareRule;
use Illuminate\Validation\Validator;

class Uppercase implements ValidationRule, ValidatorAwareRule
{
    /**
     * 驗證器實例.
     *
     * @var \Illuminate\Validation\Validator
     */
    protected $validator;

    // ...

    /**
     * 設定當前驗證器.
     */
    public function setValidator(Validator $validator): static
    {
        $this->validator = $validator;

        return $this;
```

```
    }
}
```

16.12.2 使用閉包涵數

如果您只需要在應用程式中一次使用自訂規則的功能，可以使用閉包涵數而不是規則對象。閉包涵數接收屬性名稱、屬性值和 $fail 回呼函數，如果驗證失敗，應該呼叫該函數：

```
use Illuminate\Support\Facades\Validator;

$validator = Validator::make($request->all(), [
    'title' => [
        'required',
        'max:255',
        function (string $attribute, mixed $value, Closure $fail) {
            if ($value === 'foo') {
                $fail("The {$attribute} is invalid.");
            }
        },
    ],
]);
```

16.12.3 隱式規則

默認情況下，當要驗證的屬性不存在或包含空字串時，正常的驗證規則，包括自訂規則，都不會執行。例如，unique 規則不會針對空字串運行：

```
use Illuminate\Support\Facades\Validator;

$rules = ['name' => 'unique:users,name'];

$input = ['name' => ''];

Validator::make($input, $rules)->passes(); // true
```

為了使自訂規則在屬性為空時也運行，規則必須暗示該屬性是必需的。您可以使用 make:rule Artisan 命令的 –implicit 選項快速生成新的隱式規則對象：

```
php artisan make:rule Uppercase --implicit
```

> **警告**
> 隱式規則僅 暗示 該屬性是必需的。實際上，缺少或空屬性是否無效取決於您。

17 錯誤處理

17.1 介紹

當你開始一個新的 Laravel 項目時，它已經為你組態了錯誤和異常處理。`App\Exceptions\Handler` 類用於記錄應用程式觸發的所有異常，然後將其呈現回使用者。我們將在本文中深入討論這個類。

17.2 組態

你的 `config/app.php` 組態檔案中的 `debug` 選項決定了對於一個錯誤實際上將顯示多少資訊給使用者。默認情況下，該選項的設定將遵照儲存在 `.env` 檔案中的 `APP_DEBUG` 環境變數的值。

對於本地開發，你應該將 `APP_DEBUG` 環境變數的值設定為 `true`。 **在生產環境中，該值應始終為 `false`。如果在生產中將該值設定為 `true`，則可能會將敏感組態值暴露給應用程式的終端使用者。**

17.3 異常處理

17.3.1 異常報告

所有異常都是由 `App\Exceptions\Handler` 類處理。此類包含一個 `register` 方法，可以在其中註冊自訂異常報告程序和渲染器回呼。我們將詳細研究每個概念。異常報告用於記錄異常或將其傳送到如 Flare、Bugsnag 或 Sentry 等外部服務。默認情況下，將根據你的日誌組態來記錄異常。不過，你可以用任何自己喜歡的方式來記錄異常。

例如，如果您需要以不同的方式報告不同類型的異常，您可以使用 reportable 方法註冊一個閉包，當需要報告給定的異常的時候便會執行它。 Laravel 將通過檢查閉包的類型提示來判斷閉包報告的異常類型：

```
use App\Exceptions\InvalidOrderException;

/**
 * 為應用程式註冊異常處理回呼
 */
public function register(): void
{
    $this->reportable(function (InvalidOrderException $e) {
        // ...
    });
}
```

當您使用 reportable 方法註冊一個自訂異常報告回呼時， Laravel 依然會使用默認的日誌組態記錄下應用異常。如果您想要在默認的日誌堆疊中停止這個行為，您可以在定義報告回呼時使用 stop 方法或者從回呼函數中返回 false：

```
$this->reportable(function (InvalidOrderException $e) {
    // ...
})->stop();

$this->reportable(function (InvalidOrderException $e) {
    return false;
});
```

技巧
要為給定的異常自訂異常報告，您可以使用 可報告異常.

17.3.1.1 全域日誌上下文

在可用的情況下， Laravel 會自動將當前使用者的編號作為資料新增到每一條異常日誌資訊中。您可以通過重寫 App 類中的 context 方法來定義您自己的全域上下文資料（環境變數）。此後，每一條異常日誌資訊都將包含這個資訊：

```
/**
 * 獲取默認日誌的上下文變數。
 *
 * @return array<string, mixed>
 */
protected function context(): array
{
    return array_merge(parent::context(), [
        'foo' => 'bar',
    ]);
}
```

17.3.1.2 異常日誌上下文

儘管將上下文新增到每個日誌消息中可能很有用，但有時特定的異常可能具有您想要包含在日誌中的唯一上下文。通過在應用程式的自訂異常中定義 `context` 方法，您可以指定與該異常相關的任何資料，應將其新增到異常的日誌條目中：

```
<?php

namespace App\Exceptions;

use Exception;

class InvalidOrderException extends Exception
{
    // ...

    /**
     * 獲取異常上下文資訊
     *
     * @return array<string, mixed>
     */
    public function context(): array
    {
        return ['order_id' => $this->orderId];
    }
}
```

17.3.1.3 `report` 助手

有時，您可能需要報告異常，但繼續處理當前請求。`report` 助手函數允許您通過異常處理程序快速報告異常，而無需向使用者呈現錯誤頁面：

```
public function isValid(string $value): bool
{
    try {
        // Validate the value...
    } catch (Throwable $e) {
        report($e);

        return false;
    }
```

```
}
```

17.3.2 異常日誌等級

當消息被寫入應用程式的日誌時，消息將以指定的日誌等級寫入，該等級指示正在記錄的消息的嚴重性或重要性。

如上所述，即使使用 `reportable` 方法註冊自訂異常報告回呼，Laravel 仍將使用應用程式的默認日誌記錄組態記錄異常；但是，由於日誌等級有時會影響消息記錄的通道，因此您可能希望組態某些異常記錄的日誌等級。

為了實現這個目標，您可以在應用程式的異常處理程序的`$levels` 屬性中定義一個異常類型陣列以及它們關聯的日誌等級：

```
use PDOException;
use Psr\Log\LogLevel;

/**
 * 包含其對應自訂日誌等級的異常類型列表。
 *
 * @var array<class-string<\Throwable>, \Psr\Log\LogLevel::*>
 */
protected $levels = [
    PDOException::class => LogLevel::CRITICAL,
];
```

17.3.3 按類型忽略異常

在建構應用程式時，您可能希望忽略某些類型的異常並永遠不報告它們。應用程式的異常處理程序包含一個 `$dontReport` 屬性，該屬性初始化為空陣列。您新增到此屬性的任何類都將不會被報告；但是它們仍然可能具有自訂渲染邏輯：

```
use App\Exceptions\InvalidOrderException;

/**
 * 不會被報告的異常類型列表。
 *
 * @var array<int, class-string<\Throwable>>
 */
protected $dontReport = [
    InvalidOrderException::class,
];
```

在內部，Laravel 已經為您忽略了一些類型的錯誤，例如由 404 HTTP 錯誤或由無效 CSRF 令牌生成的 419 HTTP 響應引起的異常。如果您想指示 Laravel 停止忽略給定類型的異常，您可以在異常處理程序的 `register` 方法中呼叫 `stopIgnoring` 方法：

```
use Symfony\Component\HttpKernel\Exception\HttpException;

/**
 * 為應用程式註冊異常處理回呼函數。
 */
public function register(): void
{
    $this->stopIgnoring(HttpException::class);

    // ...
}
```

17.3.4 渲染異常

默認情況下，Laravel 異常處理程序會將異常轉換為 HTTP 響應。但是，您可以自由地為給定類型的異常註冊自訂渲染閉包。您可以通過在異常處理程序中呼叫 `renderable` 方法來實現這一點。

傳遞給 `renderable` 方法的閉包應該返回一個 `Illuminate\Http\Response` 實例，該實例可以通過 `response` 助手生成。 Laravel 將通過檢查閉包的類型提示來推斷閉包呈現的異常類型：

```
use App\Exceptions\InvalidOrderException;
use Illuminate\Http\Request;

/**
 * Register the exception handling callbacks for the application.
 */
public function register(): void
{
    $this->renderable(function (InvalidOrderException $e, Request $request) {
        return response()->view('errors.invalid-order', [], 500);
    });
}
```

您還可以使用 `renderable` 方法來覆蓋內建的 Laravel 或 Symfony 異常的呈現行為，例如 `NotFoundHttpException`。如果傳遞給 `renderable` 方法的閉包沒有返回值，則將使用 Laravel 的默認異常呈現：

```
use Illuminate\Http\Request;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

/**
 * Register the exception handling callbacks for the application.
 */
public function register(): void
{
    $this->renderable(function (NotFoundHttpException $e, Request $request) {
        if ($request->is('api/*')) {
            return response()->json([
                'message' => 'Record not found.'
            ], 404);
        }
    });
}
```

17.3.5 Reportable & Renderable 異常

您可以直接在自訂異常類中定義 `report` 和 `render` 方法，而不是在異常處理程序的 `register` 方法中定義自訂報告和呈現行為。當存在這些方法時，框架將自動呼叫它們：

```
<?php

namespace App\Exceptions;

use Exception;
use Illuminate\Http\Request;
use Illuminate\Http\Response;

class InvalidOrderException extends Exception
{
    /**
     * Report the exception.
     */
    public function report(): void
    {
        // ...
```

```
    }

    /**
     * Render the exception into an HTTP response.
     */
    public function render(Request $request): Response
    {
        return response(/* ... */);
    }
}
```

如果您的異常擴展了已經可呈現的異常，例如內建的 Laravel 或 Symfony 異常，則可以從異常的 `render` 方法中返回 `false`，以呈現異常的默認 HTTP 響應：

```
/**
 * Render the exception into an HTTP response.
 */
public function render(Request $request): Response|bool
{
    if (/** Determine if the exception needs custom rendering */) {

        return response(/* ... */);
    }

    return false;
}
```

如果你的異常包含了只在特定條件下才需要使用的自訂報告邏輯，那麼你可能需要指示 Laravel 有時使用默認的異常處理組態來報告異常。為了實現這一點，你可以從異常的 `report` 方法中返回 `false`：

```
/**
 * Report the exception.
 */
public function report(): bool
{
    if (/** 確定異常是否需要自訂報告 */) {

        // ...

        return true;
    }

    return false;
}
```

> **注意** 你可以在 `report` 方法中類型提示任何所需的依賴項，它們將自動被 Laravel 的服務容器注入該方法中。

17.4 HTTP 異常

有些異常描述了伺服器返回的 HTTP 錯誤程式碼。例如，這可能是一個 “頁面未找到” 錯誤（404），一個 “未經授權錯誤”（401）或甚至是一個由開發者生成的 500 錯誤。為了從應用程式的任何地方生成這樣的響應，你可以使用 `abort` 幫助函數：

```
abort(404);
```

17.4.1 自訂 HTTP 錯誤頁面

Laravel 使得為各種 HTTP 狀態碼顯示自訂錯誤頁面變得很容易。例如，如果你想自訂 404 HTTP 狀態碼的錯誤頁面，請建立一個 `resources/views/errors/404.blade.php` 檢視範本。這個檢視將會被渲染在應用程式生成的所有 404 錯誤上。這個目錄中的檢視應該被命名為它們對應的 HTTP 狀態碼。`abort` 函數引發的

`Symfony\Component\HttpKernel\Exception\HttpException` 實例將會以 `$exception` 變數的形式傳遞給檢視：

```
<h2>{{ $exception->getMessage() }}</h2>
```

你可以使用 `vendor:publish` Artisan 命令發佈 Laravel 的默認錯誤頁面範本。一旦範本被發佈，你可以根據自己的喜好進行自訂：

```
php artisan vendor:publish --tag=laravel-errors
```

17.4.1.1 回退 HTTP 錯誤頁面

你也可以為給定系列的 HTTP 狀態碼定義一個“回退”錯誤頁面。如果沒有針對發生的具體 HTTP 狀態碼相應的頁面，就會呈現此頁面。為了實現這一點，在你應用程式的 `resources/views/errors` 目錄中定義一個 `4xx.blade.php` 範本和一個 `5xx.blade.php` 範本。

18 日誌

18.1 介紹

為了幫助您更多地瞭解應用程式中發生的事情，Laravel 提供了強大的日誌記錄服務，允許您將日誌記錄到檔案、系統錯誤日誌，甚至記錄到 Slack 以通知您的整個團隊。

Laravel 日誌基於「通道」。 每個通道代表一種寫入日誌資訊的特定方式。 例如，`single` 通道是將日誌寫入到單個記錄檔中。而 `slack` 通道是將日誌傳送到 Slack 上。 基於它們的重要程度，日誌可以被寫入到多個通道中去。

在底層，Laravel 利用 Monolog 庫，它為各種強大的日誌處理程序提供了支援。 Laravel 使組態這些處理程序變得輕而易舉，允許您混合和匹配它們，以自訂應用程式的方式完成日誌處理。

18.2 組態

所有應用程式的日誌行為組態選項都位於 `config/logging.php` 組態檔案中。 該檔案允許您組態應用程式的日誌通道，因此請務必查看每個可用通道及其選項。 我們將在下面回顧一些常見的選項。

默認情況下，Laravel 在記錄日誌消息時使用 `stack` 頻道。`stack` 頻道用於將多個日誌頻道聚合到一個頻道中。有關建構堆疊的更多資訊，請查看下面的文件。

18.2.1.1 組態頻道名稱

默認情況下，Monolog 使用與當前環境相匹配的“頻道名稱”（例如 `production` 或 `local`）進行實例化。要更改此值，請向頻道的組態中新增一個 `name` 選項：

```
'stack' => [
    'driver' => 'stack',
    'name' => 'channel-name',
    'channels' => ['single', 'slack'],
],
```

18.2.2 可用頻道驅動程式

每個日誌頻道都由一個“驅動程式”驅動。驅動程式確定實際記錄日誌消息的方式和位置。以下日誌頻道驅動程式在每個 Laravel 應用程式中都可用。大多數這些驅動程式的條目已經在應用程式的 `config/logging.php` 組態檔案中存在，因此請務必查看此檔案以熟悉其內容：

名稱	描述
`custom`	呼叫指定工廠建立頻道的驅動程式
`daily`	基於 `RotatingFileHandler` 的 Monolog 驅動程式，每天輪換一次記錄檔
`errorlog`	基於 `ErrorLogHandler` 的 Monolog 驅動程式
`monolog`	可使用任何支援的 Monolog 處理程序的 Monolog 工廠驅動程式
`null`	丟棄所有日誌消息的驅動程式
`papertrail`	基於 `SyslogUdpHandler` 的 Monolog 驅動程式
`single`	單個檔案或路徑為基礎的記錄器頻道（`StreamHandler`）
`slack`	基於 `SlackWebhookHandler` 的 Monolog 驅動程式
`stack`	包裝器，用於方便地建立“多通道”頻道

名稱	描述
syslog	基於 SyslogHandler 的 Monolog 驅動程式

注意 查看 高級頻道自訂 文件，瞭解有關 monolog 和 custom 驅動程式的更多資訊。

18.2.3 頻道前提條件

18.2.3.1 組態單一和日誌頻道

在處理消息時，single 和 daily 頻道有三個可選組態選項：bubble，permission 和 locking。

名稱	描述	預設值
bubble	表示是否在處理後將消息傳遞到其他頻道	true
locking	在寫入記錄檔之前嘗試鎖定記錄檔	false
permission	記錄檔的權限	0644

另外，可以通過 days 選項組態 daily 頻道的保留策略：

名稱	描述	預設值
days	保留每日記錄檔的天數	7

18.2.3.2 組態 Papertrail 頻道

papertrail 頻道需要 host 和 port 組態選項。您可以從 Papertrail 獲取這些值。

18.2.3.3 組態 Slack 頻道

slack 頻道需要一個 url 組態選項。此 URL 應該與您為 Slack 團隊組態的 incoming webhook 的 URL 匹配。

默認情況下，Slack 僅會接收 critical 等級及以上的日誌；但是，您可以通過修改 config/logging.php 組態檔案中您的 Slack 日誌頻道組態陣列中的 level 組態選項來調整此設定。

18.2.4 記錄棄用警告

PHP、Laravel 和其他庫通常會通知其使用者，一些功能已被棄用，將在未來版本中刪除。如果您想記錄這些棄用警告，可以在應用程式的 config/logging.php 組態檔案中指定您首選的 deprecations 日誌頻道：

```
'deprecations' => env('LOG_DEPRECATIONS_CHANNEL', 'null'),

'channels' => [
    ...
]
```

或者，您可以定義一個名為 deprecations 的日誌通道。如果存在此名稱的日誌通道，則始終將其用於記錄棄用：

```
'channels' => [
    'deprecations' => [
        'driver' => 'single',
        'path' => storage_path('logs/php-deprecation-warnings.log'),
    ],
],
```

18.3 建構日誌堆疊

如前所述，stack 驅動程式允許您將多個通道組合成一個方便的日誌通道。為了說明如何使用日誌堆疊，讓我們看一個您可能在生產應用程式中看到的示例組態：

```
'channels' => [
    'stack' => [
        'driver' => 'stack',
        'channels' => ['syslog', 'slack'],
    ],

    'syslog' => [
        'driver' => 'syslog',
        'level' => 'debug',
    ],

    'slack' => [
        'driver' => 'slack',
        'url' => env('LOG_SLACK_WEBHOOK_URL'),
        'username' => 'Laravel Log',
        'emoji' => ':boom:',
        'level' => 'critical',
    ],
],
```

讓我們分解一下這個組態。首先，請注意我們的 stack 通道通過其 channels 選項聚合了兩個其他通道：syslog 和 slack。因此，在記錄消息時，這兩個通道都有機會記錄消息。但是，正如我們將在下面看到的那樣，這些通道是否實際記錄消息可能取決於消息的嚴重程度/“等級”。

18.3.1.1 日誌等級

請注意上面示例中 syslog 和 slack 通道組態中存在的 level 組態選項。此選項確定必須記錄消息的最小“等級”。Laravel 的日誌服務採用 Monolog，提供 RFC 5424 規範中定義的所有日誌等級。按嚴重程度遞減的順序，這些日誌等級是：**emergency**，**alert**，**critical**，**error**，**warning**，**notice**，**info** 和 **debug**。

在我們的組態中，如果我們使用 debug 方法記錄消息：

```
Log::debug('An informational message.');
```

根據我們的組態，syslog 管道將把消息寫入系統日誌；但由於錯誤消息不是 critical 或以上等級，它不會被傳送到 Slack。然而，如果我們記錄一個 emergency 等級的消息，則會傳送到系統日誌和 Slack，因為 emergency 等級高於我們兩個管道的最小等級閾值：

```
Log::emergency('The system is down!');
```

18.4 寫入日誌消息

您可以使用 Log facade 向日誌寫入資訊。正如之前提到的，日誌記錄器提供了 RFC 5424 規範 中定義的八個日誌等級：**emergency**、**alert**、**critical**、**error**、**warning**、**notice**、**info** 和 **debug**：

```
use Illuminate\Support\Facades\Log;

Log::emergency($message);
Log::alert($message);
Log::critical($message);
Log::error($message);
Log::warning($message);
Log::notice($message);
Log::info($message);
Log::debug($message);
```

您可以呼叫其中任何一個方法來記錄相應等級的消息。默認情況下，該消息將根據您的 `logging` 組態檔案組態的默認日誌管道進行寫入：

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\User;
use Illuminate\Support\Facades\Log;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     */
    public function show(string $id): View
    {
        Log::info('Showing the user profile for user: '.$id);

        return view('user.profile', [
            'user' => User::findOrFail($id)
        ]);
    }
}
```

18.4.1 上下文資訊

可以向日誌方法傳遞一組上下文資料。這些上下文資料將與日誌消息一起格式化和顯示：

```
use Illuminate\Support\Facades\Log;

Log::info('User failed to login.', ['id' => $user->id]);
```

偶爾，您可能希望指定一些上下文資訊，這些資訊應包含在特定頻道中所有隨後的日誌條目中。例如，您可能希望記錄與應用程式的每個傳入請求相關聯的請求 ID。為了實現這一目的，您可以呼叫 `Log` 門面的 `withContext` 方法：

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Log;
use Illuminate\Support\Str;
use Symfony\Component\HttpFoundation\Response;

class AssignRequestId
{
    /**
     * Handle an incoming request.
     *
     * @param  \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\
Response)  $next
     */
    public function handle(Request $request, Closure $next): Response
    {
        $requestId = (string) Str::uuid();

        Log::withContext([
            'request-id' => $requestId
        ]);

        return $next($request)->header('Request-Id', $requestId);
```

```
    }
}
```

如果要在_所有_日誌頻道之間共享上下文資訊，則可以呼叫 `Log::shareContext()` 方法。此方法將向所有已建立的頻道提供上下文資訊，以及隨後建立的任何頻道。通常，`shareContext` 方法應從應用程式服務提供程序的 `boot` 方法中呼叫：

```
use Illuminate\Support\Facades\Log;
use Illuminate\Support\Str;

class AppServiceProvider
{
    /**
     * 啟動任何應用程式服務。
     */
    public function boot(): void
    {
        Log::shareContext([
            'invocation-id' => (string) Str::uuid(),
        ]);
    }
}
```

18.4.2 寫入特定頻道

有時，您可能希望將消息記錄到應用程式默認頻道以外的頻道。您可以使用 `Log` 門面上的 `channel` 方法來檢索並記錄組態檔案中定義的任何頻道：

```
use Illuminate\Support\Facades\Log;

Log::channel('slack')->info('Something happened!');
```

如果你想建立一個由多個通道組成的按需記錄堆疊，可以使用 `stack` 方法：

```
Log::stack(['single', 'slack'])->info('Something happened!');
```

18.4.2.1 按需通道

還可以建立一個按需通道，方法是在執行階段提供組態而無需將該組態包含在應用程式的 `logging` 組態檔案中。為此，可以將組態陣列傳遞給 `Log` 門面的 `build` 方法：

```
use Illuminate\Support\Facades\Log;

Log::build([
  'driver' => 'single',
  'path' => storage_path('logs/custom.log'),
])->info('Something happened!');
```

您可能還希望在按需記錄堆疊中包含一個按需通道。可以通過將按需通道實例包含在傳遞給 `stack` 方法的陣列中來實現：

```
use Illuminate\Support\Facades\Log;

$channel = Log::build([
  'driver' => 'single',
  'path' => storage_path('logs/custom.log'),
]);

Log::stack(['slack', $channel])->info('Something happened!');
```

18.5 Monolog 通道定製

18.5.1 為通道定製 Monolog

有時，您可能需要完全控制 Monolog 如何組態現有通道。例如，您可能希望為 Laravel 內建的 `single` 通道組態自訂的 Monolog `FormatterInterface` 實現。

要開始，請在通道組態中定義 `tap` 陣列。`tap` 陣列應包含一系列類，這些類在建立 Monolog 實例後應有機會自訂（或“tap”）它。沒有這些類應放置在何處的慣例位置，因此您可以在應用程式中建立一個目錄以包含這些類：

```
'single' => [
    'driver' => 'single',
    'tap' => [App\Logging\CustomizeFormatter::class],
    'path' => storage_path('logs/laravel.log'),
    'level' => 'debug',
],
```

一旦你在通道上組態了 `tap` 選項，你就可以定義一個類來自訂你的 Monolog 實例。這個類只需要一個方法：`__invoke`，它接收一個 `Illuminate\Log\Logger` 實例。`Illuminate\Log\Logger` 實例代理所有方法呼叫到底層的 Monolog 實例：

```
<?php

namespace App\Logging;

use Illuminate\Log\Logger;
use Monolog\Formatter\LineFormatter;

class CustomizeFormatter
{
    /**
     * 自訂給定的日誌記錄器實例。
     */
    public function __invoke(Logger $logger): void
    {
        foreach ($logger->getHandlers() as $handler) {
            $handler->setFormatter(new LineFormatter(
                '[%datetime%] %channel%.%level_name%: %message% %context% %extra%'
            ));
        }
    }
}
```

注意 所有的 “tap” 類都由 服務容器 解析，因此它們所需的任何建構函式依賴關係都將自動注入。

18.5.2 建立 Monolog 處理程序通道

Monolog 有多種 可用的處理程序，而 Laravel 並沒有為每個處理程序內建通道。在某些情況下，你可能希望建立一個自訂通道，它僅是一個特定的 Monolog 處理程序實例，該處理程序沒有相應的 Laravel 日誌驅動程式。這些通道可以使用 `monolog` 驅動程式輕鬆建立。

使用 `monolog` 驅動程式時，`handler` 組態選項用於指定將實例化哪個處理程序。可選地，可以使用 `with` 組態選項指定處理程序需要的任何建構函式參數：

```
'logentries' => [
    'driver'  => 'monolog',
    'handler' => Monolog\Handler\SyslogUdpHandler::class,
    'with' => [
```

```
            'host' => 'my.logentries.internal.datahubhost.company.com',
            'port' => '10000',
        ],
],
```

18.5.2.1 Monolog 格式化程序

使用 `monolog` 驅動程式時，Monolog `LineFormatter` 將用作默認格式化程序。但是，你可以使用 `formatter` 和 `formatter_with` 組態選項自訂傳遞給處理程序的格式化程序類型：

```
'browser' => [
    'driver' => 'monolog',
    'handler' => Monolog\Handler\BrowserConsoleHandler::class,
    'formatter' => Monolog\Formatter\HtmlFormatter::class,
    'formatter_with' => [
        'dateFormat' => 'Y-m-d',
    ],
],
```

如果你使用的是能夠提供自己的格式化程序的 Monolog 處理程序，你可以將 `formatter` 組態選項的值設定為 `default`：

```
'newrelic' => [
    'driver' => 'monolog',
    'handler' => Monolog\Handler\NewRelicHandler::class,
    'formatter' => 'default',
],
```

18.5.2.2 Monolog 處理器

Monolog 也可以在記錄消息之前對其進行處理。你可以建立你自己的處理器或使用 Monolog 提供的現有處理器。

如果你想為 `monolog` 驅動定製處理器，請在通道的組態中加入 `processors` 組態值。

```
 'memory' => [
     'driver' => 'monolog',
     'handler' => Monolog\Handler\StreamHandler::class,
     'with' => [
         'stream' => 'php://stderr',
     ],
     'processors' => [
         // Simple syntax...
         Monolog\Processor\MemoryUsageProcessor::class,

         // With options...
         [
            'processor' => Monolog\Processor\PsrLogMessageProcessor::class,
            'with' => ['removeUsedContextFields' => true],
        ],
     ],
 ],
```

18.5.3 通過工廠建立通道

如果你想定義一個完全自訂的通道，你可以在其中完全控制 Monolog 的實例化和組態，你可以在 `config/logging.php` 組態檔案中指定 `custom` 驅動程式類型。你的組態應該包括一個 `via` 選項，其中包含將被呼叫以建立 Monolog 實例的工廠類的名稱：

```
'channels' => [
    'example-custom-channel' => [
        'driver' => 'custom',
```

```
        'via' => App\Logging\CreateCustomLogger::class,
    ],
],
```

一旦你組態了 `custom` 驅動程式通道，你就可以定義將建立你的 Monolog 實例的類。這個類只需要一個 __invoke 方法，它應該返回 Monolog 記錄器實例。 該方法將接收通道組態陣列作為其唯一參數：

```
<?php

namespace App\Logging;

use Monolog\Logger;

class CreateCustomLogger
{
    /**
     * 建立一個自訂 Monolog 實例。
     */
    public function __invoke(array $config): Logger
    {
        return new Logger(/* ... */);
    }
}
```

19 Artisan 命令列

19.1 介紹

Artisan 是 Laravel 中自帶的命令列介面。Artisan 以 `artisan` 指令碼的方式存在於應用的根目錄中，提供了許多有用的命令，幫助開發者建立應用。使用 `list` 命令可以查看所有可用的 Artisan 命令：

```
php artisan list
```

每個命令都與 “help” 幫助介面，它能顯示和描述該命令可用的參數和選項。要查看幫助介面，請在命令前加上 `help` 即可：

```
php artisan help migrate
```

19.1.1.1 Laravel Sail

如果你使用 Laravel Sail 作為本地開發環境，記得使用 `sail` 命令列來呼叫 Artisan 命令。Sail 會在應用的 Docker 容器中執行 Artisan 命令：

```
./vendor/bin/sail artisan list
```

19.1.2 Tinker (REPL)

Laravel Tinker 是為 Laravel 提供的強大的 REPL（互動式直譯器），由 PsySH(https://github.com/bobthecow/psysh) 驅動支援。

19.1.2.1 安裝

所有的 Laravel 應用默認都自帶 Tinker。不過，如果你此前刪除了它，你可以使用 Composer 安裝：

```
composer require laravel/tinker
```

> **注意**
> 需要能與 Laravel 互動的圖形使用者介面嗎？試試 Tinkerwell!

19.1.2.2 使用

Tinker 允許你在命令列中和整個 Laravel 應用互動，包括 Eloquent 模型、佇列、事件等等。要進入 Tinker 環境，只需運行 `tinker` Artisan 命令：

```
php artisan tinker
```

你可以使用 `vendor:publish` 命令發佈 Tinker 的組態檔案：

```
php artisan vendor:publish --provider="Laravel\Tinker\TinkerServiceProvider"
```

> **警告**
> `dispatch` 輔助函數及 `Dispatchable` 類中 `dispatch` 方法依賴於垃圾回收將任務放置到佇列中。因此，使用 tinker 時，請使用 `Bus::dispath` 或 `Queue::push` 來分發任務。

19.1.2.3 命令白名單

Tinker 使用白名單來確定哪些 Artisan 命令可以在其 Shell 中運行。默認情況下，你可以運行 `clear-compiled`、`down`、`env`、`inspire`、`migrate`、`optimize` 和 `up` 命令。如果你想允許更多命令，你可以將

它們新增到 tinker.php 組態檔案的 commands 陣列中：

```
'commands' => [
    // App\Console\Commands\ExampleCommand::class,
],
```

19.1.2.4 別名黑名單

一般而言，Tinker 會在你引入類時自動為其新增別名。不過，你可能不希望為某些類新增別名。你可以在 tinker.php 組態檔案的 dont_alias 陣列中列舉這些類來完成此操作：

```
'dont_alias' => [
    App\Models\User::class,
],
```

19.2 編寫命令

除了 Artisan 提供的命令之外，你可以建立自訂命令。一般而言，命令保存在 app/Console/Commands 目錄；不過，你可以自由選擇命令的儲存位置，只要它能夠被 Composer 載入即可。

19.2.1 生成命令

要建立新命令，可以使用 make:command Artisan 命令。該命令會在 app/Console/Commands 目錄下建立一個新的命令類。如果該目錄不存在，也無需擔心 - 它會在第一次運行 make:command Artisan 命令的時候自動建立：

```
php artisan make:command SendEmails
```

19.2.2 命令結構

生成命令後，應該為該類的 signature 和 description 屬性設定設當的值。當在 list 螢幕上顯示命令時，將使用這些屬性。signature 屬性也會讓你定義命令輸入預期值。handle 放回會在命令執行時被呼叫。你可以在該方法中編寫命令邏輯。

讓我們看一個示例命令。請注意，我們能夠通過命令的 handle 方法引入我們需要的任何依賴項。Laravel 服務容器 將自動注入此方法簽名中帶有類型提示的所有依賴項：

```
<?php

namespace App\Console\Commands;

use App\Models\User;
use App\Support\DripEmailer;
use Illuminate\Console\Command;

class SendEmails extends Command
{
    /**
     * 控制台命令的名稱和簽名
     *
     * @var string
     */
    protected $signature = 'mail:send {user}';

    /**
     * 命令描述
     *
     * @var string
     */
```

```
    protected $description = 'Send a marketing email to a user';

    /**
     * 執行命令
     */
    public function handle(DripEmailer $drip): void
    {
        $drip->send(User::find($this->argument('user')));
    }
}
```

> **注意** 為了更好地復用程式碼，請儘量讓你的命令類保持輕量並且能夠延遲到應用服務中完成。上例中，我們注入了一個服務類來進行傳送電子郵件的「繁重工作」。

19.2.3 閉包命令

基於閉包的命令為將控制台命令定義為類提供了一種替代方法。與路由閉包可以替代 controller 一樣，可以將命令閉包視為命令類的替代。在 `app/Console/Kernel.php` 檔案的 `commands` 方法中 ，Laravel 載入 `routes/console.php` 檔案：

```
/**
 * 註冊閉包命令
 */
protected function commands(): void
{
    require base_path('routes/console.php');
}
```

儘管該檔案沒有定義 HTTP 路由，但它定義了進入應用程式的基於控制台的入口 (routes) 。在這個檔案中，你可以使用 `Artisan::command` 方法定義所有的閉包路由。 `command` 方法接受兩個參數： 命令名稱 和可呼叫的閉包，閉包接收命令的參數和選項：

```
Artisan::command('mail:send {user}', function (string $user) {
    $this->info("Sending email to: {$user}!");
});
```

該閉包繫結到基礎命令實例，因此你可以完全訪問通常可以在完整命令類上訪問的所有輔助方法。

19.2.3.1 Type-Hinting Dependencies

除了接受命令參數及選項外，命令閉包也可以使用類型約束從 服務容器 中解析其他的依賴關係：

```
use App\Models\User;
use App\Support\DripEmailer;

Artisan::command('mail:send {user}', function (DripEmailer $drip, string $user) {
    $drip->send(User::find($user));
});
```

19.2.3.2 閉包命令說明

在定義基於閉包的命令時，可以使用 `purpose` 方法向命令新增描述。當你運行 `php artisan list` 或 `php artisan help` 命令時，將顯示以下描述：

```
Artisan::command('mail:send {user}', function (string $user) {
    // ...
})->purpose('Send a marketing email to a user');
```

19.2.4 單例命令

> **警告** 要使用該特性，應用必須使用 `memcached`、`redis`、`dynamodb`、`database`、`file` 或 `array` 作為默認的快取驅動。另外，所有的伺服器必須與同一個中央快取伺服器通訊。

有時您可能希望確保一次只能運行一個命令實例。為此，你可以在命令類上實現 `Illuminate\Contracts\Console\Isolatable` 介面：

```
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;
use Illuminate\Contracts\Console\Isolatable;

class SendEmails extends Command implements Isolatable
{
    // ...
}
```

當命令被標記為 `Isolatable` 時，Laravel 會自動為該命令新增 `--isolated` 選項。當命令中使用這一選項時，Laravel 會確保不會有該命令的其他實例同時運行。Laravel 通過在應用的默認快取驅動中使用原子鎖來實現這一功能。如果這一命令有其他實例在運行，則該命令不會執行；不過，該命令仍然會使用成功退出狀態碼退出：

```
php artisan mail:send 1 --isolated
```

如果你想自己指定命令無法執行時返回的退出狀態碼，你可用通過 `isolated` 選項提供：

```
php artisan mail:send 1 --isolated=12
```

19.2.4.1 原子鎖到期時間

默認情況下，單例鎖會在命令完成後過期。或者如果命令被打斷且無法完成的話，鎖會在一小時後過期。不過你也可以通過定義命令的 `isolationLockExpiresAt` 方法來調整過期時間：

```
use DateTimeInterface;
use DateInterval;

/**
 * 定義單例鎖的到期時間
 */
public function isolationLockExpiresAt(): DateTimeInterface|DateInterval
{
    return now()->addMinutes(5);
}
```

19.3 定義輸入期望

在編寫控制台命令時，通常是通過參數和選項來收集使用者輸入的。 Laravel 讓你可以非常方便地在 `signature` 屬性中定義你期望使用者輸入的內容。`signature` 屬性允許使用單一且可讀性高，類似路由的語法來定義命令的名稱、參數和選項。

19.3.1 參數

使用者提供的所有參數和選項都用花括號括起來。在下面的示例中，該命令定義了一個必需的參數 `user`:

```
/**
 * 命令的名稱及其標識
 *
```

```
 * @var string
 */
protected $signature = 'mail:send {user}';
```

你亦可建立可選參數或為參數定義預設值：

```
// 可選參數...
'mail:send {user?}'

// 帶有預設值的可選參數...
'mail:send {user=foo}'
```

19.3.2 選項

選項類似於參數，是使用者輸入的另一種形式。在命令列中指定選項的時候，它們以兩個短橫線 (--) 作為前綴。這有兩種類型的選項：接收值和不接受值。不接收值的選項就像是一個布林值「開關」。我們來看一下這種類型的選項的示例：

```
/**
 * 命令的名稱及其標識
 *
 * @var string
 */
protected $signature = 'mail:send {user} {--queue}';
```

在這個例子中，在呼叫 Artisan 命令時可以指定 `--queue` 的開關。如果傳遞了 `--queue` 選項，該選項的值將會是 `true`。否則，其值將會是 `false`：

```
php artisan mail:send 1 --queue
```

19.3.2.1 帶值的選項

接下來，我們來看一下需要帶值的選項。如果使用者需要為一個選項指定一個值，則需要在選項名稱的末尾追加一個 = 號：

```
/**
 * 命令名稱及標識
 *
 * @var string
 */
protected $signature = 'mail:send {user} {--queue=}';
```

在這個例子中，使用者可以像如下所時的方式傳遞該選項的值。如果在呼叫命令時未指定該選項，則其值為 `null`：

```
php artisan mail:send 1 --queue=default
```

你還可以在選項名稱後指定其預設值。如果使用者沒有傳遞值給選項，將使用默認的值：

```
'mail:send {user} {--queue=default}'
```

19.3.2.2 選項簡寫

要在定義選項的時候指定一個簡寫，你可以在選項名前面使用 | 隔符將選項名稱與其簡寫分隔開來：

```
'mail:send {user} {--Q|queue}'
```

在終端上呼叫命令時，選項簡寫的前綴只用一個連字元，在為選項指定值時不應該包括=字元。

```
php artisan mail:send 1 -Qdefault
```

19.3.3 輸入陣列

如果你想要接收陣列陣列的參數或者選項，你可以使用 * 字元。首先，讓我們看一下指定了一個陣列參數的例

子：

```
'mail:send {user*}'
```

當呼叫這個方法的時候，user 參數的輸入參數將按順序傳遞給命令。例如，以下命令將會設定 user 的值為 foo 和 bar ：

```
php artisan mail:send 1 2
```

* 字元可以與可選的參數結合使用，允許您定義零個或多個參數實例：

```
'mail:send {user?*}'
```

19.3.3.1 選項陣列

當定義需要多個輸入值的選項時，傳遞給命令的每個選項值都應以選項名稱作為前綴：

```
'mail:send {--id=*}'
```

這樣的命令可以通過傳遞多個 --id 參數來呼叫：

```
php artisan mail:send --id=1 --id=2
```

19.3.4 輸入說明

你可以通過使用冒號將參數名稱與描述分隔來為輸入參數和選項指定說明。如果你需要一些額外的空間來定義命令，可以將它自由的定義在多行中：

```
/**
 * 控制台命令的名稱和簽名。
 *
 * @var string
 */
protected $signature = 'mail:send
                        {user : The ID of the user}
                        {--queue : Whether the job should be queued}';
```

19.4 命令 I/O

19.4.1 檢索輸入

當命令在執行時，你可能需要訪問命令所接受的參數和選項的值。為此，你可以使用 argument 和 option 方法。如果選項或參數不存在，將會返回 null：

```
/**
 * 執行控制台命令。
 */
public function handle(): void
{
    $userId = $this->argument('user');
}
```

如果你需要檢索所有的參數做為 array，請呼叫 arguments 方法：

```
$arguments = $this->arguments();
```

選項的檢索與參數一樣容易，使用 option 方法即可。如果要檢索所有的選項做為陣列，請呼叫 options 方法：

```
// 檢索一個指定的選項...
$queueName = $this->option('queue');

// 檢索所有選項做為陣列...
```

```
$options = $this->options();
```

19.4.2 互動式輸入

除了顯示輸出以外，你還可以要求使用者在執行命令期間提供輸入。ask 方法將詢問使用者指定的問題來接收使用者輸入，然後使用者輸入將會傳到你的命令中：

```
/**
 * 執行命令指令
 */
public function handle(): void
{
    $name = $this->ask('What is your name?');

    // ...
}
```

secret 方法與 ask 相似，區別在於使用者的輸入將不可見。這個方法在需要輸入一些諸如密碼之類的敏感資訊時是非常有用的：

```
$password = $this->secret('What is the password?');
```

19.4.2.1 請求確認

如果你需要請求使用者進行一個簡單的確認，可以使用 confirm 方法來實現。默認情況下，這個方法會返回 false。當然，如果使用者輸入 y 或 yes，這個方法將會返回 true。

```
if ($this->confirm('Do you wish to continue?')) {
    // ...
}
```

如有必要，你可以通過將 true 作為第二個參數傳遞給 confirm 方法，這樣就可以在默認情況下返回 true：

```
if ($this->confirm('Do you wish to continue?', true)) {
    // ...
}
```

19.4.2.2 自動補全

anticipate 方法可用於為可能的選項提供自動補全功能。使用者依然可以忽略自動補全的提示，進行任意回答：

```
$name = $this->anticipate('What is your name?', ['Taylor', 'Dayle']);
```

或者，你可以將一個閉包作為第二個參數傳遞給 anticipate 方法。每當使用者鍵入字元時，閉包涵數都會被呼叫。閉包涵數應該接受一個包含使用者輸入的字串形式的參數，並返回一個可供自動補全的選項的陣列：

```
$name = $this->anticipate('What is your address?', function (string $input) {
    // 返回自動完成組態...
});
```

19.4.2.3 多選擇問題

當詢問問題時，如果你需要給使用者一個預定義的選擇，你可以使用 choice 方法。如果沒有選項被選擇，你可以設定陣列索引的預設值去返回，通過這個方法的第三個參數去傳入索引：

```
$name = $this->choice(
    'What is your name?',
    ['Taylor', 'Dayle'],
    $defaultIndex
);
```

此外， choice 方法接受第四和第五可選參數 ，用於確定選擇有效響應的最大嘗試次數以及是否允許多次選擇：

```
$name = $this->choice(
    'What is your name?',
    ['Taylor', 'Dayle'],
    $defaultIndex,
    $maxAttempts = null,
    $allowMultipleSelections = false
);
```

19.4.3 文字輸出

你可以使用 line，info，comment，question 和 error 方法，傳送輸出到控制台。 這些方法中的每一個都會使用合適的 ANSI 顏色以展示不同的用途。例如，我們要為使用者展示一些常規資訊。通常，info 將會以綠色文字在控制台展示。

```
/**
 * Execute the console command.
 */
public function handle(): void
{
    // ...

    $this->info('The command was successful!');
}
```

輸出錯誤資訊，使用 error 方法。錯誤資訊通常使用紅色字型顯示：

```
$this->error('Something went wrong!');
```

你可以使用 line 方法輸出無色文字：

```
$this->line('Display this on the screen');
```

你可以使用 newLine 方法輸出空白行：

```
// 輸出單行空白...
$this->newLine();

// 輸出三行空白...
$this->newLine(3);
```

19.4.3.1 表格

table 方法可以輕鬆正確地格式化多行/多列資料。你需要做的就是提供表的列名和資料，Laravel 會自動為你計算合適的表格寬度和高度：

```
use App\Models\User;

$this->table(
    ['Name', 'Email'],
    User::all(['name', 'email'])->toArray()
);
```

19.4.3.2 進度條

對於長時間運行的任務，顯示一個進度條來告知使用者任務的完成情況會很有幫助。使用 withProgressBar 方法，Laravel 將顯示一個進度條，並在給定的可迭代值上推進每次迭代的進度：

```
use App\Models\User;

$users = $this->withProgressBar(User::all(), function (User $user) {
    $this->performTask($user);
});
```

有時，你可能需要更多手動控制進度條的前進方式。首先，定義流程將迭代的步驟總數。然後，在處理完每個項目後推進進度條：

```
$users = App\Models\User::all();

$bar = $this->output->createProgressBar(count($users));

$bar->start();

foreach ($users as $user) {
    $this->performTask($user);

    $bar->advance();
}

$bar->finish();
```

> **技巧：**有關更多高級選項，請查看 Symfony 進度條元件文件.

19.5 註冊命令

你的所有控制台命令都在您的應用程式的 `App\Console\Kernel` 類中註冊，這是你的應用程式的「控制台核心」。在此類的 `commands` 方法中，你將看到對核心的 `load` 方法的呼叫。 `load` 方法將掃描 `app/Console/Commands` 目錄並自動將其中包含的每個命令註冊到 Artisan。 你甚至可以自由地呼叫 `load` 方法來掃描其他目錄以尋找 Artisan 命令：

```
/**
 * Register the commands for the application.
 */
protected function commands(): void
{
    $this->load(__DIR__.'/Commands');
    $this->load(__DIR__.'/../Domain/Orders/Commands');

    // ...
}
```

如有必要，你可以通過將命令的類名新增到 `App\Console\Kernel` 類中的 `$commands` 屬性來手動註冊命令。如果你的核心上尚未定義此屬性，則應手動定義它。當 Artisan 啟動時，此屬性中列出的所有命令將由 服務容器 解析並註冊到 Artisan：

```
protected $commands = [
    Commands\SendEmails::class
];
```

19.6 以程式設計方式執行命令

有時你可能希望在 CLI 之外執行 Artisan 命令。例如，你可能希望從路由或 controller 執行 Artisan 命令。你可以使用 `Artisan` 外觀上的 `call` 方法來完成此操作。 `call` 方法接受命令的簽名名稱或類名作為其第一個參數，以及一個命令參數陣列作為第二個參數。將返回退出程式碼：

```
use Illuminate\Support\Facades\Artisan;

Route::post('/user/{user}/mail', function (string $user) {
    $exitCode = Artisan::call('mail:send', [
        'user' => $user, '--queue' => 'default'
    ]);

    // ...
});
```

或者，你可以將整個 Artisan 命令作為字串傳遞給 `call` 方法：

```
Artisan::call('mail:send 1 --queue=default');
```

19.6.1.1 傳遞陣列值

如果你的命令定義了一個接受陣列的選項，你可以將一組值傳遞給該選項：

```
use Illuminate\Support\Facades\Artisan;

Route::post('/mail', function () {
    $exitCode = Artisan::call('mail:send', [
        '--id' => [5, 13]
    ]);
});
```

19.6.1.2 傳遞布林值

如果你需要指定不接受字串值的選項的值，例如 `migrate:refresh` 命令上的 `--force` 標誌，則應傳遞 `true` 或 `false` 作為 選項：

```
$exitCode = Artisan::call('migrate:refresh', [
    '--force' => true,
]);
```

19.6.1.3 佇列 Artisan 命令

使用 `Artisan` 門面的 `queue` 方法，你甚至可以對 Artisan 命令進行排隊，以便你的 佇列工作者 在後台處理它們。在使用此方法之前，請確保你已組態佇列並正在運行佇列偵聽器：

```
use Illuminate\Support\Facades\Artisan;

Route::post('/user/{user}/mail', function (string $user) {
    Artisan::queue('mail:send', [
        'user' => $user, '--queue' => 'default'
    ]);

    // ...
});
```

使用 `onConnection` 和 `onQueue` 方法，你可以指定 Artisan 命令應分派到的連接或佇列：

```
Artisan::queue('mail:send', [
    'user' => 1, '--queue' => 'default'
])->onConnection('redis')->onQueue('commands');
```

19.6.2 從其他命令呼叫命令

有時你可能希望從現有的 Artisan 命令呼叫其他命令。你可以使用 `call` 方法來執行此操作。這個 `call` 方法接受命令名稱和命令參數/選項陣列：

```
/**
 * Execute the console command.
 */
public function handle(): void
{
    $this->call('mail:send', [
        'user' => 1, '--queue' => 'default'
    ]);

    // ...
}
```

如果你想呼叫另一個控制台命令並禁止其所有輸出，你可以使用 `callSilently` 方法。 `callSilently` 方

法與 call 方法具有相同的簽名：

```
$this->callSilently('mail:send', [
    'user' => 1, '--queue' => 'default'
]);
```

19.7 訊號處理

正如你可能知道的，作業系統允許向運行中的處理程序傳送訊號。例如，「SIGTERM」訊號是作業系統要求程序終止的方式。如果你想在 Artisan 控制台命令中監聽訊號，並在訊號發生時執行程式碼，你可以使用 trap 方法。

```
/**
 * 執行控制台命令。
 */
public function handle(): void
{
    $this->trap(SIGTERM, fn () => $this->shouldKeepRunning = false);

    while ($this->shouldKeepRunning) {
        // ...
    }
}
```

為了一次監聽多個訊號，你可以向 trap 方法提供一個訊號陣列。

```
$this->trap([SIGTERM, SIGQUIT], function (int $signal) {
    $this->shouldKeepRunning = false;

    dump($signal); // SIGTERM / SIGQUIT
});
```

19.8 Stub 定製

Artisan 控制台的 make 命令用於建立各種類，例如 controller 、作業、遷移和測試。這些類是使用「stub」檔案生成的，這些檔案中會根據你的輸入填充值。但是，你可能需要對 Artisan 生成的檔案進行少量更改。為此，你可以使用以下 stub:publish 命令將最常見的 Stub 命令發佈到你的應用程式中，以便可以自訂它們：

```
php artisan stub:publish
```

已發佈的 stub 將存放於你的應用根目錄下的 stubs 目錄中。對這些 stub 進行任何改動都將在你使用 Artisan make 命令生成相應的類的時候反映出來。

19.9 事件

Artisan 在運行命令時會調度三個事件： Illuminate\Console\Events\ArtisanStarting，Illuminate\Console\Events\CommandStarting 和 Illuminate\Console\Events\CommandFinished。當 Artisan 開始執行階段，會立即調度 ArtisanStarting 事件。接下來，在命令運行之前立即調度 CommandStarting 事件。最後，一旦命令執行完畢，就會調度 CommandFinished 事件。

20 廣播

20.1 介紹

在許多現代 Web 應用程式中，WebSockets 用於實現即時的、即時更新的使用者介面。當伺服器上的某些資料更新時，通常會傳送一條消息到 WebSocket 連接，以由客戶端處理。WebSockets 提供了一種更有效的替代方法，可以連續輪詢應用程式伺服器以反映 UI 中應該反映的資料更改。

舉個例子，假設你的應用程式能夠將使用者的資料匯出為 CSV 檔案並通過電子郵件傳送給他們。但是，建立這個 CSV 檔案需要幾分鐘的時間，因此你選擇在佇列任務中建立和傳送 CSV。當 CSV 檔案已經建立並行送給使用者後，我們可以使用事件廣播來分發 `App\Events\UserDataExported` 事件，該事件由我們應用程式的 JavaScript 接收。一旦接收到事件，我們可以向使用者顯示消息，告訴他們他們的 CSV 已通過電子郵件傳送給他們，而無需刷新頁面。

為了幫助你建構此類特性，Laravel 使得在 WebSocket 連接上“廣播”你的伺服器端 Laravel 事件變得簡單。廣播你的 Laravel 事件允許你在你的伺服器端 Laravel 應用和客戶端 JavaScript 應用之間共享相同的事件名稱和資料。

廣播背後的核心概念很簡單：客戶端在前端連接到命名通道，而你的 Laravel 應用在後端向這些通道廣播事件。這些事件可以包含任何你想要向前端提供的其他資料。

20.1.1.1 支援的驅動程式

默認情況下，Laravel 為你提供了兩個伺服器端廣播驅動程式可供選擇：Pusher Channels 和 Ably。但是，社區驅動的包，如 laravel-websockets 和 soketi 提供了不需要商業廣播提供者的其他廣播驅動程式。

> **注意** 在深入瞭解事件廣播之前，請確保已閱讀 Laravel 的事件和偵聽器文件。

20.2 伺服器端安裝

為了開始使用 Laravel 的事件廣播，我們需要在 Laravel 應用程式中進行一些組態，並安裝一些包。

事件廣播是通過伺服器端廣播驅動程式實現的，該驅動程式廣播你的 Laravel 事件，以便 Laravel Echo（一個 JavaScript 庫）可以在瀏覽器客戶端中接收它們。不用擔心 - 我們將逐步介紹安裝過程的每個部分。

20.2.1 組態

所有應用程式的事件廣播組態都儲存在 `config/broadcasting.php` 組態檔案中。Laravel 支援多個廣播驅動程式：Pusher Channels、Redis 和用於本地開發和偵錯的 `log` 驅動程式。此外，還包括一個 `null` 驅動程式，它允許你在測試期間完全停用廣播。`config/broadcasting.php` 組態檔案中包含每個驅動程式的組態示例。

20.2.1.1 廣播服務提供商

在廣播任何事件之前，您首先需要註冊 `App\Providers\BroadcastServiceProvider`。在新的 Laravel 應用程式中，您只需要在 `config/app.php` 組態檔案的 `providers` 陣列中取消註釋此提供程序即可。這個 `BroadcastServiceProvider` 包含了註冊廣播授權路由和回呼所需的程式碼。

20.2.1.2 佇列組態

您還需要組態和運行一個佇列工作者。所有事件廣播都是通過排隊的作業完成的，以確保您的應用程式的響應時間不會受到廣播事件的影響。

20.2.2 Pusher Channels

如果您計畫使用 Pusher Channels 廣播您的事件，您應該使用 Composer 包管理器安裝 Pusher Channels PHP SDK：

```
composer require pusher/pusher-php-server
```

接下來，您應該在 `config/broadcasting.php` 組態檔案中組態 Pusher Channels 憑據。此檔案中已經包含了一個示例 Pusher Channels 組態，讓您可以快速指定您的金鑰、金鑰、應用程式 ID。通常，這些值應該通過 `PUSHER_APP_KEY`、`PUSHER_APP_SECRET` 和 `PUSHER_APP_ID` 環境變數 設定：

```
PUSHER_APP_ID=your-pusher-app-id
PUSHER_APP_KEY=your-pusher-key
PUSHER_APP_SECRET=your-pusher-secret
PUSHER_APP_CLUSTER=mt1
```

`config/broadcasting.php` 檔案的 `pusher` 組態還允許您指定 Channels 支援的其他 `options`，例如叢集。

接下來，您需要在您的 `.env` 檔案中將廣播驅動程式更改為 `pusher`：

```
BROADCAST_DRIVER=pusher
```

最後，您已經準備好安裝和組態 Laravel Echo，它將在客戶端接收廣播事件。

20.2.2.1 開放原始碼的 Pusher 替代品

laravel-websockets 和 soketi 軟體包提供了適用於 Laravel 的 Pusher 相容的 WebSocket 伺服器。這些軟體包允許您利用 Laravel 廣播的全部功能，而無需商業 WebSocket 提供程序。有關安裝和使用這些軟體包的更多資訊，請參閱我們的開源替代品文件。

20.2.3 Ably

> **注意** 下面的文件介紹了如何在“Pusher 相容”模式下使用 Ably。然而，Ably 團隊推薦並維護一個廣播器和 Echo 客戶端，能夠利用 Ably 提供的獨特功能。有關使用 Ably 維護的驅動程式的更多資訊，請參閱 Ably 的 Laravel 廣播器文件。

如果您計畫使用 Ably 廣播您的事件，則應使用 Composer 軟體包管理器安裝 Ably PHP SDK：

```
composer require ably/ably-php
```

接下來，您應該在 `config/broadcasting.php` 組態檔案中組態您的 Ably 憑據。該檔案已經包含了一個示例 Ably 組態，允許您快速指定您的金鑰。通常，此值應通過 `ABLY_KEY` 環境變數進行設定：

```
ABLY_KEY=your-ably-key
```

Next, you will need to change your broadcast driver to `ably` in your `.env` file:

```
BROADCAST_DRIVER=ably
```

接下來，您需要在`.env` 檔案中將廣播驅動程式更改為 `ably`：

20.2.4 開源替代方案

20.2.4.1 PHP

laravel-websockets 是一個純 PHP 的，與 Pusher 相容的 Laravel WebSocket 包。該包允許您充分利用 Laravel 廣播的功能，而無需商業 WebSocket 提供商。有關安裝和使用此包的更多資訊，請參閱其官方文件。

20.2.4.2 Node

Soketi 是一個基於 Node 的，與 Pusher 相容的 Laravel WebSocket 伺服器。在幕後，Soketi 利用 µWebSockets.js 來實現極端的可擴展性和速度。該包允許您充分利用 Laravel 廣播的功能，而無需商業 WebSocket 提供商。有關安裝和使用此包的更多資訊，請參閱其官方文件。

20.3 客戶端安裝

20.3.1 Pusher Channels

Laravel Echo 是一個 JavaScript 庫，可以輕鬆訂閱通道並監聽由伺服器端廣播驅動程式廣播的事件。您可以通過 NPM 包管理器安裝 Echo。在此示例中，我們還將安裝 `pusher-js` 包，因為我們將使用 Pusher Channels 廣播器：

```
npm install --save-dev laravel-echo pusher-js
```

安裝 Echo 後，您可以在應用程式的 JavaScript 中建立一個新的 Echo 實例。一個很好的地方是在 Laravel 框架附帶的 `resources/js/bootstrap.js` 檔案的底部建立它。默認情況下，該檔案中已包含一個示例 Echo 組態 - 您只需取消註釋即可：

```
import Echo from 'laravel-echo';
import Pusher from 'pusher-js';

window.Pusher = Pusher;

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: import.meta.env.VITE_PUSHER_APP_KEY,
    cluster: import.meta.env.VITE_PUSHER_APP_CLUSTER,
    forceTLS: true
});
```

一旦您根據自己的需求取消註釋並調整了 Echo 組態，就可以編譯應用程式的資產：

```
npm run dev
```

> **注意** 要瞭解有關編譯應用程式的 JavaScript 資產的更多資訊，請參閱 Vite 上的文件。

20.3.1.1 使用現有的客戶端實例

如果您已經有一個預組態的 Pusher Channels 客戶端實例，並希望 Echo 利用它，您可以通過 `client` 組態選項將其傳遞給 Echo：

```
import Echo from 'laravel-echo';
import Pusher from 'pusher-js';

const options = {
    broadcaster: 'pusher',
    key: 'your-pusher-channels-key'
}
```

```
window.Echo = new Echo({
    ...options,
    client: new Pusher(options.key, options)
});
```

20.3.2 Ably

> **注意** 下面的文件討論如何在“Pusher 相容性”模式下使用 Ably。但是，Ably 團隊推薦和維護了一個廣播器和 Echo 客戶端，可以利用 Ably 提供的獨特功能。有關使用由 Ably 維護的驅動程式的更多資訊，請查看 Ably 的 Laravel 廣播器文件。

Laravel Echo 是一個 JavaScript 庫，可以輕鬆訂閱通道並偵聽伺服器端廣播驅動程式廣播的事件。您可以通過 NPM 包管理器安裝 Echo。在本示例中，我們還將安裝 `pusher-js` 包。

您可能會想為什麼我們要安裝 `pusher-js` JavaScript 庫，即使我們使用 Ably 來廣播事件。幸運的是，Ably 包括 Pusher 相容性模式，讓我們可以在客戶端應用程式中使用 Pusher 協議來偵聽事件：

```
npm install --save-dev laravel-echo pusher-js
```

在繼續之前，你應該在你的 Ably 應用設定中啟用 Pusher 協議支援。你可以在你的 Ably 應用設定儀表板的“協議介面卡設定”部分中啟用此功能。

安裝 Echo 後，你可以在應用的 JavaScript 中建立一個新的 Echo 實例。一個很好的地方是在 Laravel 框架附帶的 `resources/js/bootstrap.js` 檔案底部。默認情況下，此檔案中已包含一個示例 Echo 組態；但是，`bootstrap.js` 檔案中的默認組態是為 Pusher 設計的。你可以複製以下組態來將組態轉換為 Ably：

```
import Echo from 'laravel-echo';
import Pusher from 'pusher-js';

window.Pusher = Pusher;

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: import.meta.env.VITE_ABLY_PUBLIC_KEY,
    wsHost: 'realtime-pusher.ably.io',
    wsPort: 443,
    disableStats: true,
    encrypted: true,
});
```

請注意，我們的 Ably Echo 組態引用了一個 `VITE_ABLY_PUBLIC_KEY` 環境變數。該變數的值應該是你的 Ably 公鑰。你的公鑰是出現在 Ably 金鑰的 `:` 字元之前的部分。

一旦你根據需要取消註釋並調整 Echo 組態，你可以編譯應用的資產：

```
npm run dev
```

> **注意** 要瞭解有關編譯應用程式的 JavaScript 資產的更多資訊，請參閱 Vite 的文件。

20.4 概念概述

Laravel 的事件廣播允許你使用基於驅動程式的 WebSocket 方法，將伺服器端 Laravel 事件廣播到客戶端的 JavaScript 應用程式。目前，Laravel 附帶了 Pusher Channels 和 Ably 驅動程式。可以使用 Laravel Echo JavaScript 包輕鬆地在客戶端消耗這些事件。

事件通過“通道”廣播，可以指定為公共或私有。任何訪問您的應用程式的使用者都可以訂閱公共頻道，無需進行身份驗證或授權；但是，要訂閱私有頻道，使用者必須經過身份驗證和授權以便監聽該頻道。

注意
如果您想探索 Pusher 的開源替代品，請查看開源替代品。

20.4.1 使用示例應用程式

在深入瞭解事件廣播的每個元件之前，讓我們使用電子商務店舖作為示例進行高級概述。

在我們的應用程式中，假設我們有一個頁面，允許使用者查看其訂單的發貨狀態。假設在應用程式處理髮貨狀態更新時，將觸發一個 `OrderShipmentStatusUpdated` 事件：

```
use App\Events\OrderShipmentStatusUpdated;

OrderShipmentStatusUpdated::dispatch($order);
```

20.4.1.1 ShouldBroadcast 介面

當使用者查看其訂單之一時，我們不希望他們必須刷新頁面才能查看狀態更新。相反，我們希望在建立更新時將更新廣播到應用程式。因此，我們需要使用 `ShouldBroadcast` 介面標記 `OrderShipmentStatusUpdated` 事件。這將指示 Laravel 在觸發事件時廣播該事件：

```
<?php

namespace App\Events;

use App\Models\Order;
use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Queue\SerializesModels;

class OrderShipmentStatusUpdated implements ShouldBroadcast
{
    /**
     * The order instance.
     *
     * @var \App\Order
     */
    public $order;
}
```

`ShouldBroadcast` 介面要求我們的事件定義一個 `broadcastOn` 方法。該方法負責返回事件應廣播到的頻道。在生成的事件類中已經定義了這個方法的空椿，所以我們只需要填寫它的細節即可。我們只希望訂單的建立者能夠查看狀態更新，因此我們將事件廣播到與訂單相關的私有頻道上：

```
use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\PrivateChannel;

/**
 * 獲取事件應該廣播到的頻道。
 */
public function broadcastOn(): Channel
{
    return new PrivateChannel('orders.'.$this->order->id);
}
```

如果你希望事件廣播到多個頻道，可以返回一個 `array`：

```
use Illuminate\Broadcasting\PrivateChannel;

/**
 * 獲取事件應該廣播到的頻道。
 *
 * @return array<int, \Illuminate\Broadcasting\Channel>
```

```
 */
public function broadcastOn(): array
{
    return [
        new PrivateChannel('orders.'.$this->order->id),
        // ...
    ];
}
```

20.4.1.2 授權頻道

記住，使用者必須被授權才能監聽私有頻道。我們可以在應用程式的 `routes/channels.php` 檔案中定義頻道授權規則。在這個例子中，我們需要驗證任何試圖監聽私有 `orders.1` 頻道的使用者是否實際上是訂單的建立者：

```
use App\Models\Order;
use App\Models\User;

Broadcast::channel('orders.{orderId}', function (User $user, int $orderId) {
    return $user->id === Order::findOrNew($orderId)->user_id;
});
```

`channel` 方法接受兩個參數：頻道名稱和一個回呼函數，該函數返回 `true` 或 `false`，表示使用者是否被授權監聽該頻道。

所有授權回呼函數的第一個參數是當前認證的使用者，其餘的萬用字元參數是它們的後續參數。在此示例中，我們使用`{orderId}`預留位置來指示頻道名稱的“ID”部分是萬用字元。

20.4.1.3 監聽事件廣播

接下來，我們只需要在 JavaScript 應用程式中監聽事件即可。我們可以使用 Laravel Echo 來完成這個過程。首先，我們使用 `private` 方法訂閱私有頻道。然後，我們可以使用 `listen` 方法來監聽 `OrderShipmentStatusUpdated` 事件。默認情況下，廣播事件的所有公共屬性將被包括在廣播事件中：

```
Echo.private(`orders.${orderId}`)
    .listen('OrderShipmentStatusUpdated', (e) => {
        console.log(e.order);
    });
```

20.5 定義廣播事件

要通知 Laravel 給定事件應該被廣播，您必須在事件類上實現 `Illuminate\Contracts\Broadcasting\ShouldBroadcast` 介面。該介面已經被框架生成的所有事件類匯入，因此您可以輕鬆地將其新增到任何事件中。

`ShouldBroadcast` 介面要求您實現一個單獨的方法:`broadcastOn`。`broadcastOn` 方法應該返回一個頻道或頻道陣列，事件應該在這些頻道上廣播。這些頻道應該是 `Channel`、`PrivateChannel` 或 `PresenceChannel` 的實例。`Channel` 的實例表示任何使用者都可以訂閱的公共頻道，而 `PrivateChannel` 和 `PresenceChannel` 表示需要頻道授權的私有頻道：

```
<?php

namespace App\Events;

use App\Models\User;
use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
```

```
use Illuminate\Queue\SerializesModels;

class ServerCreated implements ShouldBroadcast
{
    use SerializesModels;

    /**
     * 建立一個新的事件實例。
     */
    public function __construct(
        public User $user,
    ) {}

    /**
     * 獲取事件應該廣播到哪些頻道。
     *
     * @return array<int, \Illuminate\Broadcasting\Channel>
     */
    public function broadcastOn(): array
    {
        return [
            new PrivateChannel('user.'.$this->user->id),
        ];
    }
}
```

實現 `ShouldBroadcast` 介面後，您只需要像平常一樣觸發事件。一旦事件被觸發，一個佇列任務將自動使用指定的廣播驅動程式廣播該事件。

20.5.1 廣播名稱

默認情況下，Laravel 將使用事件類名廣播事件。但是，您可以通過在事件上定義 `broadcastAs` 方法來自訂廣播名稱：

```
/**
 * 活動的廣播名稱
 */
public function broadcastAs(): string
{
    return 'server.created';
}
```

如果您使用 `broadcastAs` 方法自訂廣播名稱，則應確保使用前導“.”字元註冊您的偵聽器。這將指示 Echo 不將應用程式的命名空間新增到事件中：

```
.listen('.server.created', function (e) {
    ....
});
```

20.5.2 廣播資料

當廣播事件時，所有 `public` 屬性都將自動序列化並廣播為事件負載，使您能夠從 JavaScript 應用程式中訪問其任何公共資料。例如，如果您的事件具有單個公共 `$user` 屬性，其中包含 Eloquent 模型，則事件的廣播負載將是：

```
{
    "user": {
        "id": 1,
        "name": "Patrick Stewart"
        ...
    }
}
```

但是，如果您希望更精細地控制廣播負載，則可以向事件中新增 broadcastWith 方法。該方法應該返回您希望作為事件負載廣播的資料陣列：

```
/**
 * 獲取要廣播的資料。
 *
 * @return array<string, mixed>
 */
public function broadcastWith(): array
{
    return ['id' => $this->user->id];
}
```

20.5.3 廣播佇列

默認情況下，每個廣播事件都會被放置在您在 queue.php 組態檔案中指定的默認佇列連接的默認佇列上。您可以通過在事件類上定義 connection 和 queue 屬性來自訂廣播器使用的佇列連接和名稱：

```
/**
 * 廣播事件時要使用的佇列連接的名稱。
 *
 * @var string
 */
public $connection = 'redis';

/**
 * 廣播作業要放置在哪個佇列上的名稱。
 *
 * @var string
 */
public $queue = 'default';
```

或者，您可以通過在事件上定義一個 broadcastQueue 方法來自訂佇列名稱：

```
/**
 * 廣播作業放置在其上的佇列的名稱。
 */
public function broadcastQueue(): string
{
    return 'default';
}
```

如果您想要使用 sync 佇列而不是默認的佇列驅動程式來廣播事件，您可以實現 ShouldBroadcastNow 介面而不是 ShouldBroadcast 介面：

```
<?php

use Illuminate\Contracts\Broadcasting\ShouldBroadcastNow;

class OrderShipmentStatusUpdated implements ShouldBroadcastNow
{
    // ...
}
```

20.5.4 廣播條件

有時候您只想在給定條件為真時才廣播事件。您可以通過在事件類中新增一個 broadcastWhen 方法來定義這些條件：

```
/**
 * 確定此事件是否應該廣播。
 */
public function broadcastWhen(): bool
{
```

```
    return $this->order->value > 100;
}
```

20.5.4.1 廣播和資料庫事務

當在資料庫事務中分派廣播事件時，它們可能會在資料庫事務提交之前被佇列處理。當這種情況發生時，在資料庫中對模型或資料庫記錄所做的任何更新可能尚未反映在資料庫中。此外，在事務中建立的任何模型或資料庫記錄可能不存在於資料庫中。如果您的事件依賴於這些模型，則在處理廣播事件的作業時可能會出現意外錯誤。

如果您的佇列連接的 `after_commit` 組態選項設定為 `false`，您仍然可以通過在事件類上定義 `$afterCommit` 屬性來指示特定的廣播事件在所有打開的資料庫事務提交後被調度：

```
<?php

namespace App\Events;

use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Queue\SerializesModels;

class ServerCreated implements ShouldBroadcast
{
    use SerializesModels;

    public $afterCommit = true;
}
```

注意 要瞭解更多有關解決這些問題的資訊，請查閱有關佇列作業和資料庫事務的文件。

20.6 授權頻道

私有頻道需要您授權當前已驗證的使用者是否實際上可以監聽該頻道。這可以通過向您的 Laravel 應用程式傳送帶有頻道名稱的 HTTP 請求來完成，並允許您的應用程式確定使用者是否可以在該頻道上監聽。當使用 Laravel Echo 時，將自動進行授權訂閱私有頻道的 HTTP 請求；但是，您需要定義正確的路由來響應這些請求。

20.6.1 定義授權路由

幸運的是，Laravel 可以輕鬆定義用於響應頻道授權請求的路由。在您的 Laravel 應用程式中包含的 `App\Providers\BroadcastServiceProvider` 中，您將看到對 `Broadcast::routes` 方法的呼叫。此方法將註冊 `/broadcasting/auth` 路由以處理授權請求：

```
Broadcast::routes();
```

`Broadcast::routes` 方法將自動將其路由放置在 `web` 中介軟體組中；但是，如果您想自訂分配的屬性，則可以將路由屬性陣列傳遞給該方法：

```
Broadcast::routes($attributes);
```

20.6.1.1 自訂授權終點

默認情況下，Echo 將使用 `/broadcasting/auth` 終點來授權頻道訪問。但是，您可以通過將 `authEndpoint` 組態選項傳遞給 Echo 實例來指定自己的授權終點：

```
window.Echo = new Echo({
    broadcaster: 'pusher',
    // ...
    authEndpoint: '/custom/endpoint/auth'
});
```

20.6.1.2 自訂授權請求

您可以在初始化 Echo 時提供自訂授權器來自訂 Laravel Echo 如何執行授權請求：

```
window.Echo = new Echo({
    // ...
    authorizer: (channel, options) => {
        return {
            authorize: (socketId, callback) => {
                axios.post('/api/broadcasting/auth', {
                    socket_id: socketId,
                    channel_name: channel.name
                })
                .then(response => {
                    callback(null, response.data);
                })
                .catch(error => {
                    callback(error);
                });
            }
        };
    },
})
```

20.6.2 定義授權回呼函數

接下來，我們需要定義實際確定當前認證使用者是否可以收聽給定頻道的邏輯。這是在您的應用程式中包含的 `routes/channels.php` 檔案中完成的。在該檔案中，您可以使用 `Broadcast::channel` 方法來註冊頻道授權回呼函數：

```
use App\Models\User;

Broadcast::channel('orders.{orderId}', function (User $user, int $orderId) {
    return $user->id === Order::findOrNew($orderId)->user_id;
});
```

`channel` 方法接受兩個參數：頻道名稱和一個回呼函數，該回呼函數返回 `true` 或 `false`，指示使用者是否有權限在頻道上收聽。

所有授權回呼函數都接收當前認證使用者作為其第一個參數，任何其他萬用字元參數作為其後續參數。在此示例中，我們使用 `{orderId}` 預留位置來指示頻道名稱的 “ID” 部分是萬用字元。

您可以使用 `channel:list` Artisan 命令查看應用程式的廣播授權回呼列表：

```
php artisan channel:list
```

20.6.2.1 授權回呼模型繫結

與 HTTP 路由一樣，頻道路由也可以利用隱式和顯式的路由模型繫結。例如，您可以請求一個實際的 `Order` 模型實例，而不是接收一個字串或數字訂單 ID：

```
use App\Models\Order;
use App\Models\User;

Broadcast::channel('orders.{order}', function (User $user, Order $order) {
    return $user->id === $order->user_id;
});
```

> **警告** 與 HTTP 路由模型繫結不同，頻道模型繫結不支援自動隱式模型繫結範圍。但是，這很少是問題，因為大多數頻道可以基於單個模型的唯一主鍵進行範圍限制。

20.6.2.2 授權回呼身份驗證

私有和存在廣播頻道會通過您的應用程式的默認身份驗證保護當前使用者。如果使用者未經過身份驗證，則頻道授權將自動被拒絕，並且不會執行授權回呼。但是，您可以分配多個自訂守衛，以根據需要對傳入請求進行身份驗證：

```
Broadcast::channel('channel', function () {
    // ...
}, ['guards' => ['web', 'admin']]);
```

20.6.3 定義頻道類

如果您的應用程式正在消耗許多不同的頻道，則您的 `routes/channels.php` 檔案可能會變得臃腫。因此，您可以使用頻道類而不是使用閉包來授權頻道。要生成一個頻道類，請使用 `make:channel` Artisan 命令。該命令將在 `App/Broadcasting` 目錄中放置一個新的頻道類。

```
php artisan make:channel OrderChannel
```

接下來，在您的 `routes/channels.php` 檔案中註冊您的頻道：

```
use App\Broadcasting\OrderChannel;

Broadcast::channel('orders.{order}', OrderChannel::class);
```

最後，您可以將頻道授權邏輯放在頻道類的 `join` 方法中。這個 `join` 方法將包含您通常放置在頻道授權閉包中的相同邏輯。您還可以利用頻道模型繫結：

```
<?php

namespace App\Broadcasting;

use App\Models\Order;
use App\Models\User;

class OrderChannel
{
    /**
     * 建立一個新的頻道實例。
     */
    public function __construct()
    {
        // ...
    }

    /**
     * 驗證使用者對頻道的存取權。
     */
    public function join(User $user, Order $order): array|bool
    {
        return $user->id === $order->user_id;
    }
}
```

> **注意** 像 Laravel 中的許多其他類一樣，頻道類將自動由服務容器解析。因此，您可以在其建構函式中聲明頻道所需的任何依賴關係。

20.7 廣播事件

一旦您定義了一個事件並使用 `ShouldBroadcast` 介面標記了它，您只需要使用事件的 `dispatch` 方法來觸發事件。事件調度程序會注意到該事件已標記為 `ShouldBroadcast` 介面，並將該事件排隊進行廣播：

```
use App\Events\OrderShipmentStatusUpdated;
```

```
OrderShipmentStatusUpdated::dispatch($order);
```

20.7.1 只發給其他人

在建構使用事件廣播的應用程式時，您可能需要將事件廣播給給定頻道的所有訂閱者，除了當前使用者。您可以使用 broadcast 幫助器和 toOthers 方法來實現：

```
use App\Events\OrderShipmentStatusUpdated;

broadcast(new OrderShipmentStatusUpdated($update))->toOthers();
```

為了更好地理解何時需要使用 toOthers 方法，讓我們想像一個任務列表應用程式，使用者可以通過輸入任務名稱來建立新任務。為了建立任務，您的應用程式可能會向/task URL 發出請求，該請求廣播任務的建立並返回新任務的 JSON 表示。當 JavaScript 應用程式從端點接收到響應時，它可能會直接將新任務插入到其任務列表中，如下所示：

```
axios.post('/task', task)
    .then((response) => {
        this.tasks.push(response.data);
    });
```

然而，請記住，我們也會廣播任務的建立。如果 JavaScript 應用程式也在監聽此事件以便將任務新增到任務列表中，那麼您的列表中將有重複的任務：一個來自端點，一個來自廣播。您可以使用 toOthers 方法來解決這個問題，指示廣播器不要向當前使用者廣播事件。

> **警告** 您的事件必須使用 Illuminate\Broadcasting\InteractsWithSockets 特性才能呼叫 toOthers 方法。

20.7.1.1 組態

當您初始化一個 Laravel Echo 實例時，將為連接分配一個套接字 ID。如果您正在使用全域的 Axios 實例從 JavaScript 應用程式發出 HTTP 請求，則套接字 ID 將自動附加到每個傳出請求作為 X-Socket-ID 頭。然後，當您呼叫 toOthers 方法時，Laravel 將從標頭中提取套接字 ID，並指示廣播器不向具有該套接字 ID 的任何連接廣播。

如果您沒有使用全域的 Axios 實例，您需要手動組態 JavaScript 應用程式，以在所有傳出請求中傳送 X-Socket-ID 標頭。您可以使用 Echo.socketId 方法檢索 socket ID：

```
var socketId = Echo.socketId();
```

20.7.2 定製連接

如果您的應用程式與多個廣播連接互動，並且您想使用除默認之外的廣播器廣播事件，則可以使用 via 方法指定要將事件推送到哪個連接：

```
use App\Events\OrderShipmentStatusUpdated;

broadcast(new OrderShipmentStatusUpdated($update))->via('pusher');
```

或者，您可以在事件的建構函式中呼叫 broadcastVia 方法指定事件的廣播連接。不過，在這樣做之前，您應該確保事件類使用了 InteractsWithBroadcasting trait：

```
<?php

namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithBroadcasting;
use Illuminate\Broadcasting\InteractsWithSockets;
```

```
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Queue\SerializesModels;

class OrderShipmentStatusUpdated implements ShouldBroadcast
{
    use InteractsWithBroadcasting;

    /**
     * 建立一個新的事件實例。
     */
    public function __construct()
    {
        $this->broadcastVia('pusher');
    }
}
```

20.8 接收廣播

20.8.1 監聽事件

一旦您 安裝並實例化了 Laravel Echo，您就可以開始監聽從 Laravel 應用程式廣播的事件。首先使用 `channel` 方法檢索通道實例，然後呼叫 `listen` 方法來監聽指定的事件：

```
Echo.channel(`orders.${this.order.id}`)
    .listen('OrderShipmentStatusUpdated', (e) => {
        console.log(e.order.name);
    });
```

如需在私有頻道上監聽事件，請改用 `private` 方法。您可以繼續鏈式呼叫 `listen` 方法以偵聽單個頻道上的多個事件：

```
Echo.private(`orders.${this.order.id}`)
    .listen(/* ... */)
    .listen(/* ... */)
    .listen(/* ... */);
```

20.8.1.1 停止監聽事件

如果您想停止偵聽給定事件而不離開頻道，可以使用 `stopListening` 方法：

```
Echo.private(`orders.${this.order.id}`)
    .stopListening('OrderShipmentStatusUpdated')
```

20.8.2 離開頻道

要離開頻道，請在 Echo 實例上呼叫 `leaveChannel` 方法：

```
Echo.leaveChannel(`orders.${this.order.id}`);
```

如果您想離開頻道以及其關聯的私有和預sence 頻道，則可以呼叫 `leave` 方法：

```
Echo.leave(`orders.${this.order.id}`);
```

20.8.3 命名空間

您可能已經注意到在上面的示例中，我們沒有指定事件類的完整 `App\Events` 命名空間。這是因為 Echo 將自動假定事件位於 `App\Events` 命名空間中。但是，您可以在實例化 Echo 時通過傳遞 `namespace` 組態選項來

組態根命名空間：

```
window.Echo = new Echo({
    broadcaster: 'pusher',
    // ...
    namespace: 'App.Other.Namespace'
});
```

或者，您可以在使用 Echo 訂閱時使用。前綴為事件類新增前綴。這將允許您始終指定完全限定的類名：

```
Echo.channel('orders')
    .listen('.Namespace\\Event\\Class', (e) => {
        // ...
    });
```

20.9 存在頻道

存在頻道基於私有頻道的安全性，並公開了訂閱頻道使用者的附加功能。這使得建構強大的協作應用程式功能變得容易，例如在另一個使用者正在查看同一頁面時通知使用者，或者列出聊天室的使用者。

20.9.1 授權存在頻道

所有存在頻道也都是私有頻道，因此使用者必須獲得存取權。但是，在為存在頻道定義授權回呼時，如果使用者被授權加入該頻道，您將不會返回 `true`。相反，您應該返回有關使用者的資料陣列。

授權回呼返回的資料將在 JavaScript 應用程式中的存在頻道事件偵聽器中可用。如果使用者沒有被授權加入存在頻道，則應返回 `false` 或 `null`：

```
use App\Models\User;

Broadcast::channel('chat.{roomId}', function (User $user, int $roomId) {
    if ($user->canJoinRoom($roomId)) {
        return ['id' => $user->id, 'name' => $user->name];
    }
});
```

20.9.2 加入存在頻道

要加入存在頻道，您可以使用 Echo 的 `join` 方法。`join` 方法將返回一個 `PresenceChannel` 實現，除了公開 `listen` 方法外，還允許您訂閱 `here`，`joining` 和 `leaving` 事件。

```
Echo.join(`chat.${roomId}`)
    .here((users) => {
        // ...
    })
    .joining((user) => {
        console.log(user.name);
    })
    .leaving((user) => {
        console.log(user.name);
    })
    .error((error) => {
        console.error(error);
    });
```

成功加入頻道後，`here` 回呼將立即執行，並接收一個包含所有當前訂閱頻道使用者資訊的陣列。`joining` 方法將在新使用者加入頻道時執行，而 `leaving` 方法將在使用者離開頻道時執行。當認證端點返回 HTTP 狀態碼 200 以外的程式碼或存在解析返回的 JSON 時，將執行 `error` 方法。

20.9.3 向 Presence 頻道廣播

Presence 頻道可以像公共頻道或私有頻道一樣接收事件。以聊天室為例，我們可能希望將 `NewMessage` 事件廣播到聊天室的 Presence 頻道中。為此，我們將從事件的 `broadcastOn` 方法返回一個 `PresenceChannel` 實例：

```
/**
 * Get the channels the event should broadcast on.
 *
 * @return array<int, \Illuminate\Broadcasting\Channel>
 */
public function broadcastOn(): array
{
    return [
        new PresenceChannel('room.'.$this->message->room_id),
    ];
}
```

與其他事件一樣，您可以使用 `broadcast` 助手和 `toOthers` 方法來排除當前使用者接收廣播：

```
broadcast(new NewMessage($message));

broadcast(new NewMessage($message))->toOthers();
```

與其他類型的事件一樣，您可以使用 Echo 的 `listen` 方法來監聽傳送到 Presence 頻道的事件：

```
Echo.join(`chat.${roomId}`)
    .here(/* ... */)
    .joining(/* ... */)
    .leaving(/* ... */)
    .listen('NewMessage', (e) => {
        // ...
    });
```

20.10 模型廣播

> **警告** 在閱讀有關模型廣播的以下文件之前，我們建議您熟悉 Laravel 模型廣播服務的一般概念以及如何手動建立和監聽廣播事件。

當建立、更新或刪除應用程式的 Eloquent 模型時，通常會廣播事件。當然，這可以通過手動定義用於 Eloquent 模形狀態更改的自訂事件並將這些事件標記為 `ShouldBroadcast` 介面來輕鬆完成。

但是，如果您沒有在應用程式中使用這些事件進行任何其他用途，則為僅廣播它們的目的建立事件類可能會很麻煩。為解決這個問題，Laravel 允許您指示一個 Eloquent 模型應自動廣播其狀態更改。

開始之前，你的 Eloquent 模型應該使用 `Illuminate\Database\Eloquent\BroadcastsEvents` trait。此外，模型應該定義一個 `broadcastOn` 方法，該方法將返回一個陣列，該陣列包含模型事件應該廣播到的頻道：

```
<?php

namespace App\Models;

use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Database\Eloquent\BroadcastsEvents;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Post extends Model
{
```

```
    use BroadcastsEvents, HasFactory;

    /**
     * 獲取發帖使用者
     */
    public function user(): BelongsTo
    {
        return $this->belongsTo(User::class);
    }

    /**
     * 獲取模型事件應該廣播到的頻道
     *
     * @return array<int, \Illuminate\Broadcasting\Channel|\Illuminate\Database\
Eloquent\Model>
     */
    public function broadcastOn(string $event): array
    {
        return [$this, $this->user];
    }
}
```

一旦你的模型包含了這個 trait 並定義了它的廣播頻道，當模型實例被建立、更新、刪除、移到回收站或還原時，它將自動開始廣播事件。

另外，你可能已經注意到 `broadcastOn` 方法接收一個字串`$event` 參數。這個參數包含了在模型上發生的事件類型，將具有 `created`、`updated`、`deleted`、`trashed` 或 `restored` 的值。通過檢查這個變數的值，你可以確定模型在特定事件上應該廣播到哪些頻道（如果有）：

```
/**
 * 獲取模型事件應該廣播到的頻道
 *
 * @return array<string, array<int, \Illuminate\Broadcasting\Channel|\Illuminate\
Database\Eloquent\Model>>
 */
public function broadcastOn(string $event): array
{
    return match ($event) {
        'deleted' => [],
        default => [$this, $this->user],
    };
}
```

20.10.1.1 自訂模型廣播事件建立

有時候，您可能希望自訂 Laravel 建立底層模型廣播事件的方式。您可以通過在您的 Eloquent 模型上定義一個 `newBroadcastableEvent` 方法來實現。這個方法應該返回一個 `Illuminate\Database\Eloquent\BroadcastableModelEventOccurred` 實例：

```
use Illuminate\Database\Eloquent\BroadcastableModelEventOccurred;

/**
 * 為模型建立一個新的可廣播模型事件。
 */
protected function newBroadcastableEvent(string $event): BroadcastableModelEventOccurred
{
    return (new BroadcastableModelEventOccurred(
        $this, $event
    ))->dontBroadcastToCurrentUser();
}
```

20.10.2 模型廣播約定

20.10.2.1 頻道約定

您可能已經注意到，在上面的模型示例中，`broadcastOn` 方法沒有返回 `Channel` 實例。相反，它直接返回了 Eloquent 模型。如果您的模型的 `broadcastOn` 方法返回了 Eloquent 模型實例（或者包含在方法返回的陣列中），Laravel 將自動使用模型的類名和主鍵識別碼作為頻道名稱為模型實例實例化一個私有頻道實例。

因此，`App\Models\User` 模型的 `id` 為 `1` 將被轉換為一個名稱為 `App.Models.User.1` 的 `Illuminate\Broadcasting\PrivateChannel` 實例。當然，除了從模型的 `broadcastOn` 方法返回 Eloquent 模型實例之外，您還可以返回完整的 `Channel` 實例，以完全控制模型的頻道名稱：

```
use Illuminate\Broadcasting\PrivateChannel;

/**
 * 獲取模型事件應該廣播到的頻道。
 *
 * @return array<int, \Illuminate\Broadcasting\Channel>
 */
public function broadcastOn(string $event): array
{
    return [
        new PrivateChannel('user.'.$this->id)
    ];
}
```

如果您打算從模型的 `broadcastOn` 方法中明確返回一個頻道實例，您可以將一個 Eloquent 模型實例傳遞給頻道的建構函式。這樣做時，Laravel 將使用上面討論的模型頻道約定將 Eloquent 模型轉換為頻道名稱字串：

```
return [new Channel($this->user)];
```

如果您需要確定模型的頻道名稱，可以在任何模型實例上呼叫 `broadcastChannel` 方法。例如，對於一個 `App\Models\User` 模型，它的 `id` 為 `1`，這個方法將返回字串 `App.Models.User.1`：

```
$user->broadcastChannel()
```

20.10.2.2 事件約定

由於模型廣播事件與應用程式的 `App\Events` 目錄中的“實際”事件沒有關聯，它們會根據約定分配名稱和負載。 Laravel 的約定是使用模型的類名（不包括命名空間）和觸發廣播的模型事件的名稱來廣播事件。

例如，對 `App\Models\Post` 模型進行更新會將事件廣播到您的客戶端應用程式中，名稱為 `PostUpdated`，負載如下：

```
{
    "model": {
        "id": 1,
        "title": "My first post"
        ...
    },
    ...
    "socket": "someSocketId",
}
```

刪除 `App\Models\User` 模型將廣播名為 `UserDeleted` 的事件。

如果需要，您可以通過在模型中新增 `broadcastAs` 和 `broadcastWith` 方法來定義自訂廣播名稱和負載。這些方法接收正在發生的模型事件/操作的名稱，允許您為每個模型操作自訂事件的名稱和負載。如果從 `broadcastAs` 方法返回 `null`，則 Laravel 將在廣播事件時使用上述討論的模型廣播事件名稱約定：

```
/**
 * 模型事件的廣播名稱。
```

```
     */
    public function broadcastAs(string $event): string|null
    {
        return match ($event) {
            'created' => 'post.created',
            default => null,
        };
    }

    /**
     * 獲取要廣播到模型的資料。
     *
     * @return array<string, mixed>
     */
    public function broadcastWith(string $event): array
    {
        return match ($event) {
            'created' => ['title' => $this->title],
            default => ['model' => $this],
        };
    }
```

20.10.3 監聽模型廣播

一旦您將 `BroadcastsEvents` trait 新增到您的模型中並定義了模型的 `broadcastOn` 方法，您就可以開始在客戶端應用程式中監聽廣播的模型事件。在開始之前，您可能希望查閱完整的事件監聽文件。

首先，使用 `private` 方法獲取一個通道實例，然後呼叫 `listen` 方法來監聽指定的事件。通常，傳遞給 `private` 方法的通道名稱應該對應於 Laravel 的模型廣播規則。

獲取通道實例後，您可以使用 `listen` 方法來監聽特定事件。由於模型廣播事件與應用程式的 `App\Events` 目錄中的”實際”事件不相關聯，因此必須在事件名稱前加上 `.` 以表示它不屬於特定的命名空間。每個模型廣播事件都有一個 `model` 屬性，其中包含模型的所有可廣播屬性：

```
Echo.private(`App.Models.User.${this.user.id}`)
    .listen('.PostUpdated', (e) => {
        console.log(e.model);
    });
```

20.11 客戶端事件

> **注意** 當使用 Pusher Channels 時，您必須在應用程式儀表板的”應用程式設定”部分中啟用”客戶端事件”選項，以便傳送客戶端事件。

有時您可能希望將事件廣播給其他連接的客戶端，而根本不會觸發您的 Laravel 應用程式。這對於諸如”正在輸入”通知非常有用，其中您希望嚮應用程序的使用者通知另一個使用者正在給定螢幕上輸入消息。

要廣播客戶端事件，你可以使用 Echo 的 `whisper` 方法：

```
Echo.private(`chat.${roomId}`)
    .whisper('typing', {
        name: this.user.name
    });
```

要監聽客戶端事件，你可以使用 `listenForWhisper` 方法：

```
Echo.private(`chat.${roomId}`)
    .listenForWhisper('typing', (e) => {
        console.log(e.name);
    });
```

20.12 通知

通過將事件廣播與 notifications 配對，你的 JavaScript 應用程式可以在新通知發生時接收它們，而無需刷新頁面。在開始之前，請務必閱讀有關使用 廣播通知頻道 的文件。

一旦你組態了一個使用廣播頻道的通知，你就可以使用 Echo 的 `notification` 方法來監聽廣播事件。請記住，通道名稱應與接收通知的實體的類名稱匹配：

```
Echo.private(`App.Models.User.${userId}`)
    .notification((notification) => {
        console.log(notification.type);
    });
```

在這個例子中，所有通過 `broadcast` 通道傳送到 `App\Models\User` 實例的通知都會被回呼接收。`App.Models.User.{id}` 頻道的頻道授權回呼包含在 Laravel 框架附帶的默認 `BroadcastServiceProvider` 中。

21 快取系統

21.1 簡介

在某些應用中，一些查詢資料或處理任務的操作會在某段時間裡短時間內大量進行，或是一個操作花費好幾秒鐘。當出現這種情況時，通常會將檢索到的資料快取起來，從而為後面請求同一資料的請求迅速返回結果。這些快取資料通常會儲存在極快的儲存系統中，例如 Memcached 和 Redis。

Laravel 為各種快取後端提供了富有表現力且統一的 API，以便你利用它們極快的查詢資料來加快你的應用。

21.2 組態

快取組態檔案位於 `config/cache.php`。在這個檔案中，你可以指定應用默認使用哪個快取驅動。Laravel 支援的快取後端包括 Memcached、Redis、DynamoDB，以及現成的關係型資料庫。此外，還支援基於檔案的快取驅動，以及方便自動化測試的快取驅動 `array` 和 `null`。

快取組態檔案還包含檔案中記錄的各種其他選項，因此請務必閱讀這些選項。默認情況下，Laravel 組態為使用 `file` 快取驅動，它將序列化的快取對象儲存在伺服器的檔案系統中。對於較大的應用程式，建議你使用更強大的驅動，例如 Memcached 或 Redis。你甚至可以為同一個驅動組態多個快取組態。

21.2.1 驅動先決條件

21.2.1.1 Database

使用 `database` 快取驅動時，你需要設定一個表來包含快取項。你將在下表中找到 `Schema` 聲明的示例：

```
Schema::create('cache', function (Blueprint $table) {
    $table->string('key')->unique();
    $table->text('value');
    $table->integer('expiration');
});
```

注意 你還可以使用 `php artisan cache:table` Artisan 命令生成具有適當模式的遷移。

21.2.1.2 Memcached

使用 Memcached 驅動程式需要安裝 Memcached PECL 包。你可以在 `config/cache.php` 組態檔案中列出所有的 Memcached 伺服器。該檔案已經包含一個 `memcached.servers` 來幫助你入門：

```
'memcached' => [
    'servers' => [
        [
            'host' => env('MEMCACHED_HOST', '127.0.0.1'),
            'port' => env('MEMCACHED_PORT', 11211),
            'weight' => 100,
        ],
    ],
],
```

如果需要，你可以將 `host` 選項設定為 UNIX socket path。如果這樣做，`port` 選項應設定為 `0`：

```
'memcached' => [
```

```
    [
        'host' => '/var/run/memcached/memcached.sock',
        'port' => 0,
        'weight' => 100
    ],
],
```

21.2.1.3 Redis

在將 Redis 快取與 Laravel 一起使用之前，您需要通過 PECL 安裝 PhpRedis PHP 擴展或通過 Composer 安裝 `predis/predis` 包（~1.0）。Laravel Sail 已經包含了這個擴展。另外，Laravel 官方部署平台如 Laravel Forge 和 Laravel Vapor 也默認安裝了 PhpRedis 擴展。

有關組態 Redis 的更多資訊，請參閱其 Laravel documentation page.

21.2.1.4 DynamoDB

在使用 DynamoDB 快取驅動程式之前，您必須建立一個 DynamoDB 表來儲存所有快取的資料。通常，此表應命名為 `cache`。但是，您應該根據應用程式的快取組態檔案中的 `stores.dynamodb.table` 組態值來命名表。

該表還應該有一個字串分區鍵，其名稱對應於應用程式的快取組態檔案中的 `stores.dynamodb.attributes.key` 組態項的值。 默認情況下，分區鍵應命名為 `key`。

21.3 快取用法

21.3.1 獲取快取實例

要獲取快取儲存實例，您可以使用 `Cache` 門面類，我們將在本文件中使用它。`Cache` 門面類提供了對 Laravel 快取底層實現的方便、簡單的訪問：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

class UserController extends Controller
{
    /**
     * Show a list of all users of the application.
     */
    public function index(): array
    {
        $value = Cache::get('key');

        return [
            // ...
        ];
    }
}
```

21.3.1.1 訪問多個快取儲存

使用 `Cache` 門面類, 您可以通過 `store` 方法訪問各種快取儲存。傳遞給 `store` 方法的鍵應該對應於 `cache` 組態檔案中的 `stores` 組態陣列中列出的儲存之一：

```
$value = Cache::store('file')->get('foo');

Cache::store('redis')->put('bar', 'baz', 600); // 10 Minutes
```

21.3.2 從快取中檢索項目

Cache 門面的 get 方法用於從快取中檢索項目。如果快取中不存在該項目，則將返回 null。如果您願意，您可以將第二個參數傳遞給 get 方法，指定您希望在項目不存在時返回的預設值：

```
$value = Cache::get('key');

$value = Cache::get('key', 'default');
```

您甚至可以將閉包作為預設值傳遞。如果指定的項在快取中不存在，則返回閉包的結果。傳遞閉包允許您推遲從資料庫或其他外部服務中檢索預設值：

```
$value = Cache::get('key', function () {
    return DB::table(/* ... */)->get();
});
```

21.3.2.1 檢查項目是否存在

has 方法可用於確定快取中是否存在項目。如果項目存在但其值為 null，此方法也將返回 false：

```
if (Cache::has('key')) {
    // ...
}
```

21.3.2.2 遞增 / 遞減值

increment 和 decrement 方法可用於調整快取中整數項的值。這兩種方法都接受一個可選的第二個參數，指示增加或減少項目值的數量：

```
Cache::increment('key');
Cache::increment('key', $amount);
Cache::decrement('key');
Cache::decrement('key', $amount);
```

21.3.2.3 檢索和儲存

有時你可能希望從快取中檢索一個項目，但如果請求的項目不存在，則儲存一個預設值。 例如， 你可能希望從快取中檢索所有使用者，如果使用者不存在，則從資料庫中檢索並將它們新增到快取中。 你可以使用 Cache::remember 方法執行此操作：

```
$value = Cache::remember('users', $seconds, function () {
    return DB::table('users')->get();
});
```

如果該項不存在於快取中，將執行傳遞給 remember 方法的閉包，並將其結果放入快取中。

你可以使用 rememberForever 方法從快取中檢索一個項目，如果它不存在則永久儲存它：

```
$value = Cache::rememberForever('users', function () {
    return DB::table('users')->get();
});
```

21.3.2.4 檢索和刪除

如果你需要從快取中檢索一項後並刪除該項，你可以使用 pull 方法。 與 get 方法一樣，如果該項不存在於快取中，將返回 null：

```
$value = Cache::pull('key');
```

21.3.3 在快取中儲存項目

你可以使用 Cache Facade 上的 put 方法將項目儲存在快取中：

```
Cache::put('key', 'value', $seconds = 10);
```

如果儲存時間沒有傳遞給 put 方法，則該項目將無限期儲存：

```
Cache::put('key', 'value');
```

除了將秒數作為整數傳遞之外，你還可以傳遞一個代表快取項所需過期時間的 DateTime 實例：

```
Cache::put('key', 'value', now()->addMinutes(10));
```

21.3.3.1 如果不存在則儲存

add 方法只會將快取儲存中不存在的項目新增到快取中。如果項目實際新增到快取中，該方法將返回 true。否則，該方法將返回 false。 add 方法是一個原子操作：

```
Cache::add('key', 'value', $seconds);
```

21.3.3.2 永久儲存

forever 方法可用於將項目永久儲存在快取中。由於這些項目不會過期，因此必須使用 forget 方法手動將它們從快取中刪除：

```
Cache::forever('key', 'value');
```

> 注意：如果您使用的是 Memcached 驅動程式，則當快取達到其大小限制時，可能會刪除「永久」儲存的項目。

21.3.4 從快取中刪除項目

您可以使用 forget 方法從快取中刪除項目：

```
Cache::forget('key');
```

您還可以通過提供零或負數的過期秒數來刪除項目：

```
Cache::put('key', 'value', 0);

Cache::put('key', 'value', -5);
```

您可以使用 flush 方法清除整個快取：

```
Cache::flush();
```

> 注意：刷新快取不會考慮您組態的快取「前綴」，並且會從快取中刪除所有條目。在清除由其他應用程式共享的快取時，請考慮到這一點。

21.3.5 快取助手函數

除了使用 Cache 門面之外，您還可以使用全域 cache 函數通過快取檢索和儲存資料。當使用單個字串參數呼叫 cache 函數時，它將返回給定鍵的值：

```
$value = cache('key');
```

如果您向函數提供鍵 / 值對陣列和過期時間，它將在指定的持續時間內將值儲存在快取中：

```
cache(['key' => 'value'], $seconds);

cache(['key' => 'value'], now()->addMinutes(10));
```

當不帶任何參數呼叫 cache 函數時，它會返回 Illuminate 實現的實例，允許您呼叫其他快取方法：

```
cache()->remember('users', $seconds, function () {
    return DB::table('users')->get();
});
```

技巧
在測試對全域 `cache` 函數的呼叫時，您可以使用 `Cache::shouldReceive` 方法，就像 testing the facade.

21.4 快取標籤

注意
使用 `file`, `dynamodb` 或 `database` 存驅動程式時不支援快取標記。 此外，當使用帶有 “永久” 儲存的快取的多個標籤時，使用諸如 “memcached” 之類的驅動程式會獲得最佳性能，它會自動清除陳舊的記錄。

21.4.1 儲存快取標籤

快取標籤允許您在快取中標記相關項目，然後刷新所有已分配給定標籤的快取值。您可以通過傳入標記名稱的有序陣列來訪問標記快取。例如，讓我們訪問一個標記的快取並將一個值 `put` 快取中：

```
Cache::tags(['people', 'artists'])->put('John', $john, $seconds);

Cache::tags(['people', 'authors'])->put('Anne', $anne, $seconds);
```

21.4.2 訪問快取標籤

要檢索標記的快取項，請將相同的有序標籤列表傳遞給 tags 方法，然後使用您要檢索的鍵呼叫 `get` 方法：

```
$john = Cache::tags(['people', 'artists'])->get('John');

$anne = Cache::tags(['people', 'authors'])->get('Anne');
```

21.4.3 刪除被標記的快取資料

你可以刷新所有分配了標籤或標籤列表的項目。 例如，此語句將刪除所有標記有 `people`, `authors` 或兩者的快取。因此，`Anne` 和 `John` 都將從快取中刪除：

```
Cache::tags(['people', 'authors'])->flush();
```

相反，此語句將僅刪除帶有 `authors` 標記的快取，因此將刪除 `Anne`，但不會刪除 `John`：

```
Cache::tags('authors')->flush();
```

21.4.4 清理過期的快取標記

注意：僅在使用 Redis 作為應用程式的快取驅動程式時，才需要清理過期的快取標記。

為了在使用 Redis 快取驅動程式時正確清理過期的快取標記，Laravel 的 Artisan 命令 `cache:prune-stale-tags` 應該被新增到 任務調度 中，在應用程式的 `App\Console\Kernel` 類裡：

```
$schedule->command('cache:prune-stale-tags')->hourly();
```

21.5 原子鎖

注意：要使用此功能，您的應用程式必須使用 `memcached`、`redis`、`dynamicodb`、`database`、`file` 或 `array` 快取驅動程式作為應用程式的默認快取驅動程式。 此外，所有伺服器都必須與同一中央快取伺服器通訊。

21.5.1 驅動程式先決條件

21.5.1.1 資料庫

使用“資料庫”快取驅動程式時，您需要設定一個表來包含應用程式的快取鎖。您將在下表中找到一個示例 `Schema` 聲明：

```
Schema::create('cache_locks', function (Blueprint $table) {
    $table->string('key')->primary();
    $table->string('owner');
    $table->integer('expiration');
});
```

21.5.2 管理鎖

原子鎖允許操作分佈式鎖而不用擔心競爭條件。例如，Laravel Forge 使用原子鎖來確保伺服器上一次只執行一個遠端任務。您可以使用 `Cache::lock` 方法建立和管理鎖：

```
use Illuminate\Support\Facades\Cache;

$lock = Cache::lock('foo', 10);

if ($lock->get()) {
    // Lock acquired for 10 seconds...

    $lock->release();
}
```

`get` 方法也接受一個閉包。閉包執行後，Laravel 會自動釋放鎖：

```
Cache::lock('foo', 10)->get(function () {
    // Lock acquired for 10 seconds and automatically released...
});
```

如果在您請求時鎖不可用，您可以指示 Laravel 等待指定的秒數。如果在指定的時間限制內無法獲取鎖，則會拋出 Illuminate：

```
use Illuminate\Contracts\Cache\LockTimeoutException;

$lock = Cache::lock('foo', 10);

try {
    $lock->block(5);

    // Lock acquired after waiting a maximum of 5 seconds...
} catch (LockTimeoutException $e) {
    // Unable to acquire lock...
} finally {
    $lock?->release();
}
```

上面的例子可以通過將閉包傳遞給 `block` 方法來簡化。當一個閉包被傳遞給這個方法時，Laravel 將嘗試在指定的秒數內獲取鎖，並在閉包執行後自動釋放鎖：

```
Cache::lock('foo', 10)->block(5, function () {
    // Lock acquired after waiting a maximum of 5 seconds...
});
```

21.5.3 跨處理程序管理鎖

有時，您可能希望在一個處理程序中獲取鎖並在另一個處理程序中釋放它。例如，您可能在 Web 請求期間獲取鎖，並希望在由該請求觸發的排隊作業結束時釋放鎖。在這種情況下，您應該將鎖的範疇 `owner token` 傳遞給排隊的作業，以便作業可以使用給定的令牌重新實例化鎖。

在下面的示例中，如果成功獲取鎖，我們將調度一個排隊的作業。 此外，我們將通過鎖的 `owner` 方法將鎖的所有者令牌傳遞給排隊的作業：

```
$podcast = Podcast::find($id);

$lock = Cache::lock('processing', 120);

if ($lock->get()) {
    ProcessPodcast::dispatch($podcast, $lock->owner());
}
```

在我們應用程式的 `ProcessPodcast` 作業中，我們可以使用所有者令牌恢復和釋放鎖：

```
Cache::restoreLock('processing', $this->owner)->release();
```

如果你想釋放一個鎖而不考慮它的當前所有者，你可以使用 `forceRelease` 方法：

```
Cache::lock('processing')->forceRelease();
```

21.6 新增自訂快取驅動

21.6.1 編寫驅動

要建立我們的自訂快取驅動程式，我們首先需要實現 `Illuminate\Contracts\Cache\Store` contract。 因此，MongoDB 快取實現可能如下所示：

```
<?php

namespace App\Extensions;

use Illuminate\Contracts\Cache\Store;

class MongoStore implements Store
{
    public function get($key) {}
    public function many(array $keys) {}
    public function put($key, $value, $seconds) {}
    public function putMany(array $values, $seconds) {}
    public function increment($key, $value = 1) {}
    public function decrement($key, $value = 1) {}
    public function forever($key, $value) {}
    public function forget($key) {}
    public function flush() {}
    public function getPrefix() {}
}
```

我們只需要使用 MongoDB 連接來實現這些方法中的每一個。有關如何實現這些方法的示例，請查看 Laravel 框架原始碼中的 `Illuminate\Cache\MemcachedStore`。 一旦我們的實現完成，我們可以通過呼叫 `Cache` 門面的 `extend` 方法來完成我們的自訂驅動程式註冊：

```
Cache::extend('mongo', function (Application $app) {
    return Cache::repository(new MongoStore);
```

```
});
```

技巧
如果你想知道將自訂快取驅動程式程式碼放在哪裡，可以在你的 app 目錄中建立一個 Extensions 命名空間。 但是請記住，Laravel 沒有嚴格的應用程式結構，你可以根據自己的喜好自由組織應用程式。

21.6.2 註冊驅動

要向 Laravel 註冊自訂快取驅動程式，我們將使用 Cache 門面的 extend 方法。 由於其他服務提供者可能會嘗試在他們的 boot 方法中讀取快取值，我們將在 booting 回呼中註冊我們的自訂驅動程式。 通過使用 booting 回呼，我們可以確保在應用程式的服務提供者呼叫 boot 方法之前但在所有服務提供者呼叫 register 方法之後註冊自訂驅動程式。 我們將在應用程式的 App\Providers\AppServiceProvider 類的 register 方法中註冊我們的 booting 回呼：

```
<?php

namespace App\Providers;

use App\Extensions\MongoStore;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\ServiceProvider;

class CacheServiceProvider extends ServiceProvider
{
    /**
     * 註冊任何應用程式服務。
     */
    public function register(): void
    {
        $this->app->booting(function () {
            Cache::extend('mongo', function (Application $app) {
                return Cache::repository(new MongoStore);
            });
        });
    }

    /**
     * 引導任何應用程式服務。
     */
    public function boot(): void
    {
        // ...
    }
}
```

傳遞給 extend 方法的第一個參數是驅動程式的名稱。這將對應於 config/cache.php 組態檔案中的 driver 選項。 第二個參數是一個閉包，它應該返回一個 Illuminate\Cache\Repository 實例。閉包將傳遞一個$app 實例，它是服務容器的一個實例。

註冊擴展程序後，將 config/cache.php 組態檔案的 driver 選項更新為擴展程序的名稱。

21.7 事件

要在每個快取操作上執行程式碼，你可以偵聽快取觸發的 events 。 通常，你應該將這些事件偵聽器放在應用程式的 App\Providers\EventServiceProvider 類中：

```
use App\Listeners\LogCacheHit;
```

```
use App\Listeners\LogCacheMissed;
use App\Listeners\LogKeyForgotten;
use App\Listeners\LogKeyWritten;
use Illuminate\Cache\Events\CacheHit;
use Illuminate\Cache\Events\CacheMissed;
use Illuminate\Cache\Events\KeyForgotten;
use Illuminate\Cache\Events\KeyWritten;

/**
 * 應用程式的事件偵聽器對應。
 *
 * @var array
 */
protected $listen = [
    CacheHit::class => [
        LogCacheHit::class,
    ],

    CacheMissed::class => [
        LogCacheMissed::class,
    ],

    KeyForgotten::class => [
        LogKeyForgotten::class,
    ],

    KeyWritten::class => [
        LogKeyWritten::class,
    ],
];
```

22 集合

22.1 介紹

`Illuminate\Support\Collection` 類為處理資料陣列提供了一個流暢、方便的包裝器。 例如，查看以下程式碼。 我們將使用 `collect` 助手從陣列中建立一個新的集合實例，對每個元素運行 `strtoupper` 函數，然後刪除所有空元素：

```
$collection = collect(['taylor', 'abigail', null])->map(function (string $name) {
    return strtoupper($name);
})->reject(function (string $name) {
    return empty($name);
});
```

如你所見，`Collection` 類允許你連結其方法以執行流暢的對應和減少底層陣列。一般來說，集合是不可變的，這意味著每個 `Collection` 方法都會返回一個全新的 `Collection` 實例。

22.1.1 建立集合

如上所述，`collect` 幫助器為給定陣列返回一個新的 `Illuminate\Support\Collection` 實例。因此，建立一個集合非常簡單：

```
$collection = collect([1, 2, 3]);
```

> **技巧：**Eloquent 查詢的結果總是作為 `Collection` 實例返回。

22.1.2 擴展集合

集合是「可宏化的」，它允許你在執行階段向 `Collection` 類新增其他方法。 `Illuminate\Support\Collection` 類的 `macro` 方法接受一個閉包，該閉包將在呼叫宏時執行。宏閉包可以通過 `$this` 訪問集合的其他方法，就像它是集合類的真實方法一樣。例如，以下程式碼在 `Collection` 類中新增了 `toUpper` 方法：

```
use Illuminate\Support\Collection;
use Illuminate\Support\Str;

Collection::macro('toUpper', function () {
    return $this->map(function (string $value) {
        return Str::upper($value);
    });
});

$collection = collect(['first', 'second']);

$upper = $collection->toUpper();

// ['FIRST', 'SECOND']
```

通常，你應該在服務提供者的 `boot` 方法中聲明集合宏。

22.1.2.1 宏參數

如有必要，可以定義接受其他參數的宏：

```
use Illuminate\Support\Collection;
```

```
use Illuminate\Support\Facades\Lang;

Collection::macro('toLocale', function (string $locale) {
    return $this->map(function (string $value) use ($locale) {
        return Lang::get($value, [], $locale);
    });
});

$collection = collect(['first', 'second']);

$translated = $collection->toLocale('es');
```

22.2 可用的方法

對於剩餘的大部分集合文件，我們將討論 `Collection` 類中可用的每個方法。請記住，所有這些方法都可以鏈式呼叫，以便流暢地操作底層陣列。此外，幾乎每個方法都會返回一個新的 `Collection` 實例，允許你在必要時保留集合的原始副本：

不列印可用的方法

22.3 Higher Order Messages

集合也提供對「高階消息傳遞」的支援，即集合常見操作的快捷方式。支援高階消息傳遞的集合方法有：`average`、`avg`、`contains`、`each`、`every`、`filter`、`first`、`flatMap`、`groupBy`、`keyBy`、`map`、`max`、`min`、`partition`、`reject`、`skipUntil`、`skipWhile`、`some`、`sortBy`、`sortByDesc`、`sum`、`takeUntil`、`takeWhile` 和 `unique`。

每個高階消息都可以作為集合實例上的動態屬性進行訪問。例如，讓我們使用 `each` 高階消息來呼叫集合中每個對象的方法：

```
use App\Models\User;

$users = User::where('votes', '>', 500)->get();

$users->each->markAsVip();
```

同樣，我們可以使用 `sum` 高階消息來收集使用者集合的「votes」總數：

```
$users = User::where('group', 'Development')->get();

return $users->sum->votes;
```

22.4 惰性集合

22.4.1 介紹

> **注意：**在進一步瞭解 Laravel 的惰性集合之前，花點時間熟悉一下 PHP 生成器.

為了補充已經強大的 `Collection` 類，`LazyCollection` 類利用 PHP 的 generators 允許你使用非常 大型資料集，同時保持較低的記憶體使用率。

例如，假設你的應用程式需要處理數 GB 的記錄檔，同時利用 Laravel 的集合方法來解析日誌。可以使用惰性集合在給定時間僅將檔案的一小部分保留在記憶體中，而不是一次將整個檔案讀入記憶體：

```
use App\Models\LogEntry;
use Illuminate\Support\LazyCollection;
```

```
LazyCollection::make(function () {
    $handle = fopen('log.txt', 'r');

    while (($line = fgets($handle)) !== false) {
        yield $line;
    }
})->chunk(4)->map(function (array $lines) {
    return LogEntry::fromLines($lines);
})->each(function (LogEntry $logEntry) {
    // Process the log entry...
});
```

或者，假設你需要遍歷 10,000 個 Eloquent 模型。使用傳統 Laravel 集合時，所有 10,000 個 Eloquent 模型必須同時載入到記憶體中：

```
use App\Models\User;

$users = User::all()->filter(function (User $user) {
    return $user->id > 500;
});
```

但是，查詢建構器的 `cursor` 方法返回一個 `LazyCollection` 實例。這允許你仍然只對資料庫運行一個查詢，而且一次只在記憶體中載入一個 Eloquent 模型。在這個例子中，`filter` 回呼在我們實際單獨遍歷每個使用者之前不會執行，從而可以大幅減少記憶體使用量：

```
use App\Models\User;

$users = User::cursor()->filter(function (User $user) {
    return $user->id > 500;
});

foreach ($users as $user) {
    echo $user->id;
}
```

22.4.2 建立惰性集合

要建立惰性集合實例，你應該將 PHP 生成器函數傳遞給集合的 `make` 方法：

```
use Illuminate\Support\LazyCollection;

LazyCollection::make(function () {
    $handle = fopen('log.txt', 'r');

    while (($line = fgets($handle)) !== false) {
        yield $line;
    }
});
```

22.4.3 列舉契約

`Collection` 類上幾乎所有可用的方法也可以在 `LazyCollection` 類上使用。這兩個類都實現了 `Illuminate\Support\Enumerable` 契約，它定義了以下方法：

不列印 LazyCollection 方法清單

> **注意：**改變集合的方法（例如 `shift`、`pop`、`prepend` 等）在 `LazyCollection` 類中**不**可用。

22.4.4 惰性集合方法

除了在 `Enumerable` 契約中定義的方法外， `LazyCollection` 類還包含以下方法：

22.4.4.1 takeUntilTimeout()

takeUntilTimeout 方法返回新的惰性集合，它會在給定時間前去列舉集合值，之後集合將停止列舉：

```
$lazyCollection = LazyCollection::times(INF)
     ->takeUntilTimeout(now()->addMinute());

$lazyCollection->each(function (int $number) {
     dump($number);

     sleep(1);
});

// 1
// 2
// ...
// 58
// 59
```

為了具體闡述此方法，請設想一個使用游標從資料庫提交發票的例子。你可以定義一個 工作排程，它每十五分鐘執行一次，並且只執行發票提交操作的最大時間是 14 分鐘：

```
use App\Models\Invoice;
use Illuminate\Support\Carbon;

Invoice::pending()->cursor()
     ->takeUntilTimeout(
          Carbon::createFromTimestamp(LARAVEL_START)->add(14, 'minutes')
     )
     ->each(fn (Invoice $invoice) => $invoice->submit());
```

22.4.4.2 tapEach()

當 each 方法為集合中每一個元素呼叫給定回呼時， tapEach 方法僅呼叫給定回呼，因為這些元素正在逐個從列表中拉出：

```
// 沒有任何輸出
$lazyCollection = LazyCollection::times(INF)->tapEach(function (int $value) {
     dump($value);
});

// 列印出三條資料
$array = $lazyCollection->take(3)->all();

// 1
// 2
// 3
```

22.4.4.3 remember()

remember 方法返回一個新的惰性集合，這個集合已經記住（快取）已列舉的所有值，當再次列舉該集合時不會獲取它們：

```
// 沒執行任何查詢
$users = User::cursor()->remember();

//  執行了查詢操作
// The first 5 users are hydrated from the database...
$users->take(5)->all();

// 前 5 個使用者資料從快取中獲取
// The rest are hydrated from the database...
$users->take(20)->all();
```

23 契約 Contract

23.1 簡介

Laravel 的「契約（Contract）」是一組介面，它們定義由框架提供的核心服務。例如，`illuste\Contracts\Queue\Queue` Contract 定義了佇列所需的方法，而 `illuste\Contracts\Mail\Mailer` Contract 定義了傳送郵件所需的方法。

每個契約都有由框架提供的相應實現。例如，Laravel 提供了一個支援各種驅動的佇列實現，還有一個由 SwiftMailer 提供支援的郵件程序實現等等。

所有的 Laravel Contract 都存在於它們各自的 GitHub 倉庫。這為所有可用的契約提供了一個快速的參考點，以及一個可以被包開發人員使用的獨立的包。

23.1.1 Contract 對比 Facade

Laravel 的 Facade 和輔助函數提供了一種利用 Laravel 服務的簡單方法，無需類型提示並可以從服務容器中解析 Contract。在大多數情況下，每個 Facade 都有一個等效的 Contract。

和 Facade（不需要在建構函式中引入）不同，Contract 允許你為類定義顯式依賴關係。一些開發者更喜歡以這種方式顯式定義其依賴項，所以更喜歡使用 Contract，而其他開發者則享受 Facade 帶來的便利。**通常，大多數應用都可以在開發過程中使用 Facade**。

23.2 何時使用 Contract

使用 Contract 或 Facades 取決於個人喜好和開發團隊的喜好。Contract 和 Facade 均可用於建立功能強大且經過良好測試的 Laravel 應用。Contract 和 Facade 並不是一道單選題，你可以在同一個應用內同時使用 Contract 和 Facade。只要聚焦在類的職責應該單一上，你會發現 Contract 和 Facade 的實際差異其實很小。

通常情況下，大部分使用 Facade 的應用都不會在開發中遇到問題。但如果你在建立一個可以由多個 PHP 框架使用的擴展包，你可能會希望使用 `illuminate/contracts` 擴展包來定義該包和 Laravel 整合，而不需要引入完整的 Laravel 實現（不需要在 `composer.json` 中具體顯式引入 Laravel 框架來實現）。

23.3 如何使用 Contract

那麼，如何實現契約呢？它其實很簡單。

Laravel 中的許多類都是通過 服務容器 解析的，包括 controller 、事件偵聽器、中介軟體、佇列任務，甚至路由閉包。因此，要實現契約，你只需要在被解析的類的建構函式中「類型提示」介面。

例如，看看下面的這個事件監聽器：

```
<?php

namespace App\Listeners;

use App\Events\OrderWasPlaced;
use App\Models\User;
use Illuminate\Contracts\Redis\Factory;
```

```
class CacheOrderInformation
{
    /**
     * 建立一個新的事件監聽器實例
     */
    public function __construct(
        protected Factory $redis,
    ) {}

    /**
     * 處理該事件。
     */
    public function handle(OrderWasPlaced $event): void
    {
        // ...
    }
}
```

當解析事件監聽器時，服務容器將讀取建構函式上的類型提示，並注入適當的值。 要瞭解更多有關在服務容器中註冊內容的資訊，請查看 其文件。

23.4 Contract 參考

下表提供了所有 Laravel Contract 及對應的 Facade 的快速參考：

不列印表格

24 事件系統

24.1 介紹

Laravel 的事件系統提供了一個簡單的觀察者模式的實現，允許你能夠訂閱和監聽在你的應用中的發生的各種事件。事件類一般來說儲存在 `app/Events` 目錄，監聽者的類儲存在 `app/Listeners` 目錄。不要擔心在你的應用中沒有看到這兩個目錄，因為通過 Artisan 命令列來建立事件和監聽者的時候目錄會同時被建立。

事件系統可以作為一個非常棒的方式來解耦你的系統的方方面面，因為一個事件可以有多個完全不相關的監聽者。例如，你希望每當有訂單發出的時候都給你傳送一個 Slack 通知。你大可不必將你的處理訂單的程式碼和傳送 slack 消息的程式碼放在一起，你只需要觸發一個 App 事件，然後事件監聽者可以收到這個事件然後傳送 slack 通知

24.2 註冊事件和監聽器

在系統的服務提供者 `App\Providers\EventServiceProvider` 中提供了一個簡單的方式來註冊你所有的事件監聽者。屬性 `listen` 包含所有的事件 (作為鍵) 和對應的監聽器 (值)。你可以新增任意多系統需要的監聽器在這個陣列中，讓我們新增一個 `OrderShipped` 事件：

```
use App\Events\OrderShipped;
use App\Listeners\SendShipmentNotification;

/**
 * 系統中的事件和監聽器的對應關係。
 *
 * @var array
 */
protected $listen = [
    OrderShipped::class => [
        SendShipmentNotification::class,
    ],
];
```

注意 可以使用 `event:list` 命令顯示應用程式

24.2.1 生成事件和監聽器

當然，為每個事件和監聽器手動建立檔案是很麻煩的。相反，將監聽器和事件新增到 `EventServiceProvider` 並使用 `event:generate` Artisan 命令。此命令將生成 `EventServiceProvider` 中列出的、尚不存在的任何事件或偵聽器：

```
php artisan event:generate
```

或者，你可以使用 `make:event` 以及 `make:listener` 用於生成單個事件和監聽器的 Artisan 命令：

```
php artisan make:event PodcastProcessed

php artisan make:listener SendPodcastNotification --event=PodcastProcessed
```

24.2.2 手動註冊事件

通常，事件應該通過 `EventServiceProvider` `$listen` 陣列註冊；但是，你也可以在 `EventServiceProvider` 的 `boot` 方法中手動註冊基於類或閉包的事件監聽器：

```
use App\Events\PodcastProcessed;
use App\Listeners\SendPodcastNotification;
use Illuminate\Support\Facades\Event;

/**
 * 註冊任意的其他事件和監聽器。
 */
public function boot(): void
{
    Event::listen(
        PodcastProcessed::class,
        [SendPodcastNotification::class, 'handle']
    );

    Event::listen(function (PodcastProcessed $event) {
        // ...
    });
}
```

24.2.2.1 可排隊匿名事件監聽器

手動註冊基於閉包的事件監聽器時，可以將監聽器閉包包裝在 `Illuminate\Events\queueable` 函數中，以指示 Laravel 使用 佇列 執行偵聽器：

```
use App\Events\PodcastProcessed;
use function Illuminate\Events\queueable;
use Illuminate\Support\Facades\Event;

/**
 * 註冊任意的其他事件和監聽器。
 */
public function boot(): void
{
    Event::listen(queueable(function (PodcastProcessed $event) {
        // ...
    }));
}
```

與佇列任務一樣，可以使用 `onConnection`、`onQueue` 和 `delay` 方法自訂佇列監聽器的執行：

```
Event::listen(queueable(function (PodcastProcessed $event) {
    // ...
})->onConnection('redis')->onQueue('podcasts')->delay(now()->addSeconds(10)));
```

如果你想處理匿名佇列監聽器失敗，你可以在定義 `queueable` 監聽器時為 `catch` 方法提供一個閉包。這個閉包將接收導致監聽器失敗的事件實例和 `Throwable` 實例：

```
use App\Events\PodcastProcessed;
use function Illuminate\Events\queueable;
use Illuminate\Support\Facades\Event;
use Throwable;

Event::listen(queueable(function (PodcastProcessed $event) {
    // ...
})->catch(function (PodcastProcessed $event, Throwable $e) {
    // 佇列監聽器失敗了
}));
```

24.2.2.2 萬用字元事件監聽器

你甚至可以使用 * 作為萬用字元參數註冊監聽器，這允許你在同一個監聽器上捕獲多個事件。萬用字元監聽器接收事件名作為其第一個參數，整個事件資料陣列作為其第二個參數：

```
Event::listen('event.*', function (string $eventName, array $data) {
    // ...
});
```

24.2.3 事件的發現

你可以啟用自動事件發現，而不是在 `EventServiceProvider` 的 `$listen` 陣列中手動註冊事件和偵聽器。當事件發現啟用，Laravel 將自動發現和註冊你的事件和監聽器掃描你的應用程式的 `Listeners` 目錄。此外，在 `EventServiceProvider` 中列出的任何顯式定義的事件仍將被註冊。

Laravel 通過使用 PHP 的反射服務掃描監聽器類來尋找事件監聽器。當 Laravel 發現任何以 `handle` 或 `__invoke` 開頭的監聽器類方法時，Laravel 會將這些方法註冊為該方法簽名中類型暗示的事件的事件監聽器：

```
use App\Events\PodcastProcessed;

class SendPodcastNotification
{
    /**
     * 處理給定的事件
     */
    public function handle(PodcastProcessed $event): void
    {
        // ...
    }
}
```

事件發現在默認情況下是停用的，但你可以通過重寫應用程式的 `EventServiceProvider` 的 `shouldDiscoverEvents` 方法來啟用它：

```
/**
 * 確定是否應用自動發現事件和監聽器。
 */
public function shouldDiscoverEvents(): bool
{
    return true;
}
```

默認情況下，應用程式 `app/listeners` 目錄中的所有監聽器都將被掃描。如果你想要定義更多的目錄來掃描，你可以重寫 `EventServiceProvider` 中的 `discoverEventsWithin` 方法：

```
/**
 * 獲取應用於發現事件的監聽器目錄。
 *
 * @return array<int, string>
 */
protected function discoverEventsWithin(): array
{
    return [
        $this->app->path('Listeners'),
    ];
}
```

24.2.3.1 生產中的事件發現

在生產環境中，框架在每個請求上掃描所有監聽器的效率並不高。因此，在你的部署過程中，你應該運行 `event:cache` Artisan 命令來快取你的應用程式的所有事件和監聽器清單。框架將使用該清單來加速事件註冊過程。`event:clear` 命令可以用來銷毀快取。

24.3 定義事件

事件類本質上是一個資料容器，它保存與事件相關的資訊。例如，讓我們假設一個 App\Events\OrderShipped 事件接收到一個 Eloquent ORM 對象：

```
<?php

namespace App\Events;

use App\Models\Order;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Queue\SerializesModels;

class OrderShipped
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    /**
     * 建立一個新的事件實例。
     */
    public function __construct(
        public Order $order,
    ) {}
}
```

如你所見，這個事件類不包含邏輯。它是一個被購買的 App\Models\Order 實例容器。 如果事件對像是使用 PHP 的 SerializesModels 函數序列化的，事件使用的 SerializesModels trait 將會優雅地序列化任何 Eloquent 模型，比如在使用 佇列偵聽器。

24.4 定義監聽器

接下來，讓我們看一下示例事件的監聽器。事件監聽器在其 handle 方法中接收事件實例。Artisan 命令 event:generate 和 make:listener 會自動匯入正確的事件類，並在 handle 方法上對事件進行類型提示。在 handle 方法中，你可以執行任何必要的操作來響應事件：

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;

class SendShipmentNotification
{
    /**
     * 建立事件監聽器
     */
    public function __construct()
    {
        // ...
    }

    /**
     * 處理事件
     */
    public function handle(OrderShipped $event): void
    {
        // 使用 $event->order 來訪問訂單 ...
    }
}
```

技巧 事件監聽器還可以在建構函式中加入任何依賴關係的類型提示。所有的事件監聽器都是通過

Laravel 的 服務容器 解析的，因此所有的依賴都將會被自動注入。

24.4.1.1 停止事件傳播

有時，你可能希望停止將事件傳播到其他監聽器。你可以通過從監聽器的 `handle` 方法中返回 `false` 來做到這一點。

24.5 佇列事件監聽器

如果你的監聽器要執行一個緩慢的任務，如傳送電子郵件或進行 HTTP 請求，那麼佇列化監聽器就很有用了。在使用佇列監聽器之前，請確保 組態你的佇列 並在你的伺服器或本地開發環境中啟動一個佇列 worker。

要指定監聽器啟動佇列，請將 `ShouldQueue` 介面新增到監聽器類。 由 Artisan 命令 `event:generate` 和 `make:listener` 生成的監聽器已經將此介面匯入當前命名空間，因此你可以直接使用：

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    // ...
}
```

就是這樣！ 現在，當此監聽器處理的事件被調度時，監聽器將使用 Laravel 的 佇列系統 自動由事件調度器排隊。 如果監聽器被佇列執行時沒有拋出異常，佇列中的任務處理完成後會自動刪除。

24.5.1.1 自訂佇列連接和佇列名稱

如果你想自訂事件監聽器的佇列連接、佇列名稱或佇列延遲時間，可以在監聽器類上定義 `$connection`、`$queue` 或 `$delay` 屬性：

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    /**
     * 任務傳送到的連接的名稱。
     *
     * @var string|null
     */
    public $connection = 'sqs';

    /**
     * 任務傳送到的佇列的名稱。
     *
     * @var string|null
     */
    public $queue = 'listeners';

    /**
     * 處理作業前的時間（秒）。
```

```
     *
     * @var int
     */
    public $delay = 60;
}
```

如果你想在執行階段定義監聽器的佇列連接或佇列名稱，可以在監聽器上定義 `viaConnection` 或 `viaQueue` 方法：

```
/**
 * 獲取偵聽器的佇列連接的名稱。
 */
public function viaConnection(): string
{
    return 'sqs';
}

/**
 * 獲取偵聽器佇列的名稱。
 */
public function viaQueue(): string
{
    return 'listeners';
}
```

24.5.1.2 有條件地佇列監聽器

有時，你可能需要根據一些僅在執行階段可用的資料來確定是否應將偵聽器排隊。 為此，可以將「shouldQueue」方法新增到偵聽器以確定是否應將偵聽器排隊。 如果 `shouldQueue` 方法返回 `false`，監聽器將不會被執行：

```
<?php

namespace App\Listeners;

use App\Events\OrderCreated;
use Illuminate\Contracts\Queue\ShouldQueue;

class RewardGiftCard implements ShouldQueue
{
    /**
     * 獎勵客戶一張禮品卡。
     */
    public function handle(OrderCreated $event): void
    {
        // ...
    }

    /**
     * 確定偵聽器是否應排隊。
     */
    public function shouldQueue(OrderCreated $event): bool
    {
        return $event->order->subtotal >= 5000;
    }
}
```

24.5.2 手動與佇列互動

如果你需要手動訪問偵聽器的底層佇列作業的 delete 和 release 方法，可以使用 `Illuminate\Queue\InteractsWithQueue` 特性來實現。 這個 trait 默認匯入生成的偵聽器並提供對這些方法的訪問：

```
<?php
```

```
namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    /**
     * Handle the event.
     */
    public function handle(OrderShipped $event): void
    {
        if (true) {
            $this->release(30);
        }
    }
}
```

24.5.3 佇列事件監聽器和資料庫事務

當排隊的偵聽器在資料庫事務中被分派時，它們可能在資料庫事務提交之前由佇列處理。 發生這種情況時，在資料庫事務期間對模型或資料庫記錄所做的任何更新可能尚未反映在資料庫中。 此外，在事務中建立的任何模型或資料庫記錄可能不存在於資料庫中。 如果你的偵聽器依賴於這些模型，則在處理調度排隊偵聽器的作業時可能會發生意外錯誤。

如果你的佇列連接的 `after_commit` 組態選項設定為 `false`，你仍然可以通過在偵聽器類上定義 `$afterCommit` 屬性來指示在提交所有打開的資料庫事務後應該調度特定的排隊偵聽器：

```
<?php

namespace App\Listeners;

use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    public $afterCommit = true;
}
```

注意 要瞭解有關解決這些問題的更多資訊，請查看有關佇列作業和資料庫事務 的文件。

24.5.4 處理失敗的佇列

有時佇列的事件監聽器可能會失敗。如果排隊的監聽器超過了佇列工作者定義的最大嘗試次數，則將對監聽器呼叫 `failed` 方法。`failed` 方法接收導致失敗的事件實例和 `Throwable`：

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;
use Throwable;

class SendShipmentNotification implements ShouldQueue
{
```

```
    use InteractsWithQueue;

    /**
     * 事件處理。
     */
    public function handle(OrderShipped $event): void
    {
        // ...
    }

    /**
     * 處理失敗任務。
     */
    public function failed(OrderShipped $event, Throwable $exception): void
    {
        // ...
    }
}
```

24.5.4.1 指定佇列監聽器的最大嘗試次數

如果佇列中的某個監聽器遇到錯誤，你可能不希望它無限期地重試。因此，Laravel 提供了各種方法來指定監聽器的嘗試次數或嘗試時間。

你可以在監聽器類上定義 `$tries` 屬性，以指定監聽器在被認為失敗之前可能嘗試了多少次：

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    /**
     * 嘗試佇列監聽器的次數。
     *
     * @var int
     */
    public $tries = 5;
}
```

作為定義偵聽器在失敗之前可以嘗試多少次的替代方法，你可以定義不再嘗試偵聽器的時間。這允許在給定的時間範圍內嘗試多次監聽。若要定義不再嘗試監聽器的時間，請在你的監聽器類中新增 `retryUntil` 方法。此方法應返回一個 `DateTime` 實例：

```
use DateTime;

/**
 * 確定監聽器應該超時的時間。
 */
public function retryUntil(): DateTime
{
    return now()->addMinutes(5);
}
```

24.6 調度事件

要分派一個事件，你可以在事件上呼叫靜態的 `dispatch` 方法。這個方法是通過 `Illuminate\`

`Foundation\Events\Dispatchable` 特性提供給事件的。 傳遞給 `dispatch` 方法的任何參數都將被傳遞給事件的建構函式：

```
<?php

namespace App\Http\Controllers;

use App\Events\OrderShipped;
use App\Http\Controllers\Controller;
use App\Models\Order;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class OrderShipmentController extends Controller
{
    /**
     * 運送給定的訂單。
     */
    public function store(Request $request): RedirectResponse
    {
        $order = Order::findOrFail($request->order_id);

        // 訂單出貨邏輯...

        OrderShipped::dispatch($order);

        return redirect('/orders');
    }
}
```

你可以使用 `dispatchIf` 和 `dispatchUnless` 方法根據條件分派事件：

```
OrderShipped::dispatchIf($condition, $order);

OrderShipped::dispatchUnless($condition, $order);
```

> **提示**
> 在測試時，斷言某些事件是在沒有實際觸發其偵聽器的情況下被分派的，這可能會有所幫助。 Laravel 的 內建助手 讓它變得很簡單。

24.7 事件訂閱者

24.7.1 建構事件訂閱者

事件訂閱者是可以從訂閱者類本身中訂閱多個事件的類，允許你在單個類中定義多個事件處理程序。訂閱者應該定義一個 `subscribe` 方法，它將被傳遞一個事件分派器實例。你可以在給定的分派器上呼叫 `listen` 方法來註冊事件監聽器：

```
<?php

namespace App\Listeners;

use Illuminate\Auth\Events\Login;
use Illuminate\Auth\Events\Logout;
use Illuminate\Events\Dispatcher;

class UserEventSubscriber
{
    /**
     * 處理使用者登錄事件。
     */
    public function handleUserLogin(string $event): void {}
```

```
    /**
     * 處理使用者退出事件。
     */
    public function handleUserLogout(string $event): void {}

    /**
     * 為訂閱者註冊偵聽器。
     */
    public function subscribe(Dispatcher $events): void
    {
        $events->listen(
            Login::class,
            [UserEventSubscriber::class, 'handleUserLogin']
        );

        $events->listen(
            Logout::class,
            [UserEventSubscriber::class, 'handleUserLogout']
        );
    }
}
```

如果你的事件偵聽器方法是在訂閱者本身中定義的，你可能會發現從訂閱者的「訂閱」方法返回一組事件和方法名稱會更方便。 Laravel 會在註冊事件監聽器時自動判斷訂閱者的類名：

```
<?php

namespace App\Listeners;

use Illuminate\Auth\Events\Login;
use Illuminate\Auth\Events\Logout;
use Illuminate\Events\Dispatcher;

class UserEventSubscriber
{
    /**
     * 處理使用者登錄事件。
     */
    public function handleUserLogin(string $event): void {}

    /**
     * 處理使用者註銷事件。
     */
    public function handleUserLogout(string $event): void {}

    /**
     * 為訂閱者註冊監聽器。
     *
     * @return array<string, string>
     */
    public function subscribe(Dispatcher $events): array
    {
        return [
            Login::class => 'handleUserLogin',
            Logout::class => 'handleUserLogout',
        ];
    }
}
```

24.7.2 註冊事件訂閱者

編寫訂閱者後，你就可以將其註冊到事件調度程序。 可以使用 `EventServiceProvider` 上的 `$subscribe` 屬性註冊訂閱者。 例如，讓我們將 `UserEventSubscriber` 新增到列表中：

```
<?php

namespace App\Providers;

use App\Listeners\UserEventSubscriber;
use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvider;

class EventServiceProvider extends ServiceProvider
{
    /**
     * The event listener mappings for the application.
     *
     * @var array
     */
    protected $listen = [
        // ...
    ];

    /**
     * The subscriber classes to register.
     *
     * @var array
     */
    protected $subscribe = [
        UserEventSubscriber::class,
    ];
}
```

24.8 測試

當測試分發事件的程式碼時，你可能希望指示 Laravel 不要實際執行事件的監聽器，因為監聽器的程式碼可以直接和分發相應事件的程式碼分開測試。 當然，要測試監聽器本身，你可以實例化一個監聽器實例並直接在測試中呼叫 handle 方法。

使用 `Event` 門面的 `fake` 方法，你可以阻止偵聽器執行，執行測試程式碼，然後使用 `assertDispatched`、`assertNotDispatched` 和 `assertNothingDispatched` 方法斷言你的應用程式分派了哪些事件：

```
<?php

namespace Tests\Feature;

use App\Events\OrderFailedToShip;
use App\Events\OrderShipped;
use Illuminate\Support\Facades\Event;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * 測試訂單發貨。
     */
    public function test_orders_can_be_shipped(): void
    {
        Event::fake();

        // 執行訂單發貨...

        // 斷言事件已傳送...
        Event::assertDispatched(OrderShipped::class);

        // 斷言一個事件被傳送了兩次.....
        Event::assertDispatched(OrderShipped::class, 2);
```

```
        // 斷言事件未被傳送...
        Event::assertNotDispatched(OrderFailedToShip::class);

        // 斷言沒有事件被傳送...
        Event::assertNothingDispatched();
    }
}
```

你可以將閉包傳遞給 `assertDispatched` 或 `assertNotDispatched` 方法，以斷言已派發的事件通過了給定的「真實性測試」。 如果至少傳送了一個通過給定真值測試的事件，則斷言將成功：

```
Event::assertDispatched(function (OrderShipped $event) use ($order) {
    return $event->order->id === $order->id;
});
```

如果你只想斷言事件偵聽器正在偵聽給定事件，可以使用 `assertListening` 方法：

```
Event::assertListening(
    OrderShipped::class,
    SendShipmentNotification::class
);
```

> **警告** 呼叫 `Event::fake()` 後，不會執行任何事件偵聽器。 因此，如果你的測試使用依賴於事件的模型工廠，例如在模型的「建立」事件期間建立 UUID，則您應該在使用您的工廠**之後**呼叫“Event::fake()”。

24.8.1 偽造一部分事件

如果你只想為一組特定的事件偽造事件監聽器，你可以將它們傳遞給 `fake` 或 `fakeFor` 方法：

```
/**
 * 測試訂單流程。
 */
public function test_orders_can_be_processed(): void
{
    Event::fake([
        OrderCreated::class,
    ]);

    $order = Order::factory()->create();

    Event::assertDispatched(OrderCreated::class);

    // 其他事件正常傳送...
    $order->update([...]);
}
```

你可以使用 `except` 方法排除指定事件：

```
Event::fake()->except([
    OrderCreated::class,
]);
```

24.8.2 Fakes 範疇事件

如果你只想為測試的一部分建立事件偵聽器，你可以使用 `fakeFor` 方法：

```
<?php

namespace Tests\Feature;

use App\Events\OrderCreated;
use App\Models\Order;
```

```
use Illuminate\Support\Facades\Event;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * 測試訂單程序
     */
    public function test_orders_can_be_processed(): void
    {
        $order = Event::fakeFor(function () {
            $order = Order::factory()->create();

            Event::assertDispatched(OrderCreated::class);

            return $order;
        });

        // 事件按正常方式調度，觀察者將會運行...
        $order->update([...]);
    }
}
```

25 檔案儲存

25.1 簡介

Laravel 提供了一個強大的檔案系統抽象，這要感謝 Frank de Jonge 的 Flysystem PHP 包。Laravel 的 Flysystem 整合提供了 簡單的驅動來處理本地檔案系統、SFTP 和 Amazon S3。更棒的是，在你的本地開發機器和生產伺服器之間切換這些儲存選項是非常簡單的，因為每個系統的 API 都是一樣的。

25.2 組態

Laravel 的檔案系統組態檔案位於 `config/filesystems.php`。 在這個檔案中，你可以組態你所有的檔案系統「磁碟」。每個磁碟代表一個特定的儲存驅動器和儲存位置。 每種支援的驅動器的組態示例都包含在組態檔案中, 因此你可以修改組態以反映你的儲存偏好和證書。

`local` 驅動用於與運行 Laravel 應用程式的伺服器上儲存的檔案進行互動，而 `s3` 驅動用於寫入 Amazon 的 S3 雲端儲存服務。

> **注意** 你可以組態任意數量的磁碟，甚至可以新增多個使用相同驅動的磁碟。

25.2.1 本地驅動

使用 `local` 驅動時，所有檔案操作都與 `filesystems` 組態檔案中定義的 `root` 目錄相關。 默認情況下，此值設定為 `storage/app` 目錄。因此，以下方法會把檔案儲存在 `storage/app/example.txt` 中：

```
use Illuminate\Support\Facades\Storage;

Storage::disk('local')->put('example.txt', 'Contents');
```

25.2.2 公共磁碟

在 `filesystems` 組態檔案中定義的 `public` 磁碟適用於要公開訪問的檔案。默認情況下， `public` 磁碟使用 `local` 驅動，並且將這些檔案儲存在 `storage/app/public` 目錄下。

要使這些檔案可從 web 訪問，應建立從 `public/storage` 到 `storage/app/public` 的符號連結。這種方式能把可公開訪問檔案都保留在同一個目錄下，以便在使用零停機時間部署系統如 Envoyer 的時候，就可以輕鬆地在不同的部署之間共享這些檔案。

你可以使用 Artisan 命令 `storage:link` 來建立符號連結：

```
php artisan storage:link
```

一旦一個檔案被儲存並且已經建立了符號連結，你就可以使用輔助函數 `asset` 來建立檔案的 URL：

```
echo asset('storage/file.txt');
```

你可以在 `filesystems` 組態檔案中組態額外的符號連結。這些連結將會在運行 `storage:link` 命令時自動建立：

```
'links' => [
    public_path('storage') => storage_path('app/public'),
    public_path('images') => storage_path('app/images'),
],
```

25.2.3 驅動先決要求

25.2.3.1 S3 驅動組態

在使用 S3 驅動之前，你需要通過 Composer 包管理器安裝 Flysystem S3 軟體包：

```
composer require league/flysystem-aws-s3-v3 "^3.0"
```

S3 驅動組態資訊位於你的 `config/filesystems.php` 組態檔案中。該檔案包含一個 S3 驅動的示例組態陣列。你可以自由使用自己的 S3 組態和憑證修改此陣列。為方便起見，這些環境變數與 AWS CLI 使用的命名約定相匹配。

25.2.3.2 FTP 驅動組態

在使用 FTP 驅動之前，你需要通過 Composer 包管理器安裝 Flysystem FTP 包：

```
composer require league/flysystem-ftp "^3.0"
```

Laravel 的 Flysystem 能與 FTP 很好的適配；然而，框架的默認 `filesystems.php` 組態檔案中並未包含示例組態。如果你需要組態 FTP 檔案系統，可以使用下面的組態示例：

```
'ftp' => [
    'driver' => 'ftp',
    'host' => env('FTP_HOST'),
    'username' => env('FTP_USERNAME'),
    'password' => env('FTP_PASSWORD'),

    // 可選的 FTP 設定...
    // 'port' => env('FTP_PORT', 21),
    // 'root' => env('FTP_ROOT'),
    // 'passive' => true,
    // 'ssl' => true,
    // 'timeout' => 30,
],
```

25.2.3.3 SFTP 驅動組態

在使用 SFTP 驅動之前，你需要通過 Composer 包管理器安裝 Flysystem SFTP 軟體包。

```
composer require league/flysystem-sftp-v3 "^3.0"
```

Laravel 的 Flysystem 能與 SFTP 很好的適配；然而，框架默認的 `filesystems.php` 組態檔案中並未包含示例組態。如果你需要組態 SFTP 檔案系統，可以使用下面的組態示例：

```
'sftp' => [
    'driver' => 'sftp',
    'host' => env('SFTP_HOST'),

    // 基本認證的設定...
    'username' => env('SFTP_USERNAME'),
    'password' => env('SFTP_PASSWORD'),

    // 基於 SSH 金鑰的認證與加密密碼的設定...
    'privateKey' => env('SFTP_PRIVATE_KEY'),
    'passphrase' => env('SFTP_PASSPHRASE'),

    // 可選的 SFTP 設定...
    // 'hostFingerprint' => env('SFTP_HOST_FINGERPRINT'),
    // 'maxTries' => 4,
    // 'passphrase' => env('SFTP_PASSPHRASE'),
    // 'port' => env('SFTP_PORT', 22),
    // 'root' => env('SFTP_ROOT', ''),
```

```
    // 'timeout' => 30,
    // 'useAgent' => true,
],
```

25.2.4 驅動先決條件

25.2.4.1 S3 驅動組態

在使用 S3 驅動之前，你需要通過 Composer 安裝 Flysystem S3 包：

```
composer require league/flysystem-aws-s3-v3 "^3.0"
```

S3 驅動組態資訊位於你的 `config/filesystems.php` 組態檔案中。 此檔案包含 S3 驅動的示例組態陣列。 你可以使用自己的 S3 組態和憑據自由修改此陣列。 為方便起見，這些環境變數與 AWS CLI 使用的命名約定相匹配。

25.2.4.2 FTP 驅動組態

在使用 FTP 驅動之前，你需要通過 Composer 安裝 Flysystem FTP 包：

```
composer require league/flysystem-ftp "^3.0"
```

Laravel 的 Flysystem 整合與 FTP 配合得很好； 但是，框架的默認 `filesystems.php` 組態檔案中不包含示例組態。 如果需要組態 FTP 檔案系統，可以使用下面的組態示例：

```
'ftp' => [
    'driver' => 'ftp',
    'host' => env('FTP_HOST'),
    'username' => env('FTP_USERNAME'),
    'password' => env('FTP_PASSWORD'),

    // 可選的 FTP 設定...
    // 'port' => env('FTP_PORT', 21),
    // 'root' => env('FTP_ROOT'),
    // 'passive' => true,
    // 'ssl' => true,
    // 'timeout' => 30,
],
```

25.2.4.3 SFTP 驅動組態

在使用 SFTP 驅動之前，你需要通過 Composer 安裝 Flysystem SFTP 包：

```
composer require league/flysystem-sftp-v3 "^3.0"
```

Laravel 的 Flysystem 整合與 SFTP 配合得很好； 但是，框架的默認 `filesystems.php` 組態檔案中不包含示例組態。 如果你需要組態 SFTP 檔案系統，可以使用下面的組態示例：

```
'sftp' => [
    'driver' => 'sftp',
    'host' => env('SFTP_HOST'),

    // 基本身份驗證設定...
    'username' => env('SFTP_USERNAME'),
    'password' => env('SFTP_PASSWORD'),

    // 基於 SSH 金鑰的加密密碼認證設定…
    'privateKey' => env('SFTP_PRIVATE_KEY'),
    'passphrase' => env('SFTP_PASSPHRASE'),

    // 可選的 SFTP 設定...
    // 'hostFingerprint' => env('SFTP_HOST_FINGERPRINT'),
```

```
    // 'maxTries' => 4,
    // 'passphrase' => env('SFTP_PASSPHRASE'),
    // 'port' => env('SFTP_PORT', 22),
    // 'root' => env('SFTP_ROOT', ''),
    // 'timeout' => 30,
    // 'useAgent' => true,
],
```

25.2.5 分區和唯讀檔案系統

分區磁碟允許你定義一個檔案系統，其中所有的路徑都自動帶有給定的路徑前綴。在建立一個分區檔案系統磁碟之前，你需要通過 Composer 包管理器安裝一個額外的 Flysystem 包：

```
composer require league/flysystem-path-prefixing "^3.0"
```

你可以通過定義一個使用 `scoped` 驅動的磁碟來建立任何現有檔案系統磁碟的路徑分區實例。例如，你可以建立一個磁碟，它將你現有的 `s3` 磁碟限定在特定的路徑前綴上，然後使用你的分區磁碟進行的每個檔案操作都將使用指定的前綴：

```
's3-videos' => [
    'driver' => 'scoped',
    'disk' => 's3',
    'prefix' => 'path/to/videos',
],
```

「唯讀」磁碟允許你建立不允許寫入操作的檔案系統磁碟。在使用 `read-only` 組態選項之前，你需要通過 Composer 包管理器安裝一個額外的 Flysystem 包：

```
composer require league/flysystem-read-only "^3.0"
```

接下來，你可以在一個或多個磁碟的組態陣列中包含 `read-only` 組態選項：

```
's3-videos' => [
    'driver' => 's3',
    // ...
    'read-only' => true,
],
```

25.2.6 Amazon S3 相容檔案系統

默認情況下，你的應用程式的 `filesystems` 組態檔案包含一個 `s3` 磁碟的磁碟組態。除了使用此磁碟與 Amazon S3 互動外，你還可以使用它與任何相容 S3 的檔案儲存服務（如 MinIO 或 DigitalOcean Spaces）進行互動。

通常，在更新磁碟憑據以匹配你計畫使用的服務的憑據後，你只需要更新 `endpoint` 組態選項的值。此選項的值通常通過 `AWS_ENDPOINT` 環境變數定義：

```
'endpoint' => env('AWS_ENDPOINT', 'https://minio:9000'),
```

25.2.6.1 MinIO

為了讓 Laravel 的 Flysystem 整合在使用 MinIO 時生成正確的 URL，你應該定義 `AWS_URL` 環境變數，使其與你的應用程式的本地 URL 匹配，並在 URL 路徑中包含儲存桶名稱：

```
AWS_URL=http://localhost:9000/local
```

> **警告** 當使用 MinIO 時，不支援通過 `temporaryUrl` 方法生成臨時儲存 URL。

25.3 獲取磁碟實例

Storage Facade 可用於與所有已組態的磁碟進行互動。例如，你可以使用 Facade 中的 put 方法將頭像儲存到默認磁碟。如果你在未先呼叫 disk 方法的情況下呼叫 Storage Facade 中的方法，則該方法將自動傳遞給默認磁碟：

```
use Illuminate\Support\Facades\Storage;

Storage::put('avatars/1', $content);
```

如果你的應用與多個磁碟進行互動，可使用 Storage Facade 中的 disk 方法對特定磁碟上的檔案進行操作：

```
Storage::disk('s3')->put('avatars/1', $content);
```

25.3.1 按需組態磁碟

有時你可能希望在執行階段使用給定組態建立磁碟，而無需在應用程式的 filesystems 組態檔案中實際存在該組態。為了實現這一點，你可以將組態陣列傳遞給 Storage Facade 的 build 方法：

```
use Illuminate\Support\Facades\Storage;

$disk = Storage::build([
    'driver' => 'local',
    'root' => '/path/to/root',
]);

$disk->put('image.jpg', $content);
```

25.4 檢索檔案

get 方法可用於檢索檔案的內容。該方法將返回檔案的原始字串內容。切記，所有檔案路徑的指定都應該相對於該磁碟所組態的「root」目錄：

```
$contents = Storage::get('file.jpg');
```

exists 方法可以用來判斷一個檔案是否存在於磁碟上：

```
if (Storage::disk('s3')->exists('file.jpg')) {
    // ...
}
```

missing 方法可以用來判斷一個檔案是否缺失於磁碟上：

```
if (Storage::disk('s3')->missing('file.jpg')) {
    // ...
}
```

25.4.1 下載檔案

download 方法可以用來生成一個響應，強制使用者的瀏覽器下載給定路徑的檔案。download 方法接受一個檔案名稱作為方法的第二個參數，這將決定使用者下載檔案時看到的檔案名稱。最後，你可以傳遞一個 HTTP 頭部的陣列作為方法的第三個參數：

```
return Storage::download('file.jpg');

return Storage::download('file.jpg', $name, $headers);
```

25.4.2 檔案 URL

你可以使用 `url` 方法來獲取給定檔案的 URL。如果你使用的是 `local` 驅動，這通常只會在給定路徑前加上 `/storage`，並返回一個相對 URL 到檔案。如果你使用的是 `s3` 驅動，將返回完全限定的遠端 URL：

```
use Illuminate\Support\Facades\Storage;

$url = Storage::url('file.jpg');
```

當使用 `local` 驅動時，所有應該公開訪問的檔案都應放置在 `storage/app/public` 目錄中。此外，你應該在 `public/storage` 處 建立一個符號連接 指向 `storage/app/public` 目錄。

> **警告** 當使用 `local` 驅動時，url 的返回值不是 URL 編碼的。因此，我們建議始終使用能夠建立有效 URL 的名稱儲存檔案。

25.4.2.1 定製 URL 的 Host

如果你想預定義使用 `Storage` Facade 生成的 URL 的 Host，則可以在磁碟的組態陣列中新增一個 `url` 選項：

```
'public' => [
    'driver' => 'local',
    'root' => storage_path('app/public'),
    'url' => env('APP_URL').'/storage',
    'visibility' => 'public',
],
```

25.4.3 臨時 URL

使用 `temporaryUrl` 方法，你可以為使用 `s3` 驅動儲存的檔案建立臨時 URL。此方法接受一個路徑和一個 `DateTime` 實例，指定 URL 的過期時間：

```
use Illuminate\Support\Facades\Storage;

$url = Storage::temporaryUrl(
    'file.jpg', now()->addMinutes(5)
);
```

如果你需要指定額外的 S3 請求參數，你可以將請求參數陣列作為第三個參數傳遞給 `temporaryUrl` 方法。

```
$url = Storage::temporaryUrl(
    'file.jpg',
    now()->addMinutes(5),
    [
        'ResponseContentType' => 'application/octet-stream',
        'ResponseContentDisposition' => 'attachment; filename=file2.jpg',
    ]
);
```

如果你需要為一個特定的儲存磁碟定製臨時 URL 的建立方式，可以使用 `buildTemporaryUrlsUsing` 方法。例如，如果你有一個 controller 允許你通過不支援臨時 URL 的磁碟下載儲存的檔案，這可能很有用。通常，此方法應從服務提供者的 `boot` 方法中呼叫：

```
<?php

namespace App\Providers;

use DateTime;
use Illuminate\Support\Facades\Storage;
use Illuminate\Support\Facades\URL;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
```

```
    /**
     * 啟動任何應用程式服務。
     */
    public function boot(): void
    {
        Storage::disk('local')->buildTemporaryUrlsUsing(
            function (string $path, DateTime $expiration, array $options) {
                return URL::temporarySignedRoute(
                    'files.download',
                    $expiration,
                    array_merge($options, ['path' => $path])
                );
            }
        );
    }
}
```

25.4.3.1 URL Host 自訂

如果你想為使用 `Storage` Facade 生成的 URL 預定義 Host，可以將 `url` 選項新增到磁碟的組態陣列：

```
'public' => [
    'driver' => 'local',
    'root' => storage_path('app/public'),
    'url' => env('APP_URL').'/storage',
    'visibility' => 'public',
],
```

25.4.3.2 臨時上傳 URL

> **警告** 生成臨時上傳 URL 的能力僅由 `s3` 驅動支援。

如果你需要生成一個臨時 URL，可以直接從客戶端應用程式上傳檔案，你可以使用 `temporaryUploadUrl` 方法。此方法接受一個路徑和一個 `DateTime` 實例，指定 URL 應該在何時過期。`temporaryUploadUrl` 方法返回一個關聯陣列，可以解構為上傳 URL 和應該包含在上傳請求中的頭部：

```
use Illuminate\Support\Facades\Storage;

['url' => $url, 'headers' => $headers] = Storage::temporaryUploadUrl(
    'file.jpg', now()->addMinutes(5)
);
```

此方法主要用於無伺服器環境，需要客戶端應用程式直接將檔案上傳到雲端儲存系統（如 Amazon S3）。

25.4.4 檔案中繼資料

除了讀寫檔案，Laravel 還可以提供有關檔案本身的資訊。例如，`size` 方法可用於獲取檔案大小（以位元組為單位）：

```
use Illuminate\Support\Facades\Storage;

$size = Storage::size('file.jpg');
```

`lastModified` 方法返回上次修改檔案時的時間戳：

```
$time = Storage::lastModified('file.jpg');
```

可以通過 `mimeType` 方法獲取給定檔案的 MIME 類型：

```
$mime = Storage::mimeType('file.jpg')
```

25.4.4.1 檔案路徑

你可以使用 path 方法獲取給定檔案的路徑。如果你使用的是 local 驅動，這將返回檔案的絕對路徑。如果你使用的是 s3 驅動，此方法將返回 S3 儲存桶中檔案的相對路徑：

```
use Illuminate\Support\Facades\Storage;

$path = Storage::path('file.jpg');
```

25.5 保存檔案

可以使用 put 方法將檔案內容儲存在磁碟上。你還可以將 PHP resource 傳遞給 put 方法，該方法將使用 Flysystem 的底層流支援。請記住，應相對於為磁碟組態的「根」目錄指定所有檔案路徑：

```
use Illuminate\Support\Facades\Storage;

Storage::put('file.jpg', $contents);

Storage::put('file.jpg', $resource);
```

25.5.1.1 寫入失敗

如果 put 方法（或其他「寫入」操作）無法將檔案寫入磁碟，將返回 false。

```
if (! Storage::put('file.jpg', $contents)) {
    // 該檔案無法寫入磁碟...
}
```

你可以在你的檔案系統磁碟的組態陣列中定義 throw 選項。當這個選項被定義為 true 時，「寫入」的方法如 put 將在寫入操作失敗時拋出一個 League\Flysystem\UnableToWriteFile 的實例。

```
'public' => [
    'driver' => 'local',
    // ...
    'throw' => true,
],
```

25.5.2 追加內容到檔案開頭或結尾

prepend 和 append 方法允許你將內容寫入檔案的開頭或結尾：

```
Storage::prepend('file.log', 'Prepended Text');

Storage::append('file.log', 'Appended Text');
```

25.5.3 複製 / 移動檔案

copy 方法可用於將現有檔案複製到磁碟上的新位置，而 move 方法可用於重新命名現有檔案或將其移動到新位置：

```
Storage::copy('old/file.jpg', 'new/file.jpg');

Storage::move('old/file.jpg', 'new/file.jpg');
```

25.5.4 自動流式傳輸

將檔案流式傳輸到儲存位置可顯著減少記憶體使用量。如果你希望 Laravel 自動管理將給定檔案流式傳輸到你的儲存位置，你可以使用 putFile 或 putFileAs 方法。此方法接受一個 Illuminate\Http\File 或 Illuminate\Http\UploadedFile 實例，並自動將檔案流式傳輸到你所需的位置：

```
use Illuminate\Http\File;
use Illuminate\Support\Facades\Storage;

// 為檔案名稱自動生成一個唯一的 ID...
$path = Storage::putFile('photos', new File('/path/to/photo'));

// 手動指定一個檔案名稱...
$path = Storage::putFileAs('photos', new File('/path/to/photo'), 'photo.jpg');
```

關於 putFile 方法有幾點重要的注意事項。注意，我們只指定了目錄名稱而不是檔案名稱。默認情況下，`putFile` 方法將生成一個唯一的 ID 作為檔案名稱。檔案的擴展名將通過檢查檔案的 MIME 類型來確定。檔案的路徑將由 `putFile` 方法返回，因此你可以將路徑（包括生成的檔案名稱）儲存在資料庫中。

`putFile` 和 `putFileAs` 方法還接受一個參數來指定儲存檔案的「可見性」。如果你將檔案儲存在雲盤（如 Amazon S3）上，並希望檔案通過生成的 URL 公開訪問，這一點特別有用：

```
Storage::putFile('photos', new File('/path/to/photo'), 'public');
```

25.5.5 檔案上傳

在網路應用程式中，儲存檔案的最常見用例之一是儲存使用者上傳的檔案，如照片和文件。Laravel 使用上傳檔案實例上的 `store` 方法非常容易地儲存上傳的檔案。使用你希望儲存上傳檔案的路徑呼叫 `store` 方法：

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;

class UserAvatarController extends Controller
{
    /**
     * 更新使用者的頭像。
     */
    public function update(Request $request): string
    {
        $path = $request->file('avatar')->store('avatars');

        return $path;
    }
}
```

關於這個例子有幾點重要的注意事項。注意，我們只指定了目錄名稱而不是檔案名稱。默認情況下，`store` 方法將生成一個唯一的 ID 作為檔案名稱。檔案的擴展名將通過檢查檔案的 MIME 類型來確定。檔案的路徑將由 `store` 方法返回，因此你可以將路徑（包括生成的檔案名稱）儲存在資料庫中。

你也可以在 `Storage` Facade 上呼叫 `putFile` 方法來執行與上面示例相同的檔案儲存操作：

```
$path = Storage::putFile('avatars', $request->file('avatar'));
```

25.5.5.1 指定一個檔案名稱

如果你不希望檔案名稱被自動分配給你儲存的檔案，你可以使用 `storeAs` 方法，該方法接收路徑、檔案名稱和（可選的）磁碟作為其參數：

```
$path = $request->file('avatar')->storeAs(
    'avatars', $request->user()->id
);
```

你也可以在 `Storage` Facade 使用 `putFileAs` 方法，它將執行與上面示例相同的檔案儲存操作：

```
$path = Storage::putFileAs(
    'avatars', $request->file('avatar'), $request->user()->id
```

```
);
```

> **警告** 不可列印和無效的 Unicode 字元將自動從檔案路徑中刪除。因此，你可能希望在將檔案路徑傳遞給 Laravel 的檔案儲存方法之前對其進行清理。檔案路徑使用 `League\Flysystem\WhitespacePathNormalizer::normalizePath` 方法進行規範化。

25.5.5.2 指定一個磁碟

默認情況下，此上傳檔案的 `store` 方法將使用你的默認磁碟。如果你想指定另一個磁碟，將磁碟名稱作為第二個參數傳遞給 `store` 方法：

```
$path = $request->file('avatar')->store(
    'avatars/'.$request->user()->id, 's3'
);
```

如果你正在使用 `storeAs` 方法，你可以將磁碟名稱作為第三個參數傳遞給該方法：

```
$path = $request->file('avatar')->storeAs(
    'avatars',
    $request->user()->id,
    's3'
);
```

25.5.5.3 其他上傳檔案的資訊

如果您想獲取上傳檔案的原始名稱和擴展名，可以使用 `getClientOriginalName` 和 `getClientOriginalExtension` 方法來實現：

```
$file = $request->file('avatar');

$name = $file->getClientOriginalName();
$extension = $file->getClientOriginalExtension();
```

然而，請記住，`getClientOriginalName` 和 `getClientOriginalExtension` 方法被認為是不安全的，因為檔案名稱和擴展名可能被惡意使用者篡改。因此，你通常應該更喜歡使用 `hashName` 和 `extension` 方法來獲取給定檔案上傳的名稱和擴展名：

```
$file = $request->file('avatar');

$name = $file->hashName(); // 生成一個唯一的、隨機的名字...
$extension = $file->extension(); // 根據檔案的 MIME 類型來確定檔案的擴展名...
```

25.5.6 檔案可見性

在 Laravel 的 Flysystem 整合中，「visibility」 是跨多個平台的檔案權限的抽象。檔案可以被聲明為 `public` 或 `private`。當一個檔案被聲明為 `public` 時，你表示該檔案通常應該被其他人訪問。例如，在使用 S3 驅動程式時，你可以檢索 `public` 檔案的 URL。

你可以通過 `put` 方法在寫入檔案時設定可見性：

```
use Illuminate\Support\Facades\Storage;

Storage::put('file.jpg', $contents, 'public');
```

如果檔案已經被儲存，可以通過 `getVisibility` 和 `setVisibility` 方法檢索和設定其可見性：

```
$visibility = Storage::getVisibility('file.jpg');

Storage::setVisibility('file.jpg', 'public');
```

在與上傳檔案互動時，你可以使用 `storePublicly` 和 `storePubliclyAs` 方法將上傳檔案儲存為 `public` 可見性

```
$path = $request->file('avatar')->storePublicly('avatars', 's3');

$path = $request->file('avatar')->storePubliclyAs(
    'avatars',
    $request->user()->id,
    's3'
);
```

25.5.6.1 本地檔案和可見性

當使用 `local` 驅動時，`public` 可見性轉換為目錄的 `0755` 權限和檔案的 `0644` 權限。你可以在你的應用程式的 `filesystems` 組態檔案中修改權限對應：

```
'local' => [
    'driver' => 'local',
    'root' => storage_path('app'),
    'permissions' => [
        'file' => [
            'public' => 0644,
            'private' => 0600,
        ],
        'dir' => [
            'public' => 0755,
            'private' => 0700,
        ],
    ],
],
```

25.6 刪除檔案

`delete` 方法接收一個檔案名稱或一個檔案名稱陣列來將其從磁碟中刪除：

```
use Illuminate\Support\Facades\Storage;

Storage::delete('file.jpg');

Storage::delete(['file.jpg', 'file2.jpg']);
```

如果需要，你可以指定應從哪個磁碟刪除檔案。

```
use Illuminate\Support\Facades\Storage;

Storage::disk('s3')->delete('path/file.jpg');
```

25.7 目錄

25.7.1.1 獲取目錄下所有的檔案

`files` 將以陣列的形式返回給定目錄下所有的檔案。如果你想要檢索給定目錄的所有檔案及其子目錄的所有檔案，你可以使用 `allFiles` 方法：

```
use Illuminate\Support\Facades\Storage;

$files = Storage::files($directory);

$files = Storage::allFiles($directory);
```

25.7.1.2 獲取特定目錄下的子目錄

`directories` 方法以陣列的形式返回給定目錄中的所有目錄。此外，你還可以使用 `allDirectories` 方法

遞迴地獲取給定目錄中的所有目錄及其子目錄中的目錄：

```
$directories = Storage::directories($directory);

$directories = Storage::allDirectories($directory);
```

25.7.1.3 建立目錄

`makeDirectory` 方法可遞迴的建立指定的目錄：

```
Storage::makeDirectory($directory);
```

25.7.1.4 刪除一個目錄

最後，`deleteDirectory` 方法可用於刪除一個目錄及其下所有的檔案：

```
Storage::deleteDirectory($directory);
```

25.8 測試

The `Storage` facade's `fake` method allows you to easily generate a fake disk that, combined with the file generation utilities of the `Illuminate\Http\UploadedFile` class, greatly simplifies the testing of file uploads. For example:

`Storage` 門面類的 `fake` 方法可以輕鬆建立一個虛擬磁碟，與 `Illuminate\Http\UploadedFile` 類配合使用，大大簡化了檔案的上傳測試。例如：

```
<?php

namespace Tests\Feature;

use Illuminate\Http\UploadedFile;
use Illuminate\Support\Facades\Storage;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_albums_can_be_uploaded(): void
    {
        Storage::fake('photos');

        $response = $this->json('POST', '/photos', [
            UploadedFile::fake()->image('photo1.jpg'),
            UploadedFile::fake()->image('photo2.jpg')
        ]);

        // 斷言儲存了一個或多個檔案。
        Storage::disk('photos')->assertExists('photo1.jpg');
        Storage::disk('photos')->assertExists(['photo1.jpg', 'photo2.jpg']);

        // 斷言一個或多個檔案未儲存。
        Storage::disk('photos')->assertMissing('missing.jpg');
        Storage::disk('photos')->assertMissing(['missing.jpg', 'non-existing.jpg']);

        // 斷言給定目錄為空。
        Storage::disk('photos')->assertDirectoryEmpty('/wallpapers');
    }
}
```

默認情況下，`fake` 方法將刪除臨時目錄中的所有檔案。如果你想保留這些檔案，你可以使用 “persistentFake” 方法代替。有關測試檔案上傳的更多資訊，您可以查閱 HTTP 測試文件的檔案上傳.

> **警告** `image` 方法需要 GD 擴展 .

25.9 自訂檔案系統

Laravel 內建的檔案系統提供了一些開箱即用的驅動；當然，它不僅僅是這些，它還提供了與其他儲存系統的介面卡。通過這些介面卡，你可以在你的 Laravel 應用中建立自訂驅動。

要安裝自訂檔案系統，你可能需要一個檔案系統介面卡。讓我們將社區維護的 Dropbox 介面卡新增到項目中：

```
composer require spatie/flysystem-dropbox
```

接下來，你可以在 服務提供者 中註冊一個帶有 `boot` 方法的驅動。在提供者的 `boot` 方法中，你可以使用 `Storage` 門面的 `extend` 方法來定義一個自訂驅動：

```
<?php

namespace App\Providers;

use Illuminate\Contracts\Foundation\Application;
use Illuminate\Filesystem\FilesystemAdapter;
use Illuminate\Support\Facades\Storage;
use Illuminate\Support\ServiceProvider;
use League\Flysystem\Filesystem;
use Spatie\Dropbox\Client as DropboxClient;
use Spatie\FlysystemDropbox\DropboxAdapter;

class AppServiceProvider extends ServiceProvider
{
    /**
     * 註冊任意應用程式服務。
     */
    public function register(): void
    {
        // ...
    }

    /**
     * 引導任何應用程式服務。
     */
    public function boot(): void
    {
        Storage::extend('dropbox', function (Application $app, array $config) {
            $adapter = new DropboxAdapter(new DropboxClient(
                $config['authorization_token']
            ));

            return new FilesystemAdapter(
                new Filesystem($adapter, $config),
                $adapter,
                $config
            );
        });
    }
}
```

`extend` 方法的第一個參數是驅動程式的名稱，第二個參數是接收 `$app` 和 `$config` 變數的閉包。閉包必須返回的實例 `League\Flysystem\Filesystem`。`$config` 變數包含 `config/filesystems.php` 為指定磁碟定義的值。

一旦建立並註冊了擴展的服務提供商，就可以 `dropbox` 在 `config/filesystems.php` 組態檔案中使用該驅動程式。

26 HTTP Client

26.1 簡介

Laravel 為 Guzzle HTTP 客戶端 提供了一套語義化且輕量的 API，讓你可用快速地使用 HTTP 請求與其他 Web 應用進行通訊。該 API 專注於其在常見用例中的快速實現以及良好的開發者體驗。

在開始之前，你需要確保你的項目已經安裝了 Guzzle 包作為依賴項。默認情況下，Laravel 已經包含了 Guzzle 包。但如果你此前手動移除了它，你也可以通過 Composer 重新安裝它：

```
composer require guzzlehttp/guzzle
```

26.2 建立請求

你可以使用 `Http` Facade 提供的 `head`, `get`, `post`, `put`, `patch`，以及 `delete` 方法來建立請求。首先，讓我們先看一下如何發出一個基礎的 GET 請求：

```
use Illuminate\Support\Facades\Http;

$response = Http::get('http://example.com');
```

`get` 方法返回一個 `Illuminate\Http\Client\Response` 的實例，該實例提供了大量的方法來檢查請求的響應：

```
$response->body() : string;
$response->json($key = null, $default = null) : array|mixed;
$response->object() : object;
$response->collect($key = null) : Illuminate\Support\Collection;
$response->status() : int;
$response->successful() : bool;
$response->redirect(): bool;
$response->failed() : bool;
$response->clientError() : bool;
$response->header($header) : string;
$response->headers() : array;
```

`Illuminate\Http\Client\Response` 對象同樣實現了 PHP 的 `ArrayAccess` 介面，這代表著你可以直接訪問響應的 JSON 資料：

```
return Http::get('http://example.com/users/1')['name'];
```

除了上面列出的響應方法之外，還可以使用以下方法來確定響應是否具有相映的狀態碼：

```
$response->ok() : bool;                  // 200 OK
$response->created() : bool;             // 201 Created
$response->accepted() : bool;            // 202 Accepted
$response->noContent() : bool;           // 204 No Content
$response->movedPermanently() : bool;    // 301 Moved Permanently
$response->found() : bool;               // 302 Found
$response->badRequest() : bool;          // 400 Bad Request
$response->unauthorized() : bool;        // 401 Unauthorized
$response->paymentRequired() : bool;     // 402 Payment Required
$response->forbidden() : bool;           // 403 Forbidden
$response->notFound() : bool;            // 404 Not Found
$response->requestTimeout() : bool;      // 408 Request Timeout
$response->conflict() : bool;            // 409 Conflict
$response->unprocessableEntity() : bool; // 422 Unprocessable Entity
$response->tooManyRequests() : bool;     // 429 Too Many Requests
$response->serverError() : bool;         // 500 Internal Server Error
```

26.2.1.1 URI 範本

HTTP 客戶端還允許你使用 URI 範本規範 構造請求 URL. 要定義 URI 查詢參數，你可以使用 `withUrlParameters` 方法：

```
Http::withUrlParameters([
    'endpoint' => 'https://laravel.com',
    'page' => 'docs',
    'version' => '9.x',
    'topic' => 'validation',
])->get('{+endpoint}/{page}/{version}/{topic}');
```

26.2.1.2 列印請求資訊

如果要在傳送請求之前列印輸出請求資訊並且結束指令碼運行，你應該在建立請求前呼叫 `dd` 方法：

```
return Http::dd()->get('http://example.com');
```

26.2.2 請求資料

大多數情況下，`POST`、 `PUT` 和 `PATCH` 攜帶著額外的請求資料是相當常見的。所以，這些方法的第二個參數接受一個包含著請求資料的陣列。默認情況下，這些資料會使用 `application/json` 類型隨請求傳送：

```
use Illuminate\Support\Facades\Http;

$response = Http::post('http://example.com/users', [
    'name' => 'Steve',
    'role' => 'Network Administrator',
]);
```

26.2.2.1 GET 請求查詢參數

在建立 `GET` 請求時，你可以通過直接向 URL 新增查詢字串或是將鍵值對作為第二個參數傳遞給 `get` 方法：

```
$response = Http::get('http://example.com/users', [
    'name' => 'Taylor',
    'page' => 1,
]);
```

26.2.2.2 傳送 URL 編碼請求

如果你希望使用 `application/x-www-form-urlencoded` 作為請求的資料類型，你應該在建立請求前呼叫 `asForm` 方法：

```
$response = Http::asForm()->post('http://example.com/users', [
    'name' => 'Sara',
    'role' => 'Privacy Consultant',
]);
```

26.2.2.3 傳送原始資料（Raw）請求

如果你想使用一個原始請求體傳送請求，你可以在建立請求前呼叫 `withBody` 方法。你還可以將資料類型作為第二個參數傳遞給 `withBody` 方法：

```
$response = Http::withBody(
    base64_encode($photo), 'image/jpeg'
)->post('http://example.com/photo');
```

26.2.2.4 Multi-Part 請求

如果你希望將檔案作為 Multipart 請求傳送，你應該在建立請求前呼叫 `attach` 方法。該方法接受檔案的名字（相當於 HTML Input 的 name 屬性）以及它對應的內容。你也可以在第三個參數傳入自訂的檔案名稱，這不是必須的。如果有需要，你也可以通過第三個參數來指定檔案的檔案名稱：

```
$response = Http::attach(
    'attachment', file_get_contents('photo.jpg'), 'photo.jpg'
)->post('http://example.com/attachments');
```

除了傳遞檔案的原始內容，你也可以傳遞 Stream 流資料：

```
$photo = fopen('photo.jpg', 'r');

$response = Http::attach(
    'attachment', $photo, 'photo.jpg'
)->post('http://example.com/attachments');
```

26.2.3 要求標頭

你可以通過 `withHeaders` 方法新增要求標頭。該 `withHeaders` 方法接受一個陣列格式的鍵 / 值對：

```
$response = Http::withHeaders([
    'X-First' => 'foo',
    'X-Second' => 'bar'
])->post('http://example.com/users', [
    'name' => 'Taylor',
]);
```

你可以使用 `accept` 方法指定應用程式響應你的請求所需的內容類型：

```
$response = Http::accept('application/json')->get('http://example.com/users');
```

為方便起見，你可以使用 `acceptJson` 方法快速指定應用程式需要 `application/json` 內容類型來響應你的請求：

```
$response = Http::acceptJson()->get('http://example.com/users');
```

26.2.4 認證

你可以使用 `withBasicAuth` 和 `withDigestAuth` 方法來分別指定使用 Basic 或是 Digest 認證方式：

```
// Basic 認證方式...
$response = Http::withBasicAuth('taylor@laravel.com', 'secret')->post(/* ... */);

// Digest 認證方式...
$response = Http::withDigestAuth('taylor@laravel.com', 'secret')->post(/* ... */);
```

26.2.4.1 Bearer 令牌

如果你想要為你的請求快速新增 `Authorization` Token 令牌要求標頭，你可以使用 `withToken` 方法：

```
$response = Http::withToken('token')->post(/* ... */);
```

26.2.5 超時

該 `timeout` 方法用於指定響應的最大等待秒數：

```
$response = Http::timeout(3)->get(/* ... */);
```

如果響應時間超過了指定的超時時間，將會拋出 `Illuminate\Http\Client\ConnectionException` 異常。

你可以嘗試使用 connectTimeout 方法指定連接到伺服器時等待的最大秒數：

```
$response = Http::connectTimeout(3)->get(/* ... */);
```

26.2.6 重試

如果你希望 HTTP 客戶端在發生客戶端或伺服器端錯誤時自動進行重試，你可以使用 retry 方法。該 retry 方法接受兩個參數：重新嘗試次數以及重試間隔（毫秒）：

```
$response = Http::retry(3, 100)->post(/* ... */);
```

如果需要，你可以將第三個參數傳遞給該 retry 方法。第三個參數應該是一個可呼叫的，用於確定是否應該實際嘗試重試。例如，你可能希望僅在初始請求遇到以下情況時重試請求 ConnectionException：

```
use Exception;
use Illuminate\Http\Client\PendingRequest;

$response = Http::retry(3, 100, function (Exception $exception, PendingRequest $request)
{
    return $exception instanceof ConnectionException;
})->post(/* ... */);
```

如果請求失敗，你可以在新請求之前更改請求。你可以通過修改 retry 方法的第三個請求參數來實現這一點。例如，當請求返回身份驗證錯誤，則可以使用新的授權令牌重試請求：

```
use Exception;
use Illuminate\Http\Client\PendingRequest;

$response = Http::withToken($this->getToken())->retry(2, 0, function (Exception
$exception, PendingRequest $request) {
    if (! $exception instanceof RequestException || $exception->response->status() !==
401) {
        return false;
    }

    $request->withToken($this->getNewToken());

    return true;
})->post(/* ... */);
```

所有請求都失敗時，將會拋出一個 Illuminate\Http\Client\RequestException 實例。如果不想拋出錯誤，你需要設定請求方法的 throw 參數為 false。禁止後，當所有的請求都嘗試完成後，最後一個響應將會 return 回來：

```
$response = Http::retry(3, 100, throw: false)->post(/* ... */);
```

> **注意**
> 如果所有的請求都因為連接問題失敗， 即使 throw 屬性設定為 false，Illuminate\Http\Client\ConnectionException 錯誤依舊會被拋出。

26.2.7 錯誤處理

與 Guzzle 的默認處理方式不同，Laravel 的 HTTP 客戶端在客戶端或者伺服器端出現 4xx 或者 5xx 錯誤時並不會拋出錯誤。你應該通過 successful、 clientError 或 serverError 方法來校驗返回的響應是否有錯誤資訊:

```
// 判斷狀態碼是否是 2xx
$response->successful();

// 判斷錯誤碼是否是 4xx 或 5xx
$response->failed();

// 判斷錯誤碼是 4xx
$response->clientError();
```

```
// 判斷錯誤碼是 5xx
$response->serverError();

// 如果出現客戶端或伺服器錯誤，則執行給定的回呼
$response->onError(callable $callback);
```

26.2.7.1 主動拋出錯誤

如果你想在收到的響應是客戶端或者伺服器端錯誤時拋出一個 `Illuminate\Http\Client\RequestException` 實例，你可以使用 `throw` 或 `throwIf` 方法：

```
use Illuminate\Http\Client\Response;

$response = Http::post(/* ... */);

// 當收到伺服器端或客戶端錯誤時拋出
$response->throw();

// 當滿足 condition 條件是拋出錯誤
$response->throwIf($condition);

// 當給定的閉包執行結果是 true 時拋出錯誤
$response->throwIf(fn (Response $response) => true);

// 當給定條件是 false 是拋出錯誤
$response->throwUnless($condition);

// 當給定的閉包執行結果是 false 時拋出錯誤
$response->throwUnless(fn (Response $response) => false);

// 當收到的狀態碼是 403 時拋出錯誤
$response->throwIfStatus(403);

// 當收到的狀態碼不是 200 時拋出錯誤
$response->throwUnlessStatus(200);

return $response['user']['id'];
```

`Illuminate\Http\Client\RequestException` 實例擁有一個 `$response` 公共屬性，該屬性允許你檢查返回的響應。

如果沒有發生錯誤，`throw` 方法返迴響應實例，你可以將其他操作連結到 `throw` 方法：

```
return Http::post(/* ... */)->throw()->json();
```

如果你希望在拋出異常前進行一些操作，你可以向 `throw` 方法傳遞一個閉包。異常將會在閉包執行完成後自動拋出，你不必在閉包內手動拋出異常：

```
use Illuminate\Http\Client\Response;
use Illuminate\Http\Client\RequestException;

return Http::post(/* ... */)->throw(function (Response $response, RequestException $e) {
    // ...
})->json();
```

26.2.8 Guzzle 中介軟體

由於 Laravel 的 HTTP 客戶端是由 Guzzle 提供支援的, 你可以利用 Guzzle 中介軟體 來操作發出的請求或檢查傳入的響應。要操作發出的請求，需要通過 `withMiddleware` 方法和 Guzzle 的 `mapRequest` 中介軟體工廠註冊一個 Guzzle 中介軟體：

```
use GuzzleHttp\Middleware;
use Illuminate\Support\Facades\Http;
use Psr\Http\Message\RequestInterface;
```

```
$response = Http::withMiddleware(
    Middleware::mapRequest(function (RequestInterface $request) {
        $request = $request->withHeader('X-Example', 'Value');

        return $request;
    })
)->get('http://example.com');
```

同樣地，你可以通過 `withMiddleware` 方法結合 Guzzle 的 `mapResponse` 中介軟體工廠註冊一個中介軟體來檢查傳入的 HTTP 響應：

```
use GuzzleHttp\Middleware;
use Illuminate\Support\Facades\Http;
use Psr\Http\Message\ResponseInterface;

$response = Http::withMiddleware(
    Middleware::mapResponse(function (ResponseInterface $response) {
        $header = $response->getHeader('X-Example');

        // ...

        return $response;
    })
)->get('http://example.com');
```

26.2.9 Guzzle 選項

你可以使用 `withOptions` 方法來指定額外的 Guzzle 請求組態。`withOptions` 方法接受陣列形式的鍵 / 值對：

```
$response = Http::withOptions([
    'debug' => true,
])->get('http://example.com/users');
```

26.3 並行請求

有時，你可能希望同時發出多個 HTTP 請求。換句話說，你希望同時分派多個請求，而不是按順序發出請求。當與慢速 HTTP API 互動時，這可以顯著提高性能。

值得慶幸的是，你可以使用該 `pool` 方法完成此操作。`pool` 方法接受一個接收 `Illuminate\Http\Client\Pool` 實例的閉包，能讓你輕鬆地將請求新增到請求池以進行調度：

```
use Illuminate\Http\Client\Pool;
use Illuminate\Support\Facades\Http;

$responses = Http::pool(fn (Pool $pool) => [
    $pool->get('http://localhost/first'),
    $pool->get('http://localhost/second'),
    $pool->get('http://localhost/third'),
]);

return $responses[0]->ok() &&
       $responses[1]->ok() &&
       $responses[2]->ok();
```

如你所見，每個響應實例可以按照新增到池中的順序來訪問。你可以使用 `as` 方法命名請求，該方法能讓你按名稱訪問相應的響應：

```
use Illuminate\Http\Client\Pool;
use Illuminate\Support\Facades\Http;

$responses = Http::pool(fn (Pool $pool) => [
    $pool->as('first')->get('http://localhost/first'),
    $pool->as('second')->get('http://localhost/second'),
```

```
    $pool->as('third')->get('http://localhost/third'),
]);

return $responses['first']->ok();
```

26.4 宏

Laravel HTTP 客戶端允許你定義「宏」（macros），這可以作為一種流暢、表達力強的機制，在與應用程式中的服務互動時組態常見的請求路徑和標頭。要開始使用，你可以在應用程式的 `App\Providers\AppServiceProvider` 類的 `boot` 方法中定義宏：

```
use Illuminate\Support\Facades\Http;

/**
 * 引導應用程式服務。
 */
public function boot(): void
{
    Http::macro('github', function () {
        return Http::withHeaders([
            'X-Example' => 'example',
        ])->baseUrl('https://github.com');
    });
}
```

一旦你組態了宏，你可以在應用程式的任何地方呼叫它，以使用指定的組態建立一個掛起的請求：

```
$response = Http::github()->get('/');
```

26.5 測試

許多 Laravel 服務提供功能來幫助你輕鬆、表達性地編寫測試，而 Laravel 的 HTTP 客戶端也不例外。`Http` 門面的 `fake` 方法允許你指示 HTTP 客戶端在發出請求時返回存根/虛擬響應。

26.5.1 偽造響應

例如，要指示 HTTP 客戶端在每個請求中返回空的 `200` 狀態碼響應，你可以呼叫 `fake` 方法而不傳遞參數：

```
use Illuminate\Support\Facades\Http;

Http::fake();

$response = Http::post(/* ... */);
```

26.5.1.1 偽造特定的 URL

另外，你可以向 `fake` 方法傳遞一個陣列。該陣列的鍵應該代表你想要偽造的 URL 模式及其關聯的響應。`*` 字元可以用作萬用字元。任何請求到未偽造的 URL 的請求將會被實際執行。你可以使用 `Http` 門面的 `response` 方法來建構這些端點的存根/虛擬響應：

```
Http::fake([
    // 為 GitHub 端點存根一個 JSON 響應...
    'github.com/*' => Http::response(['foo' => 'bar'], 200, $headers),

    // 為 Google 端點存根一個字串響應...
    'google.com/*' => Http::response('Hello World', 200, $headers),
]);
```

如果你想指定一個後備 URL 模式來存根所有不匹配的 URL，你可以使用單個 `*` 字元：

```
Http::fake([
    // 為 GitHub 端點存根 JSON 響應……
    'github.com/*' => Http::response(['foo' => 'bar'], 200, ['Headers']),

    // 為其他所有端點存根字串響應……
    '*' => Http::response('Hello World', 200, ['Headers']),
]);
```

26.5.1.2 偽造響應序列

有時候，你可能需要為單個 URL 指定其一系列的偽造響應的返回順序。你可以使用 `Http::sequence` 方法來建構響應，以實現這個功能：

```
Http::fake([
    // 存根 GitHub 端點的一系列響應……
    'github.com/*' => Http::sequence()
                            ->push('Hello World', 200)
                            ->push(['foo' => 'bar'], 200)
                            ->pushStatus(404),
]);
```

當響應序列中的所有響應都被消費完後，後續的任何請求都將導致相應序列拋出一個異常。如果你想要在響應序列為空時指定一個默認的響應，則可以使用 `whenEmpty` 方法：

```
Http::fake([
    // 為 GitHub 端點存根一系列響應
    'github.com/*' => Http::sequence()
                            ->push('Hello World', 200)
                            ->push(['foo' => 'bar'], 200)
                            ->whenEmpty(Http::response()),
]);
```

如果你想要偽造一個響應序列，但你又期望在偽造的時候無需指定一個特定的 URL 匹配模式，那麼你可以使用 `Http::fakeSequence` 方法：

```
Http::fakeSequence()
        ->push('Hello World', 200)
        ->whenEmpty(Http::response());
```

26.5.1.3 Fake 回呼

如果你需要更為複雜的邏輯來確定某些端點返回什麼響應，你需要傳遞一個閉包給 `fake` 方法。這個閉包應該接受一個 `Illuminate\Http\Client\Request` 實例且返回一個響應實例。在閉包中你可以執行任何必要的邏輯來確定要返回的響應類型：

```
use Illuminate\Http\Client\Request;

Http::fake(function (Request $request) {
    return Http::response('Hello World', 200);
});
```

26.5.2 避免「流浪的」請求（確保請求總是偽造的）

如果你想確保通過 HTTP 客戶端傳送的所有請求在整個單獨的測試或完整的測試套件中都是偽造的，那麼你可以呼叫 `preventStrayRequests` 方法。在呼叫該方法後，如果一個請求沒有與之相匹配的偽造的響應，則將會拋出一個異常而不是發起一個真實的請求：

```
use Illuminate\Support\Facades\Http;

Http::preventStrayRequests();

Http::fake([
    'github.com/*' => Http::response('ok'),
```

```
]);

// 將會返回 OK 響應……
Http::get('https://github.com/laravel/framework');

// 拋出一個異常……
Http::get('https://laravel.com');
```

26.5.3 檢查請求

在偽造響應時，你可能希望檢查客戶端收到的請求，以確保你的應用程式發出了正確的資料和標頭。你可以在呼叫 `Http::fake` 方法後呼叫 `Http::assertSent` 方法來實現這個功能。

`assertSent` 方法接受一個閉包，該閉包應當接收一個 `Illuminate\Http\Client\Request` 實例且返回一個布林值，該布林值指示請求是否符合預期。為了使得測試通過，必須至少發出一個符合給定預期的請求：

```
use Illuminate\Http\Client\Request;
use Illuminate\Support\Facades\Http;

Http::fake();

Http::withHeaders([
    'X-First' => 'foo',
])->post('http://example.com/users', [
    'name' => 'Taylor',
    'role' => 'Developer',
]);

Http::assertSent(function (Request $request) {
    return $request->hasHeader('X-First', 'foo') &&
           $request->url() == 'http://example.com/users' &&
           $request['name'] == 'Taylor' &&
           $request['role'] == 'Developer';
});
```

如果有需要，你可以使用 `assertNotSent` 方法來斷言未發出指定的請求：

```
use Illuminate\Http\Client\Request;
use Illuminate\Support\Facades\Http;

Http::fake();

Http::post('http://example.com/users', [
    'name' => 'Taylor',
    'role' => 'Developer',
]);

Http::assertNotSent(function (Request $request) {
    return $request->url() === 'http://example.com/posts';
});
```

你可以使用 `assertSentCount` 方法來斷言在測試過程中發出的請求數量：

```
Http::fake();

Http::assertSentCount(5);
```

或者，你也可以使用 `assertNothingSent` 方法來斷言在測試過程中沒有發出任何請求：

```
Http::fake();

Http::assertNothingSent();
```

26.5.3.1 記錄請求和響應

你可以使用 recorded 方法來收集所有的請求及其對應的響應。recorded 方法返回一個陣列集合，其中包含了 Illuminate\Http\Client\Request 實例和 Illuminate\Http\Client\Response 實例：

```
Http::fake([
    'https://laravel.com' => Http::response(status: 500),
    'https://nova.laravel.com/' => Http::response(),
]);

Http::get('https://laravel.com');
Http::get('https://nova.laravel.com/');

$recorded = Http::recorded();

[$request, $response] = $recorded[0];
```

此外，recorded 函數也接受一個閉包，該閉包接受一個 Illuminate\Http\Client\Request 和 Illuminate\Http\Client\Response 實例，該閉包可以用來按照你的期望來過濾請求和響應：

```
use Illuminate\Http\Client\Request;
use Illuminate\Http\Client\Response;

Http::fake([
    'https://laravel.com' => Http::response(status: 500),
    'https://nova.laravel.com/' => Http::response(),
]);

Http::get('https://laravel.com');
Http::get('https://nova.laravel.com/');

$recorded = Http::recorded(function (Request $request, Response $response) {
    return $request->url() !== 'https://laravel.com' &&
           $response->successful();
});
```

26.6 事件

Laravel 在發出 HTTP 請求的過程中將會觸發三個事件。在傳送請求前將會觸發 RequestSending 事件，在接收到了指定請求對應的響應時將會觸發 ResponseReceived 事件。如果沒有收到指定請求對應的響應則會觸發 ConnectionFailed 事件。

RequestSending 和 ConnectionFailed 事件都包含一個公共的 $request 屬性，你可以使用它來檢查 Illuminate\Http\Client\Request 實例。 同樣，ResponseReceived 事件包含一個 $request 屬性以及一個 $response 屬性，可用於檢查 Illuminate\Http\Client\Response 實例。 你可以在你的 App\Providers\EventServiceProvider 服務提供者中為這個事件註冊事件監聽器：

```
/**
 * 應用程式的事件偵聽器對應。
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Http\Client\Events\RequestSending' => [
        'App\Listeners\LogRequestSending',
    ],
    'Illuminate\Http\Client\Events\ResponseReceived' => [
        'App\Listeners\LogResponseReceived',
    ],
    'Illuminate\Http\Client\Events\ConnectionFailed' => [
        'App\Listeners\LogConnectionFailed',
    ],
];
```

27 本地化

27.1 簡介

> **技巧** 默認情況下，Laravel 應用程式框架不包含 `lang` 目錄。如果你想自訂 Laravel 的語言檔案，可以通過 `lang:publish` Artisan 命令發佈它們。

Laravel 的本地化功能提供了一種方便的方法來檢索各種語言的字串，從而使你可以輕鬆地在應用程式中支援多種語言。

Laravel 提供了兩種管理翻譯字串的方法。首先，語言字串可以儲存在 `lang` 目錄裡的檔案中。在此目錄中，可能存在應用程式支援的每種語言的子目錄。這是 Laravel 用於管理內建 Laravel 功能（例如驗證錯誤消息）的翻譯字串的方法：

```
/lang
    /en
        messages.php
    /es
        messages.php
```

或者，可以在 `lang` 目錄中放置的 JSON 檔案中定義翻譯字串。採用這種方法時，應用程式支援的每種語言在此目錄中都會有一個對應的 JSON 檔案。對於具有大量可翻譯字串的應用，建議使用此方法：

```
/lang
    en.json
    es.json
```

我們將在本文件中討論每種管理翻譯字串的方法。

27.1.1 發佈語言檔案

默認情況下，Laravel 應用程式框架不包含 `lang` 目錄。如果你想自訂 Laravel 的語言檔案或建立自己的語言檔案，則應通過 `lang:publish` Artisan 命令建構 `lang` 目錄。 `lang:publish` 命令將在應用程式中建立 `lang` 目錄，並行布 Laravel 使用的默認語言檔案集：

```
php artisan lang:publish
```

27.1.2 組態語言環境

應用程式的默認語言儲存在 `config/app.php` 組態檔案的 `locale` 組態選項中。你可以隨意修改此值以適合你的應用程式的需求。

你可以使用 `App` Facade 提供的 `setLocale` 方法，在執行階段通過單個 HTTP 請求修改默認語言：

```
use Illuminate\Support\Facades\App;

Route::get('/greeting/{locale}', function (string $locale) {
    if (! in_array($locale, ['en', 'es', 'fr'])) {
        abort(400);
    }

    App::setLocale($locale);

    // ...
});
```

你可以組態一個 「備用語言」，噹噹前語言不包含給定的翻譯字串時，將使用該語言。和默認語言一樣，備用語言也是在 config/app.php 組態檔案中組態的。

```
'fallback_locale' => 'en',
```

27.1.2.1 確定當前的語言環境

你可以使用 `currentLocale` 和 `isLocale` 方法來確定當前的 `locale` 或檢查 `locale` 是否是一個給定值。

```
use Illuminate\Support\Facades\App;

$locale = App::currentLocale();

if (App::isLocale('en')) {
    // ...
}
```

27.1.3 多語種

你可以使用 Laravel 的「pluralizer」來使用英語以外的語言，Eloquent 和框架的其他部分使用它來將單數字字串轉為複數字串。這可以通過呼叫應用程式服務提供的 `boot` 方法中的 `useLanguage` 方法來實現。複數器目前支援的語言有 **法語**, **挪威語**, **葡萄牙語**, **西班牙語**, **土耳其語**:

```
use Illuminate\Support\Pluralizer;

/**
 * 引導任何應用程式服務。
 */
public function boot(): void
{
    Pluralizer::useLanguage('spanish');

    // ...
}
```

> **注意** 如果你想自訂 pluralizer 的語言，則應該明確定義 Elquent 模型的 表名。

27.2 定義翻譯字串

27.2.1 使用短鍵

通常，翻譯字串儲存在 `lang` 目錄中的檔案中。在這個目錄中，應用程式支援的每種語言都應該有一個子目錄。這是 Laravel 用於管理內建 Laravel 功能（如驗證錯誤消息）的翻譯字串的方法：

```
/lang
    /en
        messages.php
    /es
        messages.php
```

所有的語言檔案都會返回一個鍵值對陣列。比如下方這個例子：

```
<?php

// lang/en/messages.php

return [
    'welcome' => 'Welcome to our application!',
];
```

技巧 對於不同地區的語言，應根據 ISO 15897 命名語言目錄。例如，英式英語應使用「en_GB」而不是「en_gb」。

27.2.2 使用翻譯字串作為鍵

對於具有大量可翻譯字串的應用程式，在檢視中引用鍵時，使用「短鍵」定義每個字串可能會令人困惑，並且為應用程式支援的每個翻譯字串不斷髮明鍵會很麻煩。

出於這個原因，Laravel 還支援使用字串的「默認」翻譯作為鍵來定義翻譯字串。使用翻譯字串作為鍵的翻譯檔案作為 JSON 檔案儲存在 `lang` 目錄中。例如，如果你的應用程式有西班牙語翻譯，你應該建立一個 `lang/es.json` 檔案：

```
{
    "I love programming.": "Me encanta programar."
}
```

27.2.2.1 鍵 / 檔案衝突

你不應該定義和其他翻譯檔案的檔案名稱存在衝突的鍵。例如，在 `nl/action.php` 檔案存在，但 `nl.json` 檔案不存在時，對 `NL` 語言翻譯 `__('Action')` 會導致翻譯器返回 `nl/action.php` 檔案的全部內容。

27.3 檢索翻譯字串

你可以使用 __ 輔助函數從語言檔案中檢索翻譯字串。 如果你使用 「短鍵」 來定義翻譯字串，你應該使用「.」 語法將包含鍵的檔案和鍵本身傳遞給__函數。 例如，讓我們從 `lang/en/messages.php` 語言檔案中檢索 `welcome` 翻譯字串：

```
echo __('messages.welcome');
```

如果指定的翻譯字串不存在，__ 函數將返回翻譯字串鍵。 因此，使用上面的示例，如果翻譯字串不存在，__ 函數將返回 `messages.welcome`。

如果是使用 默認翻譯字串作為翻譯鍵，則應將字串的默認翻譯傳遞給 __ 函數；

```
echo __('I love programming.');
```

同理，如果翻譯字串不存在，__ 函數將返回給定的翻譯字串鍵。

如果是使用的是 Blade 範本引擎，則可以使用 `{{ }}` 語法來顯示翻譯字串：

```
{{ __('messages.welcome') }}
```

27.3.1 替換翻譯字串中的參數

如果願意，可以在翻譯字串中定義預留位置。所有預留位置的前綴都是：。例如，可以使用預留位置名稱定義歡迎消息：

```
'welcome' => 'Welcome, :name',
```

在要檢索翻譯字串時替換預留位置，可以將替換陣列作為第二個參數傳遞給 __ 函數：

```
echo __('messages.welcome', ['name' => 'dayle']);
```

如果預留位置包含所有大寫字母，或僅首字母大寫，則轉換後的值將相應地轉換成大寫：

```
'welcome' => 'Welcome, :NAME', // Welcome, DAYLE
'goodbye' => 'Goodbye, :Name', // Goodbye, Dayle
```

27.3.1.1 對象替換格式

如果試圖提供對象作為轉換預留位置，則將呼叫對象的 `__toString` 方法。`__toString` 方法是 PHP 內建的「神奇方法」之一。然而，有時你可能無法控制給定類的 `__toString` 方法，例如當你正在互動的類屬於第三方庫時。

在這些情況下，Laravel 允許你為特定類型的對象註冊自訂格式處理程序。要實現這一點，你應該呼叫轉換器的 `stringable` 方法。 `stringable` 方法接受閉包，閉包應類型提示其負責格式化的對象類型。通常，應在應用程式的 `AppServiceProvider` 類的 `boot` 方法中呼叫 `stringable` 方法：

```
use Illuminate\Support\Facades\Lang;
use Money\Money;

/**
 * 引導任何應用程式服務。
 */
public function boot(): void
{
    Lang::stringable(function (Money $money) {
        return $money->formatTo('en_GB');
    });
}
```

27.3.2 複數化

因為不同的語言有著各種複雜的複數化規則，所以複數化是個複雜的問題；不過 Laravel 可以根據你定義的複數化規則幫助你翻譯字串。使用 `|` 字元，可以區分字串的單數形式和複數形式：

```
'apples' => 'There is one apple|There are many apples',
```

當然，使用 翻譯字串作為鍵 時也支援複數化：

```
{
    "There is one apple|There are many apples": "Hay una manzana|Hay muchas manzanas"
}
```

你甚至可以建立更複雜的複數化規則，為多個值範圍指定轉換字串：

```
'apples' => '{0} There are none|[1,19] There are some|[20,*] There are many',
```

定義具有複數選項的翻譯字串後，可以使用 `trans_choice` 函數檢索給定「count」的行。在本例中，由於計數大於 1 ，因此返回翻譯字串的複數形式：

```
echo trans_choice('messages.apples', 10);
```

也可以在複數化字串中定義預留位置屬性。通過將陣列作為第三個參數傳遞給 `trans_choice` 函數，可以替換這些預留位置：

```
'minutes_ago' => '{1} :value minute ago|[2,*] :value minutes ago',

echo trans_choice('time.minutes_ago', 5, ['value' => 5]);
```

如果要顯示傳遞給 `trans_choice` 函數的整數值，可以使用內建的 `:count` 預留位置：

```
'apples' => '{0} There are none|{1} There is one|[2,*] There are :count',
```

27.4 覆蓋擴展包的語言檔案

有些包可能隨自己的語言檔案一起封裝。你可以將檔案放置在 `lang/vendor/{package}/{locale}` 目錄中，而不是更改擴展包的核心檔案來調整這些行。

例如，如果需要重寫位於名為 `skyrim/hearthfire` 的包的 `messages.php` 檔案內容，應將語言檔案放在： `lang/vendor/hearthfire/en/messages.php` 在這個檔案中，你應該只定義要覆蓋的翻譯字串。

任何未重寫的翻譯字串仍將從包的原始語言檔案中載入。

28 郵件

28.1 介紹

傳送郵件並不複雜。Laravel 基於 Symfony Mailer 元件提供了一個簡潔、簡單的郵件 API。Laravel 和 Symfony 為 Mailer SMTP 、Mailgun 、Postmark 、 Amazon SES 、 及 sendmail （傳送郵件的方式）提供驅動，允許你通過本地或者云服務來快速傳送郵件。

28.1.1 組態

Laravel 的郵件服務可以通過 `config/mail.php` 組態檔案進行組態。郵件中的每一項都在組態檔案中有單獨的組態項，甚至是獨有的「傳輸方式」，允許你的應用使用不同的郵件服務傳送郵件。例如，你的應用程式在使用 Amazon SES 傳送批次郵件時，也可以使用 Postmark 傳送事務性郵件。

在你的 `mail` 組態檔案中，你將找到 `mailers` 組態陣列。該陣列包含 Laravel 支援的每個郵件 驅動程式 / 傳輸方式 組態，而 `default` 組態值確定當你的應用程式需要傳送電子郵件時，默認情況下將使用哪個郵件驅動。

28.1.2 驅動 / 傳輸的前提

基於 API 的驅動，如 Mailgun 和 Postmark ，通常比 SMTP 伺服器更簡單快速。如果可以的話， 我們建議你使用下面這些驅動。

28.1.2.1 Mailgun 驅動

要使用 Mailgun 驅動，可以先通過 `composer` 來安裝 `Mailgun` 函數庫 ：

```
composer require symfony/mailgun-mailer symfony/http-client
```

接著，在應用的 `config/mail.php` 組態檔案中，將默認項設定成 `mailgun`。組態好之後，確認 `config/services.php` 組態檔案中包含以下選項：

```
'mailgun' => [
    'domain' => env('MAILGUN_DOMAIN'),
    'secret' => env('MAILGUN_SECRET'),
],
```

如果不使用 US Mailgun region 區域終端 ，你需要在 `service` 檔案中組態區域終端：

```
'mailgun' => [
    'domain' => env('MAILGUN_DOMAIN'),
    'secret' => env('MAILGUN_SECRET'),
    'endpoint' => env('MAILGUN_ENDPOINT', 'api.eu.mailgun.net'),
],
```

28.1.2.2 Postmark 驅動

要使用 `Postmark` 驅動，先通過 `composer` 來安裝 `Postmark` 函數庫：

```
composer require symfony/postmark-mailer symfony/http-client
```

接著，在應用的 `config/mail.php` 組態檔案中，將默認項設定成 `postmark`。組態好之後，確認 `config/services.php` 組態檔案中包含如下選項：

```
'postmark' => [
```

```
    'token' => env('POSTMARK_TOKEN'),
],
```

如果你要給指定郵件程序使用的 Postmark message stream，可以在組態陣列中新增 `message_stream_id` 組態選項。這個組態陣列在應用程式的 config/mail.php 組態檔案中：

```
'postmark' => [
    'transport' => 'postmark',
    'message_stream_id' => env('POSTMARK_MESSAGE_STREAM_ID'),
],
```

這樣，你還可以使用不同的 `message stream` 來設定多個 `Postmark 郵件驅動`。

28.1.2.3 SES 驅動

要使用 `Amazon SES` 驅動，你必須先安裝 `PHP` 的 `Amazon AWS SDK`。你可以可以通過 Composer 軟體包管理器安裝此庫：

```
composer require aws/aws-sdk-php
```

然後，將 `config/mail.php` 組態檔案的 `default` 選項設定成 `ses` 並確認你的 `config/services.php` 組態檔案包含以下選項：

```
'ses' => [
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
],
```

為了通過 session token 來使用 AWS temporary credentials ，你需要響應用的 SES 組態中新增一個 `token` 鍵：

```
'ses' => [
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
    'token' => env('AWS_SESSION_TOKEN'),
],
```

傳送郵件，如果你想傳遞一些 額外的選項 給 AWS SDK 的 `SendEmail` 方法，你可以在 `ses` 組態中定義一個 `options` 陣列：

```
'ses' => [
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
    'options' => [
        'ConfigurationSetName' => 'MyConfigurationSet',
        'EmailTags' => [
            ['Name' => 'foo', 'Value' => 'bar'],
        ],
    ],
],
```

28.1.3 備用組態

有時，已經組態好用於傳送應用程式郵件的外部服務可能已關閉。在這種情況下，定義一個或多個備份郵件傳遞組態非常有用，這些組態將在主傳遞驅動程式關閉時使用。 為此，應該在應用程式的 `mail` 組態檔案中定義一個使用 `failover` 傳輸的郵件程序。應用程式的 `failover` 郵件程序的組態陣列應包含一個 `mailers` 陣列，該陣列引用選擇郵件驅動程式進行傳遞的順序：

```
'mailers' => [
    'failover' => [
        'transport' => 'failover',
        'mailers' => [
```

```
            'postmark',
            'mailgun',
            'sendmail',
        ],
    ],

    // ...
],
```

定義故障轉移郵件程序後，應將此郵件程序設定為應用程式使用的默認郵件程序，方法是將其名稱指定為應用程式 `mail` 組態檔案中 `default` 組態金鑰的值：

```
'default' => env('MAIL_MAILER', 'failover'),
```

28.2 生成 Mailables

在建構 Laravel 應用程式時，應用程式傳送的每種類型的電子郵件都表示為一個 `mailable` 類。這些類儲存在 app/Mail 目錄中。 如果你在應用程式中看不到此目錄，請不要擔心，因為它會在你使用 make:mail Artisan 命令建立第一個郵件類時自然生成：

```
php artisan make:mail OrderShipped
```

28.3 編寫 Mailables

一旦生成了一個郵件類，就打開它，這樣我們就可以探索它的內容。郵件類的組態可以通過幾種方法完成，包括 `envelope`、`content` 和 `attachments` 方法。

`envelope` 方法返回 `Illuminate\Mail\Mailables\Envelope` 對象，該對象定義郵件的主題，有時還定義郵件的收件人。`content` 方法返回 `Illuminate\Mail\Mailables\Content` 對象，該對象定義將用於生成消息內容的 Blade 範本。

28.3.1 組態發件人

28.3.1.1 使用 Envelope

首先，讓我們來看下如何組態電子郵件的發件人。電子郵件的「發件人」。有兩種方法可以組態傳送者。首先，你可以在郵件信封上指定「發件人」地址：

```
use Illuminate\Mail\Mailables\Address;
use Illuminate\Mail\Mailables\Envelope;

/**
 * 獲取郵件信封。
 */
public function envelope(): Envelope
{
    return new Envelope(
        from: new Address('jeffrey@example.com', 'Jeffrey Way'),
        subject: '訂單發貨',
    );
}
```

除此之外，還可以指定 `replyTo` 地址：

```
return new Envelope(
    from: new Address('jeffrey@example.com', 'Jeffrey Way'),
    replyTo: [
        new Address('taylor@example.com', 'Taylor Otwell'),
```

```
    ],
    subject: '訂單發貨',
);
```

28.3.1.2 使用全域 `from` 地址

當然，如果你的應用在任何郵件中使用的「發件人」地址都一致的話，在你生成的每一個 mailable 類中呼叫 `from` 方法可能會很麻煩。因此，你可以在 `config/mail.php` 檔案中指定一個全域的「發件人」地址。當某個 mailable 類沒有指定「發件人」時，它將使用該全域「發件人」：

```
'from' => ['address' => 'example@example.com', 'name' => 'App Name'],
```

此外，你可以在 `config/mail.php` 組態檔案中定義全域 「reply_to」 地址：

```
'reply_to' => ['address' => 'example@example.com', 'name' => 'App Name'],
```

28.3.2 組態檢視

在郵件類下的 `content` 方法中使用 `view` 方法來指定在渲染郵件內容時要使用的範本。由於每封電子郵件通常使用一個 Blade 範本 來渲染其內容。因此在建構電子郵件的 HTML 時，可以充分利用 Blade 範本引擎的功能和便利性：

```
/**
 * 獲取消息內容定義。
 */
public function content(): Content
{
    return new Content(
        view: 'emails.orders.shipped',
    );
}
```

> **技巧** 你可以建立一個 `resources/views/emails` 目錄來存放所有的郵件範本；當然，也可以將其置於 `resources/views` 目錄下的任何位置。

28.3.2.1 純文字郵件

如果要定義電子郵件的純文字版本，可以在建立郵件的 `Content` 定義時指定純文字範本。與 `view` 參數一樣， `text` 參數是用於呈現電子郵件內容的範本名稱。這樣你就可以自由定義郵件的 Html 和純文字版本：

```
/**
 * 獲取消息內容定義。
 */
public function content(): Content
{
    return new Content(
        view: 'emails.orders.shipped',
        text: 'emails.orders.shipped-text'
    );
}
```

為了清晰，`html` 參數可以用作 `view` 參數的別名：

```
return new Content(
    html: 'emails.orders.shipped',
    text: 'emails.orders.shipped-text'
);
```

28.3.3 檢視數

28.3.3.1 通過 Public 屬性

通常，你需要將一些資料傳遞給檢視，以便在呈現電子郵件的 HTML 時使用。有兩種方法可以使資料對檢視可用。首先，在 mailable 類上定義的任何公共屬性都將自動對檢視可用。例如，可以將資料傳遞到可郵寄類的建構函式中，並將該資料設定為類上定義的公共方法：

```
<?php

namespace App\Mail;

use App\Models\Order;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Mail\Mailables\Content;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * 建立新的消息實例。
     */
    public function __construct(
        public Order $order,
    ) {}

    /**
     * 獲取消息內容定義。
     */
    public function content(): Content
    {
        return new Content(
            view: 'emails.orders.shipped',
        );
    }
}
```

一旦資料設定為公共屬性，它將自動在檢視中可用，因此可以像訪問 Blade 範本中的任何其他資料一樣訪問它：

```
<div>
    Price: {{ $order->price }}
</div>
```

28.3.3.2 通過 `with` 參數：

如果你想要在郵件資料傳送到範本前自訂它們的格式，你可以使用 `with` 方法來手動傳遞資料到檢視中。一般情況下，你還是需要通過 mailable 類的建構函式來傳遞資料；不過，你應該將它們定義為 `protected` 或 `private` 以防止它們被自動傳遞到檢視中。然後，在呼叫 `with` 方法的時候，可以以陣列的形式傳遞你想要傳遞給範本的資料：

```
<?php

namespace App\Mail;

use App\Models\Order;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Mail\Mailables\Content;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;
```

```
    /**
     * 建立新的消息實例。
     */
    public function __construct(
        protected Order $order,
    ) {}

    /**
     * 獲取消息內容定義。
     */
    public function content(): Content
    {
        return new Content(
            view: 'emails.orders.shipped',
            with: [
                'orderName' => $this->order->name,
                'orderPrice' => $this->order->price,
            ],
        );
    }
}
```

一旦資料被傳遞到 `with` 方法，同樣的它將自動在檢視中可用，因此可以像訪問 Blade 範本中的任何其他資料一樣訪問它：

```
<div>
    Price: {{ $orderPrice }}
</div>
```

28.3.4 附件

要向電子郵件新增附件，你將向郵件的 `attachments` 方法返回的陣列新增附件。首先，可以通過向 `Attachment` 類提供的 `fromPath` 方法提供檔案路徑來新增附件：

```
use Illuminate\Mail\Mailables\Attachment;

/**
 * 獲取郵件的附件。
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
 */
public function attachments(): array
{
    return [
        Attachment::fromPath('/path/to/file'),
    ];
}
```

將檔案附加到郵件時，還可以使用 `as` 和 `withMime` 方法來指定附件的顯示名稱 / 或 MIME 類型：

```
/**
 * 獲取郵件的附件。
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
 */
public function attachments(): array
{
    return [
        Attachment::fromPath('/path/to/file')
                ->as('name.pdf')
                ->withMime('application/pdf'),
    ];
}
```

28.3.4.1 從磁碟中新增附件

如果你已經在 檔案儲存 上儲存了一個檔案，則可以使用 `attachFromStorage` 方法將其附加到郵件中：

```
/**
 * 獲取郵件的附件。
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
 */
public function attachments(): array
{
    return [
        Attachment::fromStorage('/path/to/file'),
    ];
}
```

當然，也可以指定附件的名稱和 MIME 類型：

```
/**
 * 獲取郵件的附件。
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
 */
public function attachments(): array
{
    return [
        Attachment::fromStorage('/path/to/file')
                ->as('name.pdf')
                ->withMime('application/pdf'),
    ];
}
```

如果需要指定默認磁碟以外的儲存磁碟，可以使用 `attachFromStorageDisk` 方法：

```
/**
 * 獲取郵件的附件。
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
 */
public function attachments(): array
{
    return [
        Attachment::fromStorageDisk('s3', '/path/to/file')
                ->as('name.pdf')
                ->withMime('application/pdf'),
    ];
}
```

28.3.4.2 原始資料附件

`fromData` 附件方法可用於附加原始位元組字串作為附件。例如，如果你在記憶體中生成了 PDF，並且希望將其附加到電子郵件而不將其寫入磁碟，可以使用到此方法。 `fromData` 方法接受一個閉包，該閉包解析原始資料位元組以及應分配給附件的名稱：

```
/**
 * 獲取郵件的附件。
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
 */
public function attachments(): array
{
    return [
        Attachment::fromData(fn () => $this->pdf, 'Report.pdf')
                ->withMime('application/pdf'),
    ];
```

```
}
```

28.3.5 內聯附件

在郵件中嵌入內聯圖片通常很麻煩；不過，Laravel 提供了一種將圖像附加到郵件的便捷方法。可以使用郵件範本中 $message 變數的 embed 方法來嵌入內聯圖片。Laravel 自動使 $message 變數在全部郵件範本中可用，不需要擔心手動傳遞它：

```
<body>
    這是一張圖片：

    <img src="{{ $message->embed($pathToImage) }}">
</body>
```

注意 該 `$message` 在文字消息中不可用，因為文字消息不能使用內聯附件。

28.3.5.1 嵌入原始資料附件

如果你已經有了可以嵌入郵件範本的原始圖像資料字串，可以使用 `$message` 變數的 `embedData` 方法，當呼叫 `embedData` 方法時，需要傳遞一個檔案名稱：

```
<body>
    以下是原始資料的圖像：

    <img src="{{ $message->embedData($data, 'example-image.jpg') }}">
</body>
```

28.3.6 可附著對象

雖然通過簡單的字串路徑將檔案附加到消息通常就足夠了，但在多數情況下，應用程式中的可附加實體由類表示。例如，如果你的應用程式正在將照片附加到消息中，那麼在應用中可能還具有表示該照片的 `Photo` 模型。在這種情況下，簡單地將 `Photo` 模型傳遞給 `attach` 方法會很方便。

開始時，在可附加到郵件的對象上實現 `Illuminate\Contracts\Mail\Attachable` 介面。此介面要求類定義一個 `toMailAttachment` 方法，該方法返回一個 `Illuminate\Mail\Attachment` 實例：

```
<?php

namespace App\Models;

use Illuminate\Contracts\Mail\Attachable;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Mail\Attachment;

class Photo extends Model implements Attachable
{
    /**
     * 獲取模型的可附加表示。
     */
    public function toMailAttachment(): Attachment
    {
        return Attachment::fromPath('/path/to/file');
    }
}
```

一旦定義了可附加對象，就可以在生成電子郵件時從 `attachments` 方法返回該對象的實例：

```
/**
 * 獲取郵件的附件。
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
```

```
     */
    public function attachments(): array
    {
        return [$this->photo];
    }
```

當然，附件資料可以儲存在遠端檔案儲存服務（例如 Amazon S3）上。因此，Laravel 還允許你從儲存在應用程式 檔案系統磁碟 上的資料生成附件實例：

```
// 從默認磁碟上的檔案建立附件。。。
return Attachment::fromStorage($this->path);

// 從特定磁碟上的檔案建立附件。。。
return Attachment::fromStorageDisk('backblaze', $this->path);
```

此外，還可以通過記憶體中的資料建立附件實例。為此還提供了 `fromData` 方法的閉包。但閉包應返回表示附件的原始資料：

```
return Attachment::fromData(fn () => $this->content, 'Photo Name');
```

Laravel 還提供了其他方法，你可以使用這些方法自訂附件。例如，可以使用 `as` 和 `withMime` 方法自訂檔案名稱和 MIME 類型：

```
return Attachment::fromPath('/path/to/file')
        ->as('Photo Name')
        ->withMime('image/jpeg');
```

28.3.7 標頭

有時，你可能需要在傳出消息中附加附加的標頭。例如，你可能需要設定自訂 `Message-Id` 或其他任意文字標題。

如果要實現這一點，請在郵件中定義 `headers` 方法。 `headers` 方法應返回 `Illuminate\Mail\Mailables\Headers` 實例。此類接受 `messageId` 、 `references` 和 `text` 參數。當然，你可以只提供特定消息所需的參數：

```
use Illuminate\Mail\Mailables\Headers;

/**
 * 獲取郵件標題。
 */
public function headers(): Headers
{
    return new Headers(
        messageId: 'custom-message-id@example.com',
        references: ['previous-message@example.com'],
        text: [
            'X-Custom-Header' => 'Custom Value',
        ],
    );
}
```

28.3.8 標記 和 中繼資料

一些第三方電子郵件提供商（如 Mailgun 和 Postmark）支援消息「標籤」和 「中繼資料」，可用於對應用程式傳送的電子郵件進行分組和跟蹤。你可以通過 `Envelope` 來定義向電子郵件新增標籤和中繼資料：

```
use Illuminate\Mail\Mailables\Envelope;

/**
 * 獲取郵件信封。
 *
 * @return \Illuminate\Mail\Mailables\Envelope
```

```
     */
    public function envelope(): Envelope
    {
        return new Envelope(
            subject: '訂單發貨',
            tags: ['shipment'],
            metadata: [
                'order_id' => $this->order->id,
            ],
        );
    }
```

如果你的應用程式正在使用 Mailgun 驅動程式，你可以查閱 Mailgun 的文件以獲取有關 標籤 和 中繼資料 的更多資訊。同樣，還可以查閱郵戳文件，瞭解其對 標籤 和 中繼資料 支援的更多資訊

如果你的應用程式使用 Amazon SES 傳送電子郵件，則應使用 `metadata` 方法將 SES 「標籤」附加到郵件中。

28.3.9 自訂 Symfony 消息

Laravel 的郵件功能是由 Symfony Mailer 提供的。Laravel 在你傳送消息之前是由 Symfony Message 註冊然後再去呼叫自訂實例。這讓你有機會在傳送郵件之前對其進行深度定製。為此，請在 `Envelope` 定義上定義 `using` 參數：

```
use Illuminate\Mail\Mailables\Envelope;
use Symfony\Component\Mime\Email;

/**
 * 獲取郵件信封。
 */
public function envelope(): Envelope
{
    return new Envelope(
        subject: '訂單發貨',
        using: [
            function (Email $message) {
                // ...
            },
        ]
    );
}
```

28.4 Markdown 格式郵件

Markdown 格式郵件允許你可以使用 mailable 中的預建構範本和 郵件通知 元件。由於消息是用 Markdown 編寫，Laravel 能夠渲染出美觀的、響應式的 HTML 範本消息，同時還能自動生成純文字副本。

28.4.1 生成 Markdown 郵件

你可以在執行 `make:mail` 的 Artisan 命令時使用 `--markdown` 選項來生成一個 Markdown 格式範本的 mailable 類：

```
php artisan make:mail OrderShipped --markdown=emails.orders.shipped
```

然後，在 `Content` 方法中組態郵寄的 `content` 定義時，使用 `markdown` 參數而不是 `view` 參數：

```
use Illuminate\Mail\Mailables\Content;

/**
 * 獲取消息內容定義。
 */
```

```
public function content(): Content
{
    return new Content(
        markdown: 'emails.orders.shipped',
        with: [
            'url' => $this->orderUrl,
        ],
    );
}
```

28.4.2 編寫 Markdown 郵件

Markdown mailable 類整合了 Markdown 語法和 Blade 元件，讓你能夠非常方便的使用 Laravel 預置的 UI 元件來建構郵件消息：

```
<x-mail::message>
# 訂單發貨

你的訂單已發貨！

<x-mail::button :url="$url">
查看訂單
</x-mail::button>

謝謝,<br>
{{ config('app.name') }}
</x-mail::message>
```

> **技巧** 在編寫 Markdown 郵件的時候，請勿使用額外的縮排。Markdown 解析器會把縮排渲染成程式碼塊。

28.4.2.1 按鈕元件

按鈕元件用於渲染居中的按鈕連結。該元件接收兩個參數，一個是 `url` 一個是可選的 `color`。 支援的顏色包括 `primary`，`success` 和 `error`。你可以在郵件中新增任意數量的按鈕元件：

```
<x-mail::button :url="$url" color="success">
查看訂單
</x-mail::button>
```

28.4.2.2 面板元件

面板元件在面板內渲染指定的文字塊，面板與其他消息的背景色略有不同。它允許你繪製一個警示文字塊：

```
<x-mail::panel>
這是面板內容
</x-mail::panel>
```

28.4.2.3 表格元件

表格元件允許你將 Markdown 表格轉換成 HTML 表格。該元件接受 Markdown 表格作為其內容。列對齊支援默認的 Markdown 表格對齊語法：

```
<x-mail::table>
| Laravel       | Table         | Example  |
| ------------- |:-------------:| --------:|
| Col 2 is      | Centered      | $10      |
| Col 3 is      | Right-Aligned | $20      |
</x-mail::table>
```

28.4.3 自訂元件

你可以將所有 Markdown 郵件元件匯出到自己的應用，用作自訂元件的範本。若要匯出元件，使用 `laravel-mail` 資產標籤的 `vendor:publish` Artisan 命令：

```
php artisan vendor:publish --tag=laravel-mail
```

此命令會將 Markdown 郵件元件匯出到 `resources/views/vendor/mail` 目錄。 該 `mail` 目錄包含 `html` 和 `text` 子目錄，分別包含各自對應的可用元件描述。你可以按照自己的意願自訂這些元件。

28.4.3.1 自訂 CSS

元件匯出後，`resources/views/vendor/mail/html/themes` 目錄有一個 `default.css` 檔案。可以在此檔案中自訂 CSS，這些樣式將自動內聯到 Markdown 郵件消息的 HTML 表示中。

如果想為 Laravel 的 Markdown 元件建構一個全新的主題，你可以在 `html/themes` 目錄中新建一個 CSS 檔案。命名並保存 CSS 檔案後，並更新應用程式 `config/mail.php` 組態檔案的 `theme` 選項以匹配新主題的名稱。

要想自訂單個郵件主題，可以將 mailable 類的 `$theme` 屬性設定為傳送 mailable 時應使用的主題名稱。

28.5 傳送郵件

若要傳送郵件，使用 `Mail` facade 的方法。該 `to` 方法接受 郵件地址、使用者實例或使用者集合。如果傳遞一個對象或者對象集合，mailer 在設定收件人時將自動使用它們的 `email` 和 `name` 屬性，因此請確保對象的這些屬性可用。一旦指定了收件人，就可以將 mailable 類實例傳遞給 `send` 方法：

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Mail\OrderShipped;
use App\Models\Order;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Mail;

class OrderShipmentController extends Controller
{
    /**
     * 傳送給定的訂單資訊。
     */
    public function store(Request $request): RedirectResponse
    {
        $order = Order::findOrFail($request->order_id);

        // 發貨訂單。。。

        Mail::to($request->user())->send(new OrderShipped($order));

        return redirect('/orders');
    }
}
```

在傳送消息時不止可以指定收件人。還可以通過鏈式呼叫「to」、「cc」、「bcc」一次性指定抄送和密送收件人：

```
Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->send(new OrderShipped($order));
```

28.5.1.1 遍歷收件人列表

有時，你需要通過遍歷一個收件人 / 郵件地址陣列的方式，給一系列收件人傳送郵件。但是，由於 to 方法會給 mailable 列表中的收件人追加郵件地址，因此，你應該為每個收件人重建 mailable 實例。

```
foreach (['taylor@example.com', 'dries@example.com'] as $recipient) {
    Mail::to($recipient)->send(new OrderShipped($order));
}
```

28.5.1.2 通過特定的 Mailer 傳送郵件

默認情況下，Laravel 將使用 mail 你的組態檔案中組態為 default 郵件程序。 但是，你可以使用 mailer 方法通過特定的郵件程序組態傳送：

```
Mail::mailer('postmark')
        ->to($request->user())
        ->send(new OrderShipped($order));
```

28.5.2 郵件佇列

28.5.2.1 將郵件消息加入佇列

由於傳送郵件消息可能大幅度延長應用的響應時間，許多開發者選擇將郵件消息加入佇列放在後台傳送。Laravel 使用內建的 統一佇列 API 簡化了這一工作。若要將郵件消息加入佇列，可以在指定消息的接收者後，使用 Mail 門面的 queue 方法：

```
Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->queue(new OrderShipped($order));
```

此方法自動將作業推送到佇列中以便消息在後台傳送。使用此特性之前，需要 組態佇列 。

28.5.2.2 延遲消息佇列

想要延遲傳送佇列化的郵件消息，可以使用 later 方法。該 later 方法的第一個參數的第一個參數是標示消息何時傳送的 DateTime 實例：

```
Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->later(now()->addMinutes(10), new OrderShipped($order));
```

28.5.2.3 推送到指定佇列

由於所有使用 make:mail 命令生成的 mailable 類都是用了 Illuminate\Bus\Queueable trait，因此你可以在任何 mailable 類實例上呼叫 onQueue 和 onConnection 方法來指定消息的連接和佇列名：

```
$message = (new OrderShipped($order))
                ->onConnection('sqs')
                ->onQueue('emails');

Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->queue($message);
```

28.5.2.4 默認佇列

如果你希望你的郵件類始終使用佇列，你可以給郵件類實現 `ShouldQueue` 契約，現在即使你呼叫了 `send` 方法，郵件依舊使用佇列的方式傳送

```
use Illuminate\Contracts\Queue\ShouldQueue;

class OrderShipped extends Mailable implements ShouldQueue
{
    // ...
}
```

28.5.2.5 佇列的郵件和資料庫事務

當在資料庫事務中分發郵件佇列時，佇列可能在資料庫事務提交之前處理郵件。 發生這種情況時，在資料庫事務期間對模型或資料庫記錄所做的任何更新可能都不會反映在資料庫中。另外，在事務中建立的任何模型或資料庫記錄都可能不存在於資料庫中。如果你的郵件基於以上這些模型資料，則在處理郵件傳送時，可能會發生意外錯誤。

如果佇列連接的 `after_commit` 組態選項設定為 `false`，那麼仍然可以通過在 mailable 類上定義 `afterCommit` 屬性來設定提交所有打開的資料庫事務之後再調度特定的郵件佇列：

```
Mail::to($request->user())->send(
    (new OrderShipped($order))->afterCommit()
);
```

或者，你可以 `afterCommit` 從 mailable 的建構函式中呼叫該方法：

```
<?php

namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable implements ShouldQueue
{
    use Queueable, SerializesModels;

    /**
     * 建立新的消息實例。
     */
    public function __construct()
    {
        $this->afterCommit();
    }
}
```

> **技巧** 要瞭解有關解決這些問題的更多資訊，請查看 佇列和資料庫事物。

28.6 渲染郵件

有時你可能希望捕獲郵件的 HTML 內容而不傳送它。為此，可以呼叫郵件類的 `render` 方法。此方法將以字串形式返回郵件類的渲染內容:

```
use App\Mail\InvoicePaid;
use App\Models\Invoice;

$invoice = Invoice::find(1);
```

```
return (new InvoicePaid($invoice))->render();
```

28.6.1 在瀏覽器中預覽郵件

設計郵件範本時，可以方便地在瀏覽器中預覽郵件，就像典型的 Blade 範本一樣。因此， Laravel 允許你直接從路由閉包或 controller 返回任何郵件類。當郵件返回時，它將渲染並顯示在瀏覽器中，允許你快速預覽其設計，而無需將其傳送到實際的電子郵件地址：

```
Route::get('/mailable', function () {
    $invoice = App\Models\Invoice::find(1);

    return new App\Mail\InvoicePaid($invoice);
});
```

> **注意** 在瀏覽器中預覽郵件時，不會呈現 內聯附件 要預覽這些郵件，你應該將它們傳送到電子郵件測試應用程式，例如 Mailpit 或 HELO。

28.7 本地化郵件

Laravel 允許你在請求的當前語言環境之外的語言環境中傳送郵件，如果郵件在排隊，它甚至會記住這個語言環境。

為此， `Mail` 門面提供了一個 `locale` 方法來設定所需的語言。評估可郵寄的範本時，應用程式將更改為此語言環境，然後在評估完成後恢復為先前的語言環境：

```
Mail::to($request->user())->locale('es')->send(
    new OrderShipped($order)
);
```

28.7.1 使用者首選語言環境

有時，應用程式儲存每個使用者的首選語言環境。 通過在一個或多個模型上實現 `HasLocalePreference` 契約，你可以指示 Laravel 在傳送郵件時使用這個儲存的語言環境：

```
use Illuminate\Contracts\Translation\HasLocalePreference;

class User extends Model implements HasLocalePreference
{
    /**
     * 獲取使用者的區域設定。
     */
    public function preferredLocale(): string
    {
        return $this->locale;
    }
}
```

一旦你實現了介面，Laravel 將在向模型傳送郵件和通知時自動使用首選語言環境。 因此，使用該介面時無需呼叫 `locale` 方法：

```
Mail::to($request->user())->send(new OrderShipped($order));
```

28.8 測試郵件

28.8.1 測試郵件內容

Laravel 提供了幾種方便的方法來測試你的郵件是否包含你期望的內容。 這些方法是：`assertSeeInHtml`、`assertDontSeeInHtml`、`assertSeeInOrderInHtml`、`assertSeeInText`、`assertDontSeeInText` 和 `assertSeeInOrderInText`。

和你想的一樣，「HTML」判斷你的郵件的 HTML 版本中是否包含給定字串，而「Text」是判斷你的可郵寄郵件的純文字版本是否包含給定字串：

```
use App\Mail\InvoicePaid;
use App\Models\User;

public function test_mailable_content(): void
{
    $user = User::factory()->create();

    $mailable = new InvoicePaid($user);

    $mailable->assertFrom('jeffrey@example.com');
    $mailable->assertTo('taylor@example.com');
    $mailable->assertHasCc('abigail@example.com');
    $mailable->assertHasBcc('victoria@example.com');
    $mailable->assertHasReplyTo('tyler@example.com');
    $mailable->assertHasSubject('Invoice Paid');
    $mailable->assertHasTag('example-tag');
    $mailable->assertHasMetadata('key', 'value');

    $mailable->assertSeeInHtml($user->email);
    $mailable->assertSeeInHtml('Invoice Paid');
    $mailable->assertSeeInOrderInHtml(['Invoice Paid', 'Thanks']);

    $mailable->assertSeeInText($user->email);
    $mailable->assertSeeInOrderInText(['Invoice Paid', 'Thanks']);

    $mailable->assertHasAttachment('/path/to/file');
    $mailable->assertHasAttachment(Attachment::fromPath('/path/to/file'));
    $mailable->assertHasAttachedData($pdfData, 'name.pdf', ['mime' =>
'application/pdf']);
    $mailable->assertHasAttachmentFromStorage('/path/to/file', 'name.pdf', ['mime' =>
'application/pdf']);
    $mailable->assertHasAttachmentFromStorageDisk('s3', '/path/to/file', 'name.pdf',
['mime' => 'application/pdf']);
}
```

28.8.2 測試郵件的傳送

我們建議將郵件內容和判斷指定的郵件「傳送」給特定使用者的測試分開進行測試。通常來講，郵件的內容與你正在測試的程式碼無關，只要能簡單地判斷 Laravel 能夠傳送指定的郵件就足夠了。

你可以使用 `Mail` 方法的 `fake` 方法來阻止郵件的傳送。呼叫了 `Mail` 方法的 `fake` 方法後，你可以判斷郵件是否已被傳送給指定的使用者，甚至可以檢查郵件收到的資料：

```
<?php

namespace Tests\Feature;

use App\Mail\OrderShipped;
use Illuminate\Support\Facades\Mail;
```

```
use Tests\TestCase;

class ExampleTest extends TestCase
{
     public function test_orders_can_be_shipped(): void
     {
          Mail::fake();

          // 執行郵件傳送。。。

          // 判斷沒有傳送的郵件。。。
          Mail::assertNothingSent();

          // 判斷已傳送郵件。。。
          Mail::assertSent(OrderShipped::class);

          // 判斷已傳送兩次的郵件。。。
          Mail::assertSent(OrderShipped::class, 2);

          // 判斷郵件是否未傳送。。。
          Mail::assertNotSent(AnotherMailable::class);
     }
}
```

如果你在後台排隊等待郵件的傳遞，則應該使用 `assertQueued` 方法而不是 `assertSent` 方法。

```
Mail::assertQueued(OrderShipped::class);

Mail::assertNotQueued(OrderShipped::class);

Mail::assertNothingQueued();
```

你可以向 `assertSent`、`assertNotSent`、 `assertQueued` 或者 `assertNotQueued` 方法來傳遞閉包，來判斷髮送的郵件是否通過給定的 「真值檢驗」。如果至少傳送了一個可以通過的郵件，就可以判斷為成功。

```
Mail::assertSent(function (OrderShipped $mail) use ($order) {
     return $mail->order->id === $order->id;
});
```

當呼叫 `Mail` 外觀的判斷方法時，提供的閉包所接受的郵件實例會公開檢查郵件的可用方法：

```
Mail::assertSent(OrderShipped::class, function (OrderShipped $mail) use ($user) {
     return $mail->hasTo($user->email) &&
             $mail->hasCc('...') &&
             $mail->hasBcc('...') &&
             $mail->hasReplyTo('...') &&
             $mail->hasFrom('...') &&
             $mail->hasSubject('...');
});
```

mailable 實例還包括檢查 mailable 上的附件的幾種可用方法：

```
use Illuminate\Mail\Mailables\Attachment;

Mail::assertSent(OrderShipped::class, function (OrderShipped $mail) {
     return $mail->hasAttachment(
          Attachment::fromPath('/path/to/file')
                    ->as('name.pdf')
                    ->withMime('application/pdf')
     );
});

Mail::assertSent(OrderShipped::class, function (OrderShipped $mail) {
     return $mail->hasAttachment(
          Attachment::fromStorageDisk('s3', '/path/to/file')
     );
});
```

```
Mail::assertSent(OrderShipped::class, function (OrderShipped $mail) use ($pdfData) {
    return $mail->hasAttachment(
        Attachment::fromData(fn () => $pdfData, 'name.pdf')
    );
});
```

你可能已經注意到，有 2 種方法可以判斷郵件是否傳送, 即：`assertNotSent` 和 `assertNotQueued` 。有時你可能希望判斷郵件沒有被傳送**或**排隊。如果要實現這一點，你可以使用 `assertNothingOutgoing` 和 `assertNotOutgoing` 方法。

```
Mail::assertNothingOutgoing();

Mail::assertNotOutgoing(function (OrderShipped $mail) use ($order) {
    return $mail->order->id === $order->id;
});
```

28.9 郵件和本地開發

在開發傳送電子郵件的應用程式時，你可能不希望實際將電子郵件傳送到實際的電子郵件地址。 Laravel 提供了幾種在本地開發期間「停用」傳送電子郵件的方法。

28.9.1.1 日誌驅動

`log` 郵件驅動程式不會傳送你的電子郵件，而是將所有電子郵件資訊寫入你的記錄檔以供檢查。 通常，此驅動程式僅在本地開發期間使用。有關按環境組態應用程式的更多資訊，請查看 組態文件。

28.9.1.2 HELO / Mailtrap / Mailpit

或者，你可以使用 HELO 或 Mailtrap 之類的服務和 `smtp` 驅動程式將你的電子郵件資訊傳送到「虛擬」信箱。你可以通過在真正的電子郵件客戶端中查看它們。這種方法的好處是允許你在 Mailtrap 的消息查看實際並檢查的最終電子郵件。

如果你使用 Laravel Sail，你可以使用 Mailpit 預覽你的消息。當 Sail 執行階段，你可以訪問 Mailpit 介面：`http://localhost:8025`。

28.9.1.3 使用全域 `to` 地址

最後，你可以通過呼叫 `Mail` 門面提供的 `alwaysTo` 方法來指定一個全域的「收件人」地址。 通常，應該從應用程式的服務提供者之一的 `boot` 方法呼叫此方法：

```
use Illuminate\Support\Facades\Mail;

/**
 * 啟動應用程式服務
 */
public function boot(): void
{
    if ($this->app->environment('local')) {
        Mail::alwaysTo('taylor@example.com');
    }
}
```

28.10 事件

Laravel 在傳送郵件消息的過程中會觸發 2 個事件。`MessageSending` 事件在消息傳送之前觸發，而

MessageSent 事件在消息傳送後觸發。請記住，這些事件是在郵件**傳送**時觸發的，而不是在排隊時觸發的。你可以在你的 App\Providers\EventServiceProvider 服務中為這個事件註冊事件監聽器：

```
use App\Listeners\LogSendingMessage;
use App\Listeners\LogSentMessage;
use Illuminate\Mail\Events\MessageSending;
use Illuminate\Mail\Events\MessageSent;

/**
 * 應用程式的事件偵聽器對應。
 *
 * @var array
 */
protected $listen = [
    MessageSending::class => [
        LogSendingMessage::class,
    ],

    MessageSent::class => [
        LogSentMessage::class,
    ],
];
```

28.11 自訂傳輸

Laravel 包含多種郵件傳輸；但是，你可能希望編寫自己的傳輸程序，通過 Laravel 來傳送電子郵件。首先，定義一個擴展 Symfony\Component\Mailer\Transport\AbstractTransport 類。然後，在傳輸上實現 doSend 和 __toString() 方法：

```
use MailchimpTransactional\ApiClient;
use Symfony\Component\Mailer\SentMessage;
use Symfony\Component\Mailer\Transport\AbstractTransport;
use Symfony\Component\Mime\Address;
use Symfony\Component\Mime\MessageConverter;

class MailchimpTransport extends AbstractTransport
{
    /**
     * 建立一個新的 Mailchimp 傳輸實例。
     */
    public function __construct(
        protected ApiClient $client,
    ) {
        parent::__construct();
    }

    /**
     * {@inheritDoc}
     */
    protected function doSend(SentMessage $message): void
    {
        $email = MessageConverter::toEmail($message->getOriginalMessage());

        $this->client->messages->send(['message' => [
            'from_email' => $email->getFrom(),
            'to' => collect($email->getTo())->map(function (Address $email) {
                return ['email' => $email->getAddress(), 'type' => 'to'];
            })->all(),
            'subject' => $email->getSubject(),
            'text' => $email->getTextBody(),
        ]]);
    }
```

```
    /**
     * 獲取傳輸字串的表示形式。
     */
    public function __toString(): string
    {
        return 'mailchimp';
    }
}
```

你一旦定義了自訂傳輸，就可以通過 `Mail` 外觀提供的 `boot` 方法來註冊它。通常情況下，這應該在應用程式的 `AppServiceProvider` 服務種提供的 `boot` 方法中完成。`$config` 參數將提供給 `extend` 方法的閉包。該參數將包含在應用程式中的 `config/mail.php` 來組態檔案中為 mailer 定義的組態陣列。

```
use App\Mail\MailchimpTransport;
use Illuminate\Support\Facades\Mail;

/**
 * 啟動應用程式服務
 */
public function boot(): void
{
    Mail::extend('mailchimp', function (array $config = []) {
        return new MailchimpTransport(/* ... */);
    });
}
```

定義並註冊自訂傳輸後，你可以在應用程式中的 `config/mail.php` 組態檔案中新建一個利用自訂傳輸的郵件定義：

```
'mailchimp' => [
    'transport' => 'mailchimp',
    // ...
],
```

28.11.1 額外的 Symfony 傳輸

Laravel 同樣支援一些現有的 Symfony 維護的郵件傳輸，如 Mailgun 和 Postmark 。但是，你可能希望通過擴展 Laravel，來支援 Symfony 維護的其他傳輸。你可以通過 Composer 請求必要的 Symfony 郵件並向 Laravel 註冊運輸。例如，你可以安裝並註冊 Symfony 的「Sendinblue」 郵件：

```
composer require symfony/sendinblue-mailer symfony/http-client
```

安裝 Sendinblue 郵件包後，你可以將 Sendinblue API 憑據的條目新增到應用程式的「服務」組態檔案中：

```
'sendinblue' => [
    'key' => 'your-api-key',
],
```

最後，你可以使用 `Mail` 門面的 `extend` 方法向 Laravel 註冊傳輸。通常，這應該在服務提供者的 `boot` 方法中完成：

```
use Illuminate\Support\Facades\Mail;
use Symfony\Component\Mailer\Bridge\Sendinblue\Transport\SendinblueTransportFactory;
use Symfony\Component\Mailer\Transport\Dsn;

/**
 * 啟動應用程式服務。
 */
public function boot(): void
{
    Mail::extend('sendinblue', function () {
        return (new SendinblueTransportFactory)->create(
            new Dsn(
                'sendinblue+api',
                'default',
                config('services.sendinblue.key')
```

```
            )
        );
    });
}
```

你一旦註冊了傳輸，就可以在應用程式的 `config/mail.php` 組態檔案中建立一個用於新傳輸的 mailer 定義：

```
'sendinblue' => [
    'transport' => 'sendinblue',
    // ...
],
```

29 消息通知

29.1 介紹

除了支援 傳送電子郵件 之外，Laravel 還提供了支援通過多種傳遞管道傳送通知的功能，包括電子郵件、簡訊（通過 Vonage，前身為 Nexmo）和 Slack。此外，已經建立了多種 社區建構的通知管道，用於通過數十個不同的管道傳送通知！通知也可以儲存在資料庫中，以便在你的 Web 介面中顯示。

通常，通知應該是簡短的資訊性消息，用於通知使用者應用中發生的事情。例如，如果你正在編寫一個賬單應用，則可以通過郵件和簡訊頻道向使用者傳送一個「支付憑證」通知。

29.2 建立通知

Laravel 中，通常每個通知都由一個儲存在 `app/Notifications` 目錄下的一個類表示。如果在你的應用中沒有看到這個目錄，不要擔心，當運行 `make:notification` 命令時它將為你建立：

```
php artisan make:notification InvoicePaid
```

這個命令會在 `app/Notifications` 目錄下生成一個新的通知類。每個通知類都包含一個 `via` 方法以及一個或多個消息建構的方法比如 `toMail` 或 `toDatabase`，它們會針對特定的管道把通知轉換為對應的消息。

29.3 傳送通知

29.3.1 使用 Notifiable Trait

通知可以通過兩種方式傳送： 使用 `Notifiable` 特性的 `notify` 方法或使用 `Notification` facade。 該 `Notifiable` 特性默認包含在應用程式的 `App\Models\User` 模型中：

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;
}
```

此 `notify` 方法需要接收一個通知實例參數：

```
use App\Notifications\InvoicePaid;

$user->notify(new InvoicePaid($invoice));
```

> **技巧** 請記住，你可以在任何模型中使用 `Notifiable` trait。而不僅僅是在 `User` 模型中。

29.3.2 使用 Notification Facade

另外，你可以通過 `Notification` facade 來傳送通知。它主要用在當你需要給多個可接收通知的實體傳送的

時候，比如給使用者集合傳送通知。使用 Facade 傳送通知的話，要把可接收通知實例和通知實例傳遞給 send 方法：

```
use Illuminate\Support\Facades\Notification;

Notification::send($users, new InvoicePaid($invoice));
```

你也可以使用 sendNow 方法立即傳送通知。即使通知實現了 ShouldQueue 介面，該方法也會立即傳送通知：

```
Notification::sendNow($developers, new DeploymentCompleted($deployment));
```

29.3.3 傳送指定頻道

每個通知類都有一個 via 方法，用於確定將在哪些通道上傳遞通知。通知可以在 mail、database、broadcast、vonage 和 slack 頻道上傳送。

> **提示** 如果你想使用其他的頻道，比如 Telegram 或者 Pusher，你可以去看下社區驅動的 Laravel 通知頻道網站.

via 方法接收一個 $notifiable 實例，這個實例將是通知實際傳送到的類的實例。你可以用 $notifiable 來決定這個通知用哪些頻道來傳送：

```
/**
 * 獲取通知傳送頻道。
 *
 * @return array<int, string>
 */
public function via(object $notifiable): array
{
    return $notifiable->prefers_sms ? ['vonage'] : ['mail', 'database'];
}
```

29.3.4 通知佇列化

> **注意** 使用通知佇列前需要組態佇列並 開啟一個佇列任務。

傳送通知可能是耗時的，尤其是通道需要呼叫額外的 API 來傳輸通知。為了加速應用的響應時間，可以將通知推送到佇列中非同步傳送，而要實現推送通知到佇列，可以讓對應通知類實現 ShouldQueue 介面並使用 Queueable trait。如果通知類是通過 make:notification 命令生成的，那麼該介面和 trait 已經默認匯入，你可以快速將它們新增到通知類：

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Notification;

class InvoicePaid extends Notification implements ShouldQueue
{
    use Queueable;

    // ...
}
```

一旦將 ShouldQueue 介面新增到你的通知中，你就可以傳送通知。 Laravel 將檢測類上的 ShouldQueue 介面並自動排隊傳送通知：

```
$user->notify(new InvoicePaid($invoice));
```

排隊通知時，將為每個收件人和頻道組合建立一個排隊的作業。比如，如果你的通知有三個收件人和兩個頻道，則六個作業將被分配到佇列中。

29.3.4.1 延遲通知

如果你需要延遲傳送消息通知, 你可以在你的消息通知實例上新增 `delay` 方法:

```
$delay = now()->addMinutes(10);

$user->notify((new InvoicePaid($invoice))->delay($delay));
```

29.3.4.2 多個通道的延遲通知

將一個陣列傳遞給 `delay` 方法來指定特定通道的延遲時間:

```
$user->notify((new InvoicePaid($invoice))->delay([
     'mail' => now()->addMinutes(5),
     'sms' => now()->addMinutes(10),
]));
```

或者，你可以在通知類本身上定義一個 `withDelay` 方法。 `withDelay` 方法會返回包含通道名稱和延遲值的陣列:

```
/**
 * 確定通知的傳遞延遲.
 *
 * @return array<string, \Illuminate\Support\Carbon>
 */
public function withDelay(object $notifiable): array
{
     return [
          'mail' => now()->addMinutes(5),
          'sms' => now()->addMinutes(10),
     ];
}
```

29.3.4.3 自訂消息通知佇列連接

默認情況下，排隊的消息通知將使用應用程式的默認佇列連接進行排隊。如果你想指定一個不同的連接用於特定的通知，你可以在通知類上定義一個 `$connection` 屬性:

```
/**
 * 排隊通知時要使用的佇列連接的名稱.
 *
 * @var string
 */
public $connection = 'redis';
```

或者，如果你想為每個通知通道都指定一個特定的佇列連接，你可以在你的通知上定義一個 `viaConnections` 方法。這個方法應該返回一個通道名稱 / 佇列連接名稱的陣列。

```
/**
 * 定義每個通知通道應該使用哪個連接。
 *
 * @return array<string, string>
 */
public function viaConnections(): array
{
     return [
          'mail' => 'redis',
          'database' => 'sync',
     ];
}
```

29.3.4.4 自訂通知通道佇列

如果你想為每個通知通道指定一個特定的佇列，你可以在你的通知上定義一個 `viaQueues` 。 此方法應返回通道名稱 / 佇列名稱對的陣列：

```
/**
 * 定義每個通知通道應使用哪條佇列。
 *
 * @return array<string, string>
 */
public function viaQueues(): array
{
    return [
        'mail' => 'mail-queue',
        'slack' => 'slack-queue',
    ];
}
```

29.3.4.5 佇列通知 & 資料庫事務

當佇列通知在資料庫事務中被分發時，它們可能在資料庫事務提交之前被佇列處理。發生這種情況時，你在資料庫事務期間對模型或資料庫記錄所做的任何更新可能尚未反映在資料庫中。甚至，在事務中建立的任何模型或資料庫記錄可能不存在於資料庫中。如果你的通知依賴於這些模型，那麼在處理髮送佇列通知時可能會發生意外錯誤。

如果你的佇列連接的 `after_commit` 組態選項設定為 `false`，你仍然可以通過在傳送通知時呼叫 `afterCommit` 方法來指示應在提交所有打開的資料庫事務後傳送特定的排隊通知：

```
use App\Notifications\InvoicePaid;

$user->notify((new InvoicePaid($invoice))->afterCommit());
```

或者，你可以從通知的建構函式呼叫 `afterCommit` 方法：

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Notification;

class InvoicePaid extends Notification implements ShouldQueue
{
    use Queueable;

    /**
     * 建立一個新的通知通知實例。
     */
    public function __construct()
    {
        $this->afterCommit();
    }
}
```

> **注意**
> 要瞭解更多解決這些問題的方法，請查閱有關佇列作業和 資料庫事務 的文件。

29.3.4.6 確定是否傳送排隊的通知

在將排隊的通知分派到後台處理的佇列之後，它通常會被佇列工作處理程序接受並行送給其目標收件人。

然而，如果你想要在佇列工作處理程序處理後最終確定是否傳送排隊的通知，你可以在通知類上定義一個

shouldSend 方法。如果此方法返回 false，則通知不會被傳送：

```
/**
 * 定義通知是否應該被傳送。
 */
public function shouldSend(object $notifiable, string $channel): bool
{
    return $this->invoice->isPaid();
}
```

29.3.5 按需通知

有時你需要向一些不屬於你應用程式的「使用者」傳送通知。使用 Notification 門面的 route 方法，你可以在傳送通知之前指定即時的通知路由資訊：

```
use Illuminate\Broadcasting\Channel;
use Illuminate\Support\Facades\Notification;

Notification::route('mail', 'taylor@example.com')
            ->route('vonage', '5555555555')
            ->route('slack', 'https://hooks.slack.com/services/...')
            ->route('broadcast', [new Channel('channel-name')])
            ->notify(new InvoicePaid($invoice));
```

如果你想在向 mail 路由傳送通知時指定收件人，你可以提供一個陣列 電子郵件地址作為鍵，名字作為值。

```
Notification::route('mail', [
    'barrett@example.com' => 'Barrett Blair',
])->notify(new InvoicePaid($invoice));
```

29.4 郵件通知

29.4.1 格式化郵件

如果一個通知支援以電子郵件的形式傳送，你應該在通知類中定義一個 toMail 方法。這個方法將接收一個 $notifiable 實體，並應該返回一個 Illuminate\Notifications\Messages\MailMessage 實例。

MailMessage 類包含一些簡單的方法來幫助你建立事務性的電子郵件資訊。郵件資訊可能包含幾行文字以及一個「操作」。讓我們來看看一個 toMail 方法的例子。

```
/**
 * 獲取通知的郵件表示形式。
 */
public function toMail(object $notifiable): MailMessage
{
    $url = url('/invoice/'.$this->invoice->id);

    return (new MailMessage)
                ->greeting('你好!')
                ->line('你的一張發票已經付款了！')
                ->lineIf($this->amount > 0, "支出金額: {$this->amount}")
                ->action('查看發票', $url)
                ->line('感謝你使用我們的應用程式！');
}
```

> **注意**
> 注意我們在 toMail 方法中使用 $this->invoice->id。你可以在通知的建構函式中傳遞任何你的通知需要生成的資訊資料。

在這個例子中，我們註冊了一個問候語、一行文字、一個操作，然後是另一行文字。MailMessage 對象所提

供的這些方法使得郵件的格式化變得簡單而快速。然後，郵件通道將把資訊元件轉換封裝成一個漂亮的、響應式的 HTML 電子郵件範本，並有一個純文字對應。 下面是一個由 `mail` 通道生成的電子郵件的例子。

> **注意**
> 當傳送郵件通知時，請確保在你的 `config/app.php` 組態檔案中設定 `name` 組態選項。 這個值將在你的郵件通知資訊的標題和頁尾中使用。

29.4.1.1 錯誤消息

一些通知會通知使用者錯誤，比如支付失敗的發票。你可以通過在建構消息時呼叫 `error` 方法來指示郵件消息是關於錯誤的。當在郵件消息上使用 `error` 方法時，操作按鈕將會是紅色而不是黑色：

```
/**
 * 獲取通知的郵件表示形式。
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
                ->error()
                ->subject('發票支付失敗')
                ->line('...');
}
```

29.4.1.2 其他郵件通知格式選項

你可以使用 `view` 方法來指定應用於呈現通知電子郵件的自訂範本，而不是在通知類中定義文字「行」：

```
/**
 * 獲取通知的郵件表現形式
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)->view(
        'emails.name', ['invoice' => $this->invoice]
    );
}
```

你可以通過將檢視名稱作為陣列的第二個元素傳遞給 `view` 方法來指定郵件消息的純文字檢視：

```
/**
 * 獲取通知的郵件表現形式
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)->view(
        ['emails.name.html', 'emails.name.plain'],
        ['invoice' => $this->invoice]
    );
}
```

29.4.2 自訂發件人

默認情況下，電子郵件的發件人/寄件人地址在 `config/mail.php` 組態檔案中定義。但是，你可以使用 `from` 方法為特定的通知指定發件人地址：

```
/**
 * 獲取通知的郵件表現形式
 */
public function toMail(object $notifiable): MailMessage
{
```

```
    return (new MailMessage)
                ->from('barrett@example.com', 'Barrett Blair')
                ->line('...');
}
```

29.4.3 自訂收件人

當通過 `mail` 通道傳送通知時，通知系統將自動尋找可通知實體的 `email` 屬性。你可以通過在可通知實體上定義 `routeNotificationForMail` 方法來自訂用於傳遞通知的電子郵件地址：

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Illuminate\Notifications\Notification;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * 路由郵件通道的通知。
     *
     * @return  array<string, string>|string
     */
    public function routeNotificationForMail(Notification $notification): array|string
    {
        // 只返回電子郵件地址...
        return $this->email_address;

        // 返回電子郵件地址和姓名...
        return [$this->email_address => $this->name];
    }
}
```

29.4.4 自訂主題

默認情況下，郵件的主題是通知類的類名格式化為「標題案例」（Title Case）。因此，如果你的通知類命名為 `InvoicePaid`，則郵件的主題將是 `Invoice Paid`。如果你想為消息指定不同的主題，可以在建構消息時呼叫 `subject` 方法：

```
/**
 * 獲取通知的郵件表現形式。
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
                ->subject('通知標題')
                ->line('...');
}
```

29.4.5 自訂郵件程序

默認情況下，郵件通知將使用 `config/mail.php` 組態檔案中定義的默認郵件程序進行傳送。但是，你可以在執行階段通過在建構消息時呼叫 `mailer` 方法來指定不同的郵件程序：

```
/**
 * 獲取通知的郵件表現形式。
 */
```

```
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
                ->mailer('postmark')
                ->line('...');
}
```

29.4.6 自訂範本

你可以通過發佈通知包的資源來修改郵件通知使用的 HTML 和純文字範本。運行此命令後，郵件通知範本將位於 `resources/views/vendor/notifications` 目錄中：

```
php artisan vendor:publish --tag=laravel-notifications
```

29.4.7 附件

要在電子郵件通知中新增附件，可以在建構消息時使用 `attach` 方法。`attach` 方法接受檔案的絕對路徑作為其第一個參數：

```
/**
 * 獲取通知的郵件表現形式。
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
                ->greeting('你好!')
                ->attach('/path/to/file');
}
```

> **注意**
> 通知郵件消息提供的 `attach` 方法還接受 可附加對象。請查閱全面的 可附加對象 文件以瞭解更多資訊。

當附加檔案到消息時，你還可以通過將 `array` 作為 `attach` 方法的第二個參數來指定顯示名稱和/或 MIME 類型：

```
/**
 * 獲取通知的郵件表現形式。
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
                ->greeting('你好!')
                ->attach('/path/to/file', [
                    'as' => 'name.pdf',
                    'mime' => 'application/pdf',
                ]);
}
```

與在可郵寄對象中附加檔案不同，你不能使用 `attachFromStorage` 直接從儲存磁碟附加檔案。相反，你應該使用 `attach` 方法，並提供儲存磁碟上檔案的絕對路徑。或者，你可以從 `toMail` 方法中返回一個 可郵寄對象：

```
use App\Mail\InvoicePaid as InvoicePaidMailable;

/**
 * 獲取通知的郵件表現形式。
 */
public function toMail(object $notifiable): Mailable
{
    return (new InvoicePaidMailable($this->invoice))
                ->to($notifiable->email)
```

```
                ->attachFromStorage('/path/to/file');
}
```

必要時，可以使用 attachMany 方法將多個檔案附加到消息中：

```
/**
 * 獲取通知的郵件表現形式。
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
                ->greeting('你好!')
                ->attachMany([
                    '/path/to/forge.svg',
                    '/path/to/vapor.svg' => [
                        'as' => 'Logo.svg',
                        'mime' => 'image/svg+xml',
                    ],
                ]);
}
```

29.4.7.1 原始資料附件

attachData 方法可以用於將原始位元組陣列附加為附件。在呼叫 attachData 方法時，應提供應分配給附件的檔案名稱：

```
/**
 * 獲取通知的郵件表現形式。
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
                ->greeting('你好!')
                ->attachData($this->pdf, 'name.pdf', [
                    'mime' => 'application/pdf',
                ]);
}
```

29.4.8 新增標籤和中繼資料

一些第三方電子郵件提供商（如 Mailgun 和 Postmark）支援消息「標籤」和「中繼資料」，可用於分組和跟蹤應用程式傳送的電子郵件。可以通過 tag 和 metadata 方法將標籤和中繼資料新增到電子郵件消息中：

```
/**
 * 獲取通知的郵件表現形式。
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
                ->greeting('評論點贊！')
                ->tag('點贊')
                ->metadata('comment_id', $this->comment->id);
}
```

如果你的應用程式使用 Mailgun 驅動程式，則可以查閱 Mailgun 的文件以獲取有關 標籤 和 中繼資料 的更多資訊。同樣，也可以參考 Postmark 文件瞭解他們對 標籤 和 中繼資料 的支援。

如果你的應用程式使用 Amazon SES 傳送電子郵件，則應使用 metadata 方法將 SES 「標籤」附加到消息。

29.4.9 自訂 Symfony 消息

MailMessage 類的 withSymfonyMessage 方法允許你註冊一個閉包，在傳送消息之前將呼叫 Symfony

Message 實例。這給你在傳遞消息之前有深度自訂消息的機會：

```
use Symfony\Component\Mime\Email;

/**
 * 獲取通知的郵件表示形式。
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
                ->withSymfonyMessage(function (Email $message) {
                    $message->getHeaders()->addTextHeader(
                        'Custom-Header', 'Header Value'
                    );
                });
}
```

29.4.10 使用可郵寄對象

如果需要，你可以從通知的 `toMail` 方法返回完整的 mailable 對象。當返回 `Mailable` 而不是 `MailMessage` 時，你需要使用可郵寄對象的 `to` 方法指定消息接收者：

```
use App\Mail\InvoicePaid as InvoicePaidMailable;
use Illuminate\Mail\Mailable;

/**
 * 獲取通知的郵件表現形式。
 */
public function toMail(object $notifiable): Mailable
{
    return (new InvoicePaidMailable($this->invoice))
                ->to($notifiable->email);
}
```

29.4.10.1 可郵寄對象和按需通知

如果你正在傳送按需通知，則提供給 `toMail` 方法的`$notifiable` 實例將是 `Illuminate\Notifications\AnonymousNotifiable` 的一個實例，它提供了一個 `routeNotificationFor` 方法，該方法可用於檢索應將按需通知傳送到的電子郵件地址：

```
use App\Mail\InvoicePaid as InvoicePaidMailable;
use Illuminate\Notifications\AnonymousNotifiable;
use Illuminate\Mail\Mailable;

/**
 * 獲取通知的郵件表現形式。
 */
public function toMail(object $notifiable): Mailable
{
    $address = $notifiable instanceof AnonymousNotifiable
             ? $notifiable->routeNotificationFor('mail')
             : $notifiable->email;

    return (new InvoicePaidMailable($this->invoice))
                ->to($address);
}
```

29.4.11 預覽郵件通知

設計郵件通知範本時，可以像典型的 Blade 範本一樣在瀏覽器中快速預覽呈現的郵件消息。出於這個原因，Laravel 允許你直接從路由閉包或 controller 返回由郵件通知生成的任何郵件消息。當返回一個 `MailMessage` 時，它將在瀏覽器中呈現和顯示，讓你可以快速預覽其設計，無需將其傳送到實際的電子郵件

地址：

```
use App\Models\Invoice;
use App\Notifications\InvoicePaid;

Route::get('/notification', function () {
    $invoice = Invoice::find(1);

    return (new InvoicePaid($invoice))
                ->toMail($invoice->user);
});
```

29.5 Markdown 郵件通知

Markdown 郵件通知允許你利用郵件通知的預建構範本，同時為你提供編寫更長、定製化消息的自由。由於這些消息是用 Markdown 寫的，因此 Laravel 能夠為消息呈現漂亮、響應式的 HTML 範本，同時還會自動生成一個純文字的副本。

29.5.1 生成消息

要生成具有相應 Markdown 範本的通知，可以使用 `make:notification` Artisan 命令的 `--markdown` 選項：

```
php artisan make:notification InvoicePaid --markdown=mail.invoice.paid
```

與所有其他郵件通知一樣，使用 Markdown 範本的通知應該在其通知類上定義一個 `toMail` 方法。但是，不要使用 `line` 和 `action` 方法建構通知，而是使用 `markdown` 方法指定應該使用的 Markdown 範本的名稱。你希望提供給範本的資料陣列可以作為該方法的第二個參數傳遞：

```
/**
 * 獲取通知的郵件表現形式。
 */
public function toMail(object $notifiable): MailMessage
{
    $url = url('/invoice/'.$this->invoice->id);

    return (new MailMessage)
                ->subject('發票支付')
                ->markdown('mail.invoice.paid', ['url' => $url]);
}
```

29.5.2 編寫消息

Markdown 郵件通知使用 Blade 元件和 Markdown 語法的組合，可以讓你在利用 Laravel 的預建構通知元件的同時，輕鬆建構通知：

```
<x-mail::message>
# 發票支付

你的發票已支付！

<x-mail::button :url="$url">
查看發票
</x-mail::button>

謝謝，<br>
{{ config('app.name') }}
</x-mail::message>
```

29.5.2.1 Button 元件

Button 元件會呈現一個居中的按鈕連結。該元件接受兩個參數，`url` 和一個可選的 `color`。支援的顏色有 `primary`、`green` 和 `red`。你可以在通知中新增任意數量的 Button 元件：

```
<x-mail::button :url="$url" color="green">
查看發票
</x-mail::button>
```

29.5.2.2 Panel 元件

Panel 元件會在通知中呈現給定的文字塊，並在面板中以稍微不同的背景顏色呈現。這讓你可以引起讀者對特定文字塊的注意：

```
<x-mail::panel>
這是面板內容。
</x-mail::panel>
```

29.5.2.3 Table 元件

Table 元件允許你將 Markdown 表格轉換為 HTML 表格。該元件接受 Markdown 表格作為其內容。可以使用默認的 Markdown 表格對齊語法來支援表格列對齊：

```
<x-mail::table>
| Laravel       | Table         | Example  |
| ------------- |:-------------:| --------:|
| Col 2 is      | Centered      | $10      |
| Col 3 is      | Right-Aligned | $20      |
</x-mail::table>
```

29.5.3 定製元件

你可以將所有的 Markdown 通知元件匯出到自己的應用程式進行定製。要匯出元件，請使用 `vendor:publish` Artisan 命令來發佈 `laravel-mail` 資源標記：

這個命令會將 Markdown 郵件元件發佈到 `resources/views/vendor/mail` 目錄下。`mail` 目錄將包含一個 `html` 和一個 `text` 目錄，每個目錄都包含其可用元件的各自表示形式。你可以自由地按照自己的喜好定製這些元件。

29.5.3.1 定製 CSS 樣式

在匯出元件之後，`resources/views/vendor/mail/html/themes` 目錄將包含一個 `default.css` 檔案。你可以在此檔案中自訂 CSS 樣式，你的樣式將自動被內聯到 Markdown 通知的 HTML 表示中。

如果你想為 Laravel 的 Markdown 元件建構一個全新的主題，可以在 `html/themes` 目錄中放置一個 CSS 檔案。命名並保存 CSS 檔案後，更新 `mail` 組態檔案的 `theme` 選項以匹配你的新主題的名稱。

要為單個通知自訂主題，可以在建構通知的郵件消息時呼叫 `theme` 方法。`theme` 方法接受應該在傳送通知時使用的主題名稱：

```
/**
 * 獲取通知的郵件表現形式。
 */
public function toMail(object $notifiable): MailMessage
{
    return (new MailMessage)
                ->theme('發票')
                ->subject('發票支付')
                ->markdown('mail.invoice.paid', ['url' => $url]);
```

```
}
```

29.6 資料庫通知

29.6.1 前提條件

database 通知管道將通知資訊儲存在一個資料庫表中。該表將包含通知類型以及描述通知的 JSON 資料結構等資訊。

你可以查詢該表，在你的應用程式使用者介面中顯示通知。但是，在此之前，你需要建立一個資料庫表來保存你的通知。你可以使用 notifications:table 命令生成一個適當的表模式的 遷移：

```
php artisan notifications:table

php artisan migrate
```

29.6.2 格式化資料庫通知

如果一個通知支援被儲存在一個資料庫表中，你應該在通知類上定義一個 toDatabase 或 toArray 方法。這個方法將接收一個 $notifiable 實體，並應該返回一個普通的 PHP 陣列。返回的陣列將被編碼為 JSON，並儲存在你的 notifications 表的 data 列中。讓我們看一個 toArray 方法的例子：

```
/**
 * 獲取通知的陣列表示形式。
 *
 * @return array<string, mixed>
 */
public function toArray(object $notifiable): array
{
    return [
        'invoice_id' => $this->invoice->id,
        'amount' => $this->invoice->amount,
    ];
}
```

29.6.2.1 toDatabase Vs. toArray

toArray 方法也被 broadcast 頻道用來確定要廣播到你的 JavaScript 前端的資料。如果你想為 database 和 broadcast 頻道定義兩個不同的陣列表示形式，你應該定義一個 toDatabase 方法，而不是一個 toArray 方法。

29.6.3 訪問通知

一旦通知被儲存在資料庫中，你需要一個方便的方式從你的可通知實體中訪問它們。Illuminate\Notifications\Notifiable trait 包含在 Laravel 的默認 App\Models\User 模型中，它包括一個 notifications Eloquent 關聯，返回實體的通知。要獲取通知，你可以像訪問任何其他 Eloquent 關係一樣訪問此方法。默認情況下，通知將按照 created_at 時間戳排序，最新的通知位於集合的開頭：

```
$user = App\Models\User::find(1);

foreach ($user->notifications as $notification) {
    echo $notification->type;
}
```

如果你想要只檢索「未讀」的通知，你可以使用 unreadNotifications 關係。同樣，這些通知將按照 created_at 時間戳排序，最新的通知位於集合的開頭：

```
$user = App\Models\User::find(1);

foreach ($user->unreadNotifications as $notification) {
    echo $notification->type;
}
```

注意
要從你的 JavaScript 客戶端訪問你的通知，你應該為你的應用程式定義一個通知 controller ，該 controller 返回一個可通知實體的通知，如當前使用者。然後，你可以從你的 JavaScript 客戶端向該 controller 的 URL 傳送 HTTP 請求。

29.6.4 將通知標記為已讀

通常，當使用者查看通知時，你希望將通知標記為「已讀」。`Illuminate\Notifications\Notifiable` trait 提供了一個 `markAsRead` 方法，該方法將更新通知的資料庫記錄上的 `read_at` 列：

```
$user = App\Models\User::find(1);

foreach ($user->unreadNotifications as $notification) {
    $notification->markAsRead();
}
```

然而，你可以直接在通知集合上使用 `markAsRead` 方法，而不是遍歷每個通知：

```
$user->unreadNotifications->markAsRead();
```

你也可以使用批次更新查詢將所有通知標記為已讀而不必從資料庫中檢索它們：

```
$user = App\Models\User::find(1);

$user->unreadNotifications()->update(['read_at' => now()]);
```

你可以使用 `delete` 方法將通知刪除並從表中完全移除：

```
$user->notifications()->delete();
```

29.7 廣播通知

29.7.1 前提條件

在廣播通知之前，你應該組態並熟悉 Laravel 的 事件廣播 服務。事件廣播提供了一種從你的 JavaScript 前端響應伺服器端 Laravel 事件的方法。

29.7.2 格式化廣播通知

`broadcast` 頻道使用 Laravel 的 事件廣播 服務來廣播通知，允許你的 JavaScript 前端即時捕獲通知。如果通知支援廣播，你可以在通知類上定義一個 `toBroadcast` 方法。該方法將接收一個 `$notifiable` 實體，並應該返回一個 `BroadcastMessage` 實例。如果 `toBroadcast` 方法不存在，則將使用 `toArray` 方法來收集應該廣播的資料。返回的資料將被編碼為 JSON 並廣播到你的 JavaScript 前端。讓我們看一個 `toBroadcast` 方法的示例：

```
use Illuminate\Notifications\Messages\BroadcastMessage;

/**
 * 獲取通知的可廣播表示形式。
 */
public function toBroadcast(object $notifiable): BroadcastMessage
{
```

```
    return new BroadcastMessage([
        'invoice_id' => $this->invoice->id,
        'amount' => $this->invoice->amount,
    ]);
}
```

29.7.2.1 廣播佇列組態

所有廣播通知都會被排隊等待廣播。如果你想組態用於排隊廣播操作的佇列連接或佇列名稱，你可以使用 `BroadcastMessage` 的 `onConnection` 和 `onQueue` 方法：

```
return (new BroadcastMessage($data))
                   ->onConnection('sqs')
                   ->onQueue('broadcasts');
```

29.7.2.2 自訂通知類型

除了你指定的資料之外，所有廣播通知還包含一個 `type` 欄位，其中包含通知的完整類名。如果你想要自訂通知的 `type`，可以在通知類上定義一個 `broadcastType` 方法：

```
/**
 * 獲取正在廣播的通知類型。
 */
public function broadcastType(): string
{
    return 'broadcast.message';
}
```

29.7.3 監聽通知

通知會以 `{notifiable}.{id}` 的格式在一個私有頻道上廣播。因此，如果你向一個 ID 為 `1` 的 `App\Models\User` 實例傳送通知，通知將在 `App.Models.User.1` 私有頻道上廣播。當使用 Laravel Echo 時，你可以使用 `notification` 方法輕鬆地在頻道上監聽通知：

```
Echo.private('App.Models.User.' + userId)
    .notification((notification) => {
        console.log(notification.type);
    });
```

29.7.3.1 自訂通知頻道

如果你想自訂實體的廣播通知在哪個頻道上廣播，可以在可通知實體上定義一個 `receivesBroadcastNotificationsOn` 方法：

```
<?php

namespace App\Models;

use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * 使用者接收通知廣播的頻道。
     */
    public function receivesBroadcastNotificationsOn(): string
    {
        return 'users.'.$this->id;
```

```
    }
}
```

29.8 簡訊通知

29.8.1 先決條件

Laravel 中傳送簡訊通知是由 Vonage（之前稱為 Nexmo）驅動的。在通過 Vonage 傳送通知之前，你需要安裝 `laravel/vonage-notification-channel` 和 `guzzlehttp/guzzle` 包：

```
composer require laravel/vonage-notification-channel guzzlehttp/guzzle
```

該包包括一個 組態檔案。但是，你不需要將此組態檔案匯出到自己的應用程式。你可以簡單地使用 `VONAGE_KEY` 和 `VONAGE_SECRET` 環境變數來定義 Vonage 的公共和私有金鑰。

定義好金鑰後，你可以設定一個 `VONAGE_SMS_FROM` 環境變數，該變數定義了你傳送 SMS 消息的默認電話號碼。你可以在 Vonage 控製麵板中生成此電話號碼：

```
VONAGE_SMS_FROM=15556666666
```

29.8.2 格式化簡訊通知

如果通知支援作為 SMS 傳送，你應該在通知類上定義一個 `toVonage` 方法。此方法將接收一個 `$notifiable` 實體並應返回一個 `Illuminate\Notifications\Messages\VonageMessage` 實例：

```
use Illuminate\Notifications\Messages\VonageMessage;

/**
 * 獲取通知的 Vonage / SMS 表示式。
 */
public function toVonage(object $notifiable): VonageMessage
{
    return (new VonageMessage)
                ->content('你的簡訊內容');
}
```

29.8.2.1 Unicode 內容

如果你的 SMS 消息將包含 unicode 字元，你應該在構造 `VonageMessage` 實例時呼叫 `unicode` 方法：

```
use Illuminate\Notifications\Messages\VonageMessage;

/**
 * 獲取通知的 Vonage / SMS 表示式。
 */
public function toVonage(object $notifiable): VonageMessage
{
    return (new VonageMessage)
                ->content('你的統一碼消息')
                ->unicode();
}
```

29.8.3 自訂「來源」號碼

如果你想從一個不同於 `VONAGE_SMS_FROM` 環境變數所指定的電話號碼傳送通知，你可以在 `VonageMessage` 實例上呼叫 `from` 方法：

```
use Illuminate\Notifications\Messages\VonageMessage;
```

```
/**
 * 獲取通知的 Vonage / SMS 表示式。
 */
public function toVonage(object $notifiable): VonageMessage
{
    return (new VonageMessage)
                ->content('您的簡訊內容')
                ->from('15554443333');
}
```

29.8.4 新增客戶關聯

如果你想跟蹤每個使用者、團隊或客戶的消費，你可以在通知中新增「客戶關聯」。Vonage 將允許你使用這個客戶關聯生成報告，以便你能更好地瞭解特定客戶的簡訊使用情況。客戶關聯可以是任何字串，最多 40 個字元。

```
use Illuminate\Notifications\Messages\VonageMessage;

/**
 * 獲取通知的 Vonage / SMS 表示式。
 */
public function toVonage(object $notifiable): VonageMessage
{
    return (new VonageMessage)
                ->clientReference((string) $notifiable->id)
                ->content('你的簡訊內容');
}
```

29.8.5 路由簡訊通知

要將 Vonage 通知路由到正確的電話號碼，請在你的通知實體上定義 `routeNotificationForVonage` 方法：

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Illuminate\Notifications\Notification;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Vonage 通道的路由通知。
     */
    public function routeNotificationForVonage(Notification $notification): string
    {
        return $this->phone_number;
    }
}
```

29.9 Slack 通知

29.9.1 先決條件

在你可以通過 Slack 傳送通知之前，你必須通過 Composer 安裝 Slack 通知通道：

```
composer require laravel/slack-notification-channel
```

你還需要為你的團隊建立一個 Slack 應用。建立應用後，你應該為工作區組態一個「傳入 Webhook」。然後，Slack 將為你提供一個 webhook URL，你可以在 路由 Slack 通知 時使用該 URL。

29.9.2 格式化 Slack 通知

如果通知支援作為 Slack 消息傳送，你應在通知類上定義 `toSlack` 方法。此方法將接收一個 `$notifiable` 實體並應返回一個 `Illuminate\Notifications\Messages\SlackMessage` 實例。Slack 消息可能包含文字內容以及格式化附加文字或欄位陣列的「附件」。讓我們看一個基本的 `toSlack` 示例：

```
use Illuminate\Notifications\Messages\SlackMessage;

/**
 * 獲取通知的 Slack 表示式。
 */
public function toSlack(object $notifiable): SlackMessage
{
    return (new SlackMessage)
                ->content('你的一張發票已經付款了！');
}
```

29.9.3 Slack 附件

你還可以向 Slack 消息新增「附件」。附件提供比簡單文字消息更豐富的格式選項。在這個例子中，我們將傳送一個關於應用程式中發生的異常的錯誤通知，包括一個連結，以查看有關異常的更多詳細資訊：

```
use Illuminate\Notifications\Messages\SlackAttachment;
use Illuminate\Notifications\Messages\SlackMessage;

/**
 * 獲取通知的 Slack 表示形式。
 */
public function toSlack(object $notifiable): SlackMessage
{
    $url = url('/exceptions/'.$this->exception->id);

    return (new SlackMessage)
                ->error()
                ->content('哎呀！出了問題。')
                ->attachment(function (SlackAttachment $attachment) use ($url) {
                    $attachment->title('例外：檔案未找到', $url)
                               ->content('檔案 [background.jpg] 未找到。');
                });
}
```

附件還允許你指定應呈現給使用者的資料陣列。給定的資料將以表格形式呈現，以便於閱讀：

```
use Illuminate\Notifications\Messages\SlackAttachment;
use Illuminate\Notifications\Messages\SlackMessage;

/**
 * 獲取通知的 Slack 表示形式。
 */
public function toSlack(object $notifiable): SlackMessage
{
    $url = url('/invoices/'.$this->invoice->id);

    return (new SlackMessage)
                ->success()
                ->content('你的一張發票已經付款了！')
                ->attachment(function (SlackAttachment $attachment) use ($url) {
                    $attachment->title('Invoice 1322', $url)
```

```
                                    ->fields([
                                          'Title' => 'Server Expenses',
                                          'Amount' => '$1,234',
                                          'Via' => 'American Express',
                                          'Was Overdue' => ':-1:',
                                     ]);
                    });
}
```

29.9.3.1 Markdown 附件內容

如果你的附件欄位中包含 Markdown，則可以使用 `markdown` 方法指示 Slack 解析和顯示給定的附件欄位為 Markdown 格式的文字。此方法接受的值是：`pretext`、`text` 和/或 `fields`。有關 Slack 附件格式的更多資訊，請查看 Slack API 文件：

```
use Illuminate\Notifications\Messages\SlackAttachment;
use Illuminate\Notifications\Messages\SlackMessage;

/**
 * 獲取通知的 Slack 表示形式。
 */
public function toSlack(object $notifiable): SlackMessage
{
    $url = url('/exceptions/'.$this->exception->id);

    return (new SlackMessage)
                ->error()
                ->content('Whoops! Something went wrong.')
                ->attachment(function (SlackAttachment $attachment) use ($url) {
                    $attachment->title('Exception: File Not Found', $url)
                               ->content('File [background.jpg] was *not found*.')
                               ->markdown(['text']);
                });
}
```

29.9.4 路由 Slack 通知

為了將 Slack 通知路由到正確的 Slack 團隊和頻道，請在你的通知實體上定義一個 `routeNotificationForSlack` 方法。它應該返回要傳送通知的 Webhook URL。Webhook URL 可以通過向你的 Slack 團隊新增「傳入 Webhook」服務來生成：

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Illuminate\Notifications\Notification;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * 路由 Slack 頻道的通知。
     */
    public function routeNotificationForSlack(Notification $notification): string
    {
        return 'https://hooks.slack.com/services/...';
    }
}
```

29.10 本地化通知

Laravel 允許你在除了當前請求語言環境之外的其他語言環境中傳送通知，甚至在通知被佇列化的情況下也能記住此語言環境。

為了實現這一功能，`Illuminate\Notifications\Notification` 類提供了 `locale` 方法來設定所需的語言環境。在通知被評估時，應用程式將切換到此語言環境，然後在評估完成後恢復到以前的語言環境：

```
$user->notify((new InvoicePaid($invoice))->locale('es'));
```

通過 `Notification` 門面，也可以實現多個通知實體的本地化：

```
Notification::locale('es')->send(
    $users, new InvoicePaid($invoice)
);
```

29.10.1 使用者首選語言環境

有時，應用程式會儲存每個使用者的首選區域設定。通過在你的可通知模型上實現 `HasLocalePreference` 合同，你可以指示 Laravel 在傳送通知時使用此儲存的區域設定：

```
use Illuminate\Contracts\Translation\HasLocalePreference;

class User extends Model implements HasLocalePreference
{
    /**
     * 獲取使用者的首選語言環境。
     */
    public function preferredLocale(): string
    {
        return $this->locale;
    }
}
```

翻譯：一旦你實現了這個介面，當傳送通知和郵件到該模型時，Laravel 會自動使用首選語言環境。因此，在使用此介面時不需要呼叫 `locale` 方法：

```
$user->notify(new InvoicePaid($invoice));
```

29.11 測試

你可以使用 `Notification` 門面的 `fake` 方法來阻止通知被傳送。通常情況下，傳送通知與你實際測試的程式碼無關。很可能，只需要斷言 Laravel 被指示傳送了給定的通知即可。

在呼叫 `Notification` 門面的 `fake` 方法後，你可以斷言已經被告知將通知傳送給使用者，甚至檢查通知接收到的資料：

```
<?php

namespace Tests\Feature;

use App\Notifications\OrderShipped;
use Illuminate\Support\Facades\Notification;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_orders_can_be_shipped(): void
    {
        Notification::fake();
```

```
        // 執行訂單發貨...

        // 斷言沒有傳送通知...
        Notification::assertNothingSent();

        // 斷言通知已傳送給給定使用者...
        Notification::assertSentTo(
            [$user], OrderShipped::class
        );

        // 斷言未傳送通知...
        Notification::assertNotSentTo(
            [$user], AnotherNotification::class
        );

        // 斷言已傳送給定數量的通知...
        Notification::assertCount(3);
    }
}
```

你可以通過向 `assertSentTo` 或 `assertNotSentTo` 方法傳遞一個閉包來斷言傳送了符合給定「真實性測試」的通知。如果傳送了至少一個符合給定真實性測試的通知，則斷言將成功：

```
Notification::assertSentTo(
    $user,
    function (OrderShipped $notification, array $channels) use ($order) {
        return $notification->order->id === $order->id;
    }
);
```

29.11.1.1 按需通知

如果你正在測試的程式碼傳送 即時通知，你可以使用 `assertSentOnDemand` 方法測試是否傳送了即時通知：

```
Notification::assertSentOnDemand(OrderShipped::class);
```

通過將閉包作為 `assertSentOnDemand` 方法的第二個參數傳遞，你可以確定是否將即時通知傳送到了正確的「route」地址：

```
Notification::assertSentOnDemand(
    OrderShipped::class,
    function (OrderShipped $notification, array $channels, object $notifiable) use
($user) {
        return $notifiable->routes['mail'] === $user->email;
    }
);
```

29.12 通知事件

29.12.1.1 通知傳送事件

傳送通知時，通知系統會調度 `Illuminate\Notifications\Events\NotificationSending` 事件。這包含「可通知」實體和通知實例本身。 你可以在應用程式的 `EventServiceProvider` 中為該事件註冊監聽器：

```
use App\Listeners\CheckNotificationStatus;
use Illuminate\Notifications\Events\NotificationSending;

/**
 * 應用程式的事件偵聽器對應。
 *
```

```
 * @var array
 */
protected $listen = [
    NotificationSending::class => [
        CheckNotificationStatus::class,
    ],
];
```

如果 `NotificationSending` 事件的監聽器從它的 `handle` 方法返回 `false`，通知將不會被傳送：

```
use Illuminate\Notifications\Events\NotificationSending;

/**
 * 處理事件。
 */
public function handle(NotificationSending $event): void
{
    return false;
}
```

在事件監聽器中，你可以訪問事件的 `notifiable`、`notification` 和 `channel` 屬性，以瞭解有關通知接收者或通知本身的更多資訊。

```
/**
 * 處理事件。
 */
public function handle(NotificationSending $event): void
{
    // $event->channel
    // $event->notifiable
    // $event->notification
}
```

29.12.1.2 通知傳送事件

當通知被傳送時，通知系統會觸發 `Illuminate\Notifications\Events\NotificationSent` 事件，其中包含 「notifiable」 實體和通知實例本身。你可以在 `EventServiceProvider` 中註冊此事件的監聽器：

```
use App\Listeners\LogNotification;
use Illuminate\Notifications\Events\NotificationSent;

/**
 * 應用程式的事件偵聽器對應。
 *
 * @var array
 */
protected $listen = [
    NotificationSent::class => [
        LogNotification::class,
    ],
];
```

> **注意**
> 在 `EventServiceProvider` 中註冊了監聽器之後，可以使用 event:generate Artisan 命令快速生成監聽器類。

在事件監聽器中，你可以訪問事件上的 `notifiable`、`notification`、`channel` 和 `response` 屬性，以瞭解更多有關通知收件人或通知本身的資訊：

```
/**
 * 處理事件。
 */
public function handle(NotificationSent $event): void
{
    // $event->channel
```

```
    // $event->notifiable
    // $event->notification
    // $event->response
}
```

29.13 自訂頻道

Laravel 提供了一些通知頻道，但你可能想編寫自己的驅動程式，以通過其他頻道傳遞通知。Laravel 讓這變得簡單。要開始，定義一個包含 `send` 方法的類。該方法應接收兩個參數：`$notifiable` 和 `$notification`。

在 `send` 方法中，你可以呼叫通知上的方法來檢索一個由你的頻道理解的消息對象，然後按照你希望的方式將通知傳送給 `$notifiable` 實例：

```
<?php

namespace App\Notifications;

use Illuminate\Notifications\Notification;

class VoiceChannel
{
    /**
     * 傳送給定的通知
     */
    public function send(object $notifiable, Notification $notification): void
    {
        $message = $notification->toVoice($notifiable);

        // 將通知傳送給 $notifiable 實例...
    }
}
```

一旦你定義了你的通知頻道類，你可以從你的任何通知的 `via` 方法返回該類的名稱。在這個例子中，你的通知的 `toVoice` 方法可以返回你選擇來表示語音消息的任何對象。例如，你可以定義自己的 `VoiceMessage` 類來表示這些消息：

```
<?php

namespace App\Notifications;

use App\Notifications\Messages\VoiceMessage;
use App\Notifications\VoiceChannel;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Notification;

class InvoicePaid extends Notification
{
    use Queueable;

    /**
     * 獲取通知頻道
     */
    public function via(object $notifiable): string
    {
        return VoiceChannel::class;
    }

    /**
     * 獲取通知的語音表示形式
     */
    public function toVoice(object $notifiable): VoiceMessage
    {
        // ...
```

```
    }
}
```

30 package 開發

30.1 介紹

package 是向 Laravel 新增功能的主要方式。 package 可能是處理日期的好方法，例如 Carbon，也可能是允許您將檔案與 Eloquent 模型相關聯的 package ，例如 Spatie 的 Laravel 媒體櫃。

package 有不同類型。有些 package 是獨立的，這意味著它們可以與任何 PHP 框架一起使用。 Carbon 和 PHPUnit 是獨立 package 的示例。這種 package 可以通過 `composer.json` 檔案引入，在 Laravel 中使用。

此外，還有一些 package 是專門用在 Laravel 中。這些 package 可能 package 含路由、 controller 、檢視和組態，專門用於增強 Laravel 應用。本教學主要涵蓋的就是這些專用於 Laravel 的 package 的開發。

30.1.1 關於 Facades

編寫 Laravel 應用時，通常使用契約（Contracts）還是門面（Facades）並不重要，因為兩者都提供了基本相同的可測試性等級。但是，在編寫 package 時， package 通常是無法使用 Laravel 的所有測試輔助函數。如果您希望能夠像將 package 安裝在典型的 Laravel 應用程式中一樣編寫 package 測試，您可以使用 Orchestral Testbench package 。

30.2 package 發現

在 Laravel 應用程式的 `config/app.php` 組態檔案中，providers 選項定義了 Laravel 應該載入的服務提供者列表。當有人安裝您的軟體 package 時，您通常希望您的服務提供者也 package 含在此列表中。 您可以在 package 的 `composer.json` 檔案的 `extra` 部分中定義提供者，而不是要求使用者手動將您的服務提供者新增到列表中。除了服務提供者外，您還可以列出您想註冊的任何 facades：

```
"extra": {
    "laravel": {
        "providers": [
            "Barryvdh\\Debugbar\\ServiceProvider"
        ],
        "aliases": {
            "Debugbar": "Barryvdh\\Debugbar\\Facade"
        }
    }
},
```

當你的 package 組態了 package 發現後，Laravel 會在安裝該 package 時自動註冊服務提供者及 Facades，這樣就為你的 package 使用者創造一個便利的安裝體驗。

30.2.1 退出 package 發現

如果你是 package 消費者，要停用 package 發現功能，你可以在應用的 `composer.json` 檔案的 `extra` 區域列出 package 名：

```
"extra": {
    "laravel": {
        "dont-discover": [
            "barryvdh/laravel-debugbar"
        ]
```

```
    }
},
```

你可以在應用的 `dont-discover` 指令中使用 `*` 字元，停用所有 package 的 package 發現功能：

```
"extra": {
    "laravel": {
        "dont-discover": [
            "*"
        ]
    }
},
```

30.3 服務提供者

服務提供者是你的 package 和 Laravel 之間的連接點。服務提供者負責將事物繫結到 Laravel 的服務容器並告知 Laravel 到哪裡去載入 package 資源，比如檢視、組態及語言檔案。

服務提供者擴展了 `Illuminate/Support/ServiceProvider` 類， package 含兩個方法： `register` 和 `boot`。基本的 `ServiceProvider` 類位於 `illuminate/support` Composer package 中，你應該把它新增到你自己 package 的依賴項中。要瞭解更多關於服務提供者的結構和目的，請查看 服務提供者.

30.4 資源

30.4.1 組態

通常情況下，你需要將你的 package 的組態檔案發佈到應用程式的 `config` 目錄下。這將允許在使用 package 時覆蓋擴展 package 中的默認組態選項。發佈組態檔案，需要在服務提供者的 `boot` 方法中呼叫 `publishes` 方法:

```
/**
 * 引導 package 服務
 */
public function boot(): void
{
    $this->publishes([
        __DIR__.'/../config/courier.php' => config_path('courier.php'),
    ]);
}
```

使用擴展 package 的時候執行 Laravel 的 `vendor:publish` 命令, 你的檔案將被覆制到指定的發佈位置。 一旦你的組態被發佈, 它的值可以像其他的組態檔案一樣被訪問:

```
$value = config('courier.option');
```

> **Warning**
> 你不應該在你的組態檔案中定義閉 package 。當使用者執行 `config:cache` Artisan 命令時，它們不能被正確序列化。

30.4.1.1 默認的 package 組態

你也可以將你自己的 package 的組態檔案與應用程式的發佈副本合併。這將允許你的使用者在組態檔案的發佈副本中只定義他們真正想要覆蓋的選項。要合併組態檔案的值，請使用你的服務提供者的 `register` 方法中的 `mergeConfigFrom` 方法。

`mergeConfigFrom` 方法的第一個參數為你的 package 的組態檔案的路徑，第二個參數為應用程式的組態檔案

副本的名稱：

```
/**
 * 註冊應用程式服務
 */
public function register(): void
{
    $this->mergeConfigFrom(
        __DIR__.'/../config/courier.php', 'courier'
    );
}
```

> **Warning**
> 這個方法只合併了組態陣列的第一層。如果你的使用者部分地定義了一個多維的組態陣列，缺少的選項將不會被合併。

30.4.2 路由

如果你的軟體 package package 含路由，你可以使用 `loadRoutesFrom` 方法載入它們。這個方法會自動判斷應用程式的路由是否被快取，如果路由已經被快取，則不會載入你的路由檔案：

```
/**
 * 引導 package 服務
 */
public function boot(): void
{
    $this->loadRoutesFrom(__DIR__.'/../routes/web.php');
}
```

30.4.3 遷移

如果你的軟體 package package 含了 資料庫遷移 , 你可以使用 `loadMigrationsFrom` 方法來載入它們。`loadMigrationsFrom` 方法的參數為軟體 package 遷移檔案的路徑。

```
/**
 * 引導 package 服務
 */
public function boot(): void
{
    $this->loadMigrationsFrom(__DIR__.'/../database/migrations');
}
```

一旦你的軟體 package 的遷移被註冊，當 `php artisan migrate` 命令被執行時，它們將自動被運行。你不需要把它們匯出到應用程式的 `database/migrations` 目錄中。

30.4.4 語言檔案

如果你的軟體 package package 含 語言檔案 , 你可以使用 `loadTranslationsFrom` 方法來載入它們。 例如, 如果你的 package 被命名為 `courier` , 你應該在你的服務提供者的 `boot` 方法中加入以下內容:

```
/**
 * 引導 package 服務
 */
public function boot(): void
{
    $this->loadTranslationsFrom(__DIR__.'/../lang', 'courier');
}
```

package 的翻譯行是使用 `package::file.line` 的語法慣例來引用的。因此，你可以這樣從 `messages` 檔案中載入 `courier` package 的 `welcome` 行：

```
echo trans('courier::messages.welcome');
```

30.4.4.1 發佈語言檔案

如果你想把 package 的語言檔案發佈到應用程式的 `lang/vendor` 目錄，可以使用服務提供者的 `publishes` 方法。`publishes` 方法接受一個軟體 package 路徑和它們所需的發佈位置的陣列。例如，要發佈 `courier` package 的語言檔案，你可以做以下工作：

```
/**
 * 引導 package 服務
 */
public function boot(): void
{
    $this->loadTranslationsFrom(__DIR__.'/../lang', 'courier');

    $this->publishes([
        __DIR__.'/../lang' => $this->app->langPath('vendor/courier'),
    ]);
}
```

當你的軟體 package 的使用者執行 Laravel 的 `vendor:publish` Artisan 命令時, 你的軟體 package 的語言檔案會被發佈到指定的發佈位置。

30.4.5 檢視

要在 Laravel 註冊你的 package 的 檢視 , 你需要告訴 Laravel 這些檢視的位置. 你可以使用服務提供者的 `loadViewsFrom` 方法來完成。`loadViewsFrom` 方法接受兩個參數: 檢視範本的路徑和 package 的名稱。 例如，如果你的 package 的名字是 `courier`，你可以在服務提供者的 `boot` 方法中加入以下內容：

```
/**
 * 引導 package 服務
 */
public function boot(): void
{
    $this->loadViewsFrom(__DIR__.'/../resources/views', 'courier');
}
```

package 的檢視是使用 `package::view` 的語法慣例來引用的。因此，一旦你的檢視路徑在服務提供者中註冊，你可以像這樣從 `courier` package 中載入 `dashboard` 檢視。

```
Route::get('/dashboard', function () {
    return view('courier::dashboard');
});
```

30.4.5.1 覆蓋 package 的檢視

當你使用 `loadViewsFrom` 方法時, Laravel 實際上為你的檢視註冊了兩個位置: 應用程式的 `resources/views/vendor` 目錄和你指定的目錄。 所以, 以 `courier` package 為例, Laravel 首先會檢查檢視的自訂版本是否已經被開發者放在 `resources/views/vendor/courier` 目錄中。 然後, 如果檢視沒有被定製, Laravel 會搜尋你在呼叫 `loadViewsFrom` 時指定的 package 的檢視目錄. 這使得 package 的使用者可以很容易地定製/覆蓋你的 package 的檢視。

30.4.5.2 發佈檢視

如果你想讓你的檢視可以發佈到應用程式的 `resources/views/vendor` 目錄下，你可以使用服務提供者的 `publishes` 方法。`publishes` 方法接受一個陣列的 package 檢視路徑和它們所需的發佈位置：

```
/**
 * 引導 package 服務
```

```
 */
public function boot(): void
{
    $this->loadViewsFrom(__DIR__.'/../resources/views', 'courier');

    $this->publishes([
        __DIR__.'/../resources/views' => resource_path('views/vendor/courier'),
    ]);
}
```

當你的 package 的使用者執行 Laravel 的 `vendor:publish` Artisan 命令時, 你的 package 的檢視將被覆制到指定的發佈位置。

30.4.6 檢視元件

如果你正在建立一個用 Blade 元件的 package ，或者將元件放在非傳統的目錄中，你將需要手動註冊你的元件類和它的 HTML 標籤別名，以便 Laravel 知道在哪裡可以找到這個元件。你通常應該在你的 package 的服務提供者的 `boot` 方法中註冊你的元件:

```
use Illuminate\Support\Facades\Blade;
use VendorPackage\View\Components\AlertComponent;

/**
 * 引導你的 package 的服務
 */
public function boot(): void
{
    Blade::component('package-alert', AlertComponent::class);
}
```

當元件註冊成功後，你就可以使用標籤別名對其進行渲染：

```
<x-package-alert/>
```

30.4.6.1 自動載入 package 元件

此外，你可以使用 `compoentNamespace` 方法依照規範自動載入元件類。比如，`Nightshade` package 中可能有 `Calendar` 和 `ColorPicker` 元件，存在於 `Nightshade\Views\Components` 命名空間中：

```
use Illuminate\Support\Facades\Blade;

/**
 * 啟動 package 服務
 */
public function boot(): void
{
    Blade::componentNamespace('Nightshade\\Views\\Components', 'nightshade');
}
```

我們可以使用 `package-name::` 語法，通過 package 提供商的命名空間呼叫 package 元件：

```
<x-nightshade::calendar />
<x-nightshade::color-picker />
```

Blade 會通過元件名自動檢測連結到該元件的類。子目錄也支援使用’點’語法。

30.4.6.2 匿名元件

如果 package 中有匿名元件，則必須將它們放在 package 的檢視目錄(由 `loadViewsFrom` 方法指定)的 `components` 資料夾下。然後，你就可以通過在元件名的前面加上 package 檢視的命名空間來對其進行渲染了：

```
<x-courier::alert />
```

30.4.7 “About” Artisan 命令

Laravel 內建的 `about` Artisan 命令提供了應用環境和組態的摘要資訊。 package 可以通過 `AboutCommand` 類為該命令輸出新增附加資訊。一般而言，這些資訊可以在 package 服務提供者的 `boot` 方法中新增：

```
use Illuminate\Foundation\Console\AboutCommand;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    AboutCommand::add('My Package', fn () => ['Version' => '1.0.0']);
}
```

30.5 命令

要在 Laravel 中註冊你的 package 的 Artisan 命令，你可以使用 `commands` 方法。 此方法需要一個命令類名稱陣列。 註冊命令後，您可以使用 Artisan CLI 執行它們：

```
use Courier\Console\Commands\InstallCommand;
use Courier\Console\Commands\NetworkCommand;

/**
 * Bootstrap any package services.
 */
public function boot(): void
{
    if ($this->app->runningInConsole()) {
        $this->commands([
            InstallCommand::class,
            NetworkCommand::class,
        ]);
    }
}
```

30.6 公共資源

你的 package 可能有諸如 JavaScript 、CSS 和圖片等資源。要發佈這些資源到應用程式的 `public` 目錄，請使用服務提供者的 `publishes` 方法。在下面例子中，我們還將新增一個 `public` 資源組標籤，它可以用來輕鬆發佈相關資源組：

```
/**
 * 引導 package 服務
 */
public function boot(): void
{
    $this->publishes([
        __DIR__.'/../public' => public_path('vendor/courier'),
    ], 'public');
}
```

當你的軟體 package 的使用者執行 `vendor:publish` 命令時，你的資源將被覆制到指定的發佈位置。通常使用者需要在每次更新 package 的時候都要覆蓋資源，你可以使用 `--force` 標誌。

```
php artisan vendor:publish --tag=public --force
```

30.7 發佈檔案組

你可能想單獨發佈軟體 package 的資源和資源組。例如，你可能想讓你的使用者發佈你的 package 的組態檔案，而不被強迫發佈你的 package 的資源。你可以通過在呼叫 package 的服務提供者的 `publishes` 方法時對它們進行 `tagging` 來做到這一點。例如，讓我們使用標籤在軟體 package 服務提供者的 `boot` 方法中為 `courier` 軟體 package 定義兩個發佈組（`courier-config` 和 `courier-migrations`）。

```
/**
 * 引導 package 服務
 */
public function boot(): void
{
    $this->publishes([
        __DIR__.'/../config/package.php' => config_path('package.php')
    ], 'courier-config');

    $this->publishes([
        __DIR__.'/../database/migrations/' => database_path('migrations')
    ], 'courier-migrations');
}
```

現在你的使用者可以在執行 `vendor:publish` 命令時引用他們的標籤來單獨發佈這些組。

```
php artisan vendor:publish --tag=courier-config
```

31 處理程序管理

31.1 介紹

Laravel 通過 Symfony Process 元件 提供了一個小而美的 API，讓你可以方便地從 Laravel 應用程式中呼叫外部處理程序。 Laravel 的處理程序管理功能專注於提供最常見的用例和提升開發人員體驗。

31.2 呼叫過程

在呼叫過程中，你可以使用 **處理程序管理** facade 提供的 `run` 和 `start` 方法。 `run` 方法將呼叫一個處理程序並等待處理程序執行完畢，而 `start` 方法用於非同步處理程序執行。我們將在本文件中探究這兩種方法。首先，讓我們瞭解一下如何呼叫基本的同步處理程序並檢查其結果：

```
use Illuminate\Support\Facades\Process;

$result = Process::run('ls -la');

return $result->output();
```

當然，由 `run` 方法返回的 `Illuminate\Contracts\Process\ProcessResult` 實例提供了多種有用的方法，用於檢查處理程序處理結果：

```
$result = Process::run('ls -la');

$result->successful();
$result->failed();
$result->exitCode();
$result->output();
$result->errorOutput();
```

31.2.1.1 拋出異常

如果你有一個處理程序結果，並且希望在退出程式碼大於零（以此表明失敗）的情況下拋出 `Illuminate\Process\Exceptions\ProcessFailedException` 的一個實例，你可以使用 `throw` 和 `throwIf` 方法。如果處理程序沒有失敗，將返回處理程序結果實例：

```
$result = Process::run('ls -la')->throw();

$result = Process::run('ls -la')->throwIf($condition);
```

31.2.2 處理程序選項

當然，你可能需要在呼叫處理程序之前自訂處理程序的行為。幸運的是，Laravel 允許你調整各種處理程序特性，比如工作目錄、超時和環境變數。

31.2.2.1 工作目錄路徑

你可以使用 `path` 方法指定處理程序的工作目錄。如果不呼叫這個方法，處理程序將繼承當前正在執行的 PHP 指令碼的工作目錄

```
$result = Process::path(__DIR__)->run('ls -la');
```

31.2.2.2 輸入

你可以使用 input 方法通過處理程序的“標準輸入”提供輸入：

```
$result = Process::input('Hello World')->run('cat');
```

31.2.2.3 超時

默認情況下，處理程序在執行超過 60 秒後將拋出 Illuminate\Process\Exceptions\ProcessTimedOutException 實例。但是，你可以通過 timeout 方法自訂此行為：

```
$result = Process::timeout(120)->run('bash import.sh');
```

或者，如果要完全停用處理程序超時，你可以呼叫 forever 方法：

```
$result = Process::forever()->run('bash import.sh');
```

idleTimeout 方法可用於指定處理程序在不返回任何輸出的情況下最多運行的秒數：

```
$result = Process::timeout(60)->idleTimeout(30)->run('bash import.sh');
```

31.2.2.4 環境變數

可以通過 env 方法向處理程序提供環境變數。 呼叫的處理程序還將繼承系統定義的所有環境變數：

```
$result = Process::forever()
            ->env(['IMPORT_PATH' => __DIR__])
            ->run('bash import.sh');
```

如果你希望從呼叫的處理程序中刪除繼承的環境變數，則可以為該環境變數提供值為 false：

```
$result = Process::forever()
            ->env(['LOAD_PATH' => false])
            ->run('bash import.sh');
```

31.2.2.5 TTY 模式

tty 方法可以用於為你的處理程序啟用 TTY 模式。 TTY 模式將處理程序的輸入和輸出連接到你的程序的輸入和輸出，允許你的處理程序作為一個處理程序打開編輯器（如 Vim 或 Nano）：

```
Process::forever()->tty()->run('vim');
```

31.2.3 處理程序輸出

如前所述，處理程序輸出可以使用處理程序結果的 output （標準輸出）和 errorOutput （標準錯誤輸出）方法訪問：

```
use Illuminate\Support\Facades\Process;

$result = Process::run('ls -la');

echo $result->output();
echo $result->errorOutput();
```

但是，通過將閉包作為 run 方法的第二個參數，輸出也可以即時收集。閉包將接收兩個參數：輸出的“類型”（stdout 或 stderr）和輸出字串本身：

```
$result = Process::run('ls -la', function (string $type, string $output) {
    echo $output;
});
```

Laravel 還提供了 seeInOutput 和 seeInErrorOutput 方法，這提供了一種方便的方式來確定處理程序輸出中是否包含給定的字串：

```
if (Process::run('ls -la')->seeInOutput('laravel')) {
```

```
    // ...
}
```

31.2.3.1 停用處理程序輸出

如果你的處理程序寫入了大量你不感興趣的輸出，則可以通過在建構處理程序時呼叫 `quietly` 方法來停用輸出檢索。為此，請執行以下操作：

```
use Illuminate\Support\Facades\Process;

$result = Process::quietly()->run('bash import.sh');
```

31.3 非同步處理程序

`start` 方法可以用來非同步地呼叫處理程序，與之相對的是同步的 `run` 方法。使用 `start` 方法可以讓處理程序在背景執行，而不會阻塞應用的其他任務。一旦處理程序被呼叫，你可以使用 `running` 方法來檢查處理程序是否仍在運行：

```
$process = Process::timeout(120)->start('bash import.sh');

while ($process->running()) {
    // ...
}

$result = $process->wait();
```

你可以使用 `wait` 方法來等待處理程序執行完畢，並檢索處理程序的執行結果實例：

```
$process = Process::timeout(120)->start('bash import.sh');

// ...

$result = $process->wait();
```

31.3.1 處理程序 ID 和訊號

`id` 方法可以用來檢索正在運行處理程序的作業系統分配的處理程序 ID：

```
$process = Process::start('bash import.sh');

return $process->id();
```

你可以使用 `signal` 方法向正在運行的處理程序傳送“訊號”。在 PHP 文件中可以找到預定義的訊號常數列表:

```
$process->signal(SIGUSR2);
```

31.3.2 非同步處理程序輸出

當非同步處理程序在執行階段，你可以使用 `output` 和 `errorOutput` 方法訪問其整個當前輸出；但是，你可以使用 `latestOutput` 和 `latestErrorOutput` 方法訪問自上次檢索輸出以來的處理程序輸出：

```
$process = Process::timeout(120)->start('bash import.sh');

while ($process->running()) {
    echo $process->latestOutput();
    echo $process->latestErrorOutput();

    sleep(1);
}
```

與 `run` 方法一樣，也可以通過在 `start` 方法的第二個參數中傳遞一個閉包來從非同步處理程序中即時收集輸

出。閉包將接收兩個參數：輸出類型（stdout 或 stderr）和輸出字串本身：

```
$process = Process::start('bash import.sh', function (string $type, string $output) {
    echo $output;
});

$result = $process->wait();
```

31.4 平行處理

Laravel 還可以輕鬆地管理一組並行的非同步處理程序，使你能夠輕鬆地同時執行多個任務。要開始，請呼叫 pool 方法，該方法接受一個閉包，該閉包接收 Illuminate 實例。

在此閉包中，你可以定義屬於該池的處理程序。一旦通過 start 方法啟動了處理程序池，你可以通過 running 方法訪問正在運行的處理程序 集合：

```
use Illuminate\Process\Pool;
use Illuminate\Support\Facades\Process;

$pool = Process::pool(function (Pool $pool) {
    $pool->path(__DIR__)->command('bash import-1.sh');
    $pool->path(__DIR__)->command('bash import-2.sh');
    $pool->path(__DIR__)->command('bash import-3.sh');
})->start(function (string $type, string $output, int $key) {
    // ...
});

while ($pool->running()->isNotEmpty()) {
    // ...
}

$results = $pool->wait();
```

可以看到，你可以通過 wait 方法等待所有池處理程序完成執行並解析它們的結果。wait 方法返回一個可訪問處理程序結果實例的陣列對象，通過其鍵可以訪問池中每個處理程序的處理程序結果實例：

```
$results = $pool->wait();

echo $results[0]->output();
```

或者，為方便起見，可以使用 concurrently 方法啟動非同步處理程序池並立即等待其結果。結合 PHP 的陣列解構功能，這可以提供特別表示式的語法：

```
[$first, $second, $third] = Process::concurrently(function (Pool $pool) {
    $pool->path(__DIR__)->command('ls -la');
    $pool->path(app_path())->command('ls -la');
    $pool->path(storage_path())->command('ls -la');
});

echo $first->output();
```

31.4.1 命名處理程序池中的處理程序

通過數字鍵訪問處理程序池結果不太具有表達性，因此 Laravel 允許你通過 as 方法為處理程序池中的每個處理程序分配字串鍵。該鍵也將傳遞給提供給 start 方法的閉包，使你能夠確定輸出屬於哪個處理程序：

```
$pool = Process::pool(function (Pool $pool) {
    $pool->as('first')->command('bash import-1.sh');
    $pool->as('second')->command('bash import-2.sh');
    $pool->as('third')->command('bash import-3.sh');
})->start(function (string $type, string $output, string $key) {
    // ...
});
```

```
$results = $pool->wait();

return $results['first']->output();
```

31.4.2 處理程序池處理程序 ID 和訊號

由於處理程序池的 `running` 方法提供了一個包含池中所有已呼叫處理程序的集合，因此你可以輕鬆地訪問基礎池處理程序 ID：

```
$processIds = $pool->running()->each->id();
```

為了方便起見，你可以在處理程序池上呼叫 `signal` 方法，向池中的每個處理程序傳送訊號：

```
$pool->signal(SIGUSR2);
```

31.5 測試

許多 Laravel 服務都提供功能，以幫助你輕鬆、有表達力地編寫測試，Laravel 的處理程序服務也不例外。`Process` 門面的 `fake` 方法允許你指示 Laravel 在呼叫處理程序時返回存根/偽造結果。

31.5.1 偽造處理程序

在探索 Laravel 的偽造處理程序能力時，讓我們想像一下呼叫處理程序的路由：

```
use Illuminate\Support\Facades\Process;
use Illuminate\Support\Facades\Route;

Route::get('/import', function () {
    Process::run('bash import.sh');

    return 'Import complete!';
});
```

在測試這個路由時，我們可以通過在 `Process` 門面上呼叫無參數的 `fake` 方法，讓 Laravel 返回一個偽造的成功處理程序結果。此外，我們甚至可以斷言某個處理程序“已運行”：

```
<?php

namespace Tests\Feature;

use Illuminate\Process\PendingProcess;
use Illuminate\Contracts\Process\ProcessResult;
use Illuminate\Support\Facades\Process;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_process_is_invoked(): void
    {
        Process::fake();

        $response = $this->get('/');

        // 簡單的流程斷言...
        Process::assertRan('bash import.sh');

        // 或者，檢查流程組態...
        Process::assertRan(function (PendingProcess $process, ProcessResult $result) {
            return $process->command === 'bash import.sh' &&
                   $process->timeout === 60;
        });
    }
```

```
}
```

如前所述，在 Process 門面上呼叫 fake 方法將指示 Laravel 始終返回一個沒有輸出的成功處理程序結果。但是，你可以使用 Process 門面的 result 方法輕鬆指定偽造處理程序的輸出和退出碼：

```
Process::fake([
    '*' => Process::result(
        output: 'Test output',
        errorOutput: 'Test error output',
        exitCode: 1,
    ),
]);
```

31.5.2 偽造指定處理程序

在你測試的過程中，如果要偽造不同的處理程序執行結果，你可以通過傳遞一個陣列給 fake 方法來實現。

陣列的鍵應該表示你想偽造的命令模式及其相關結果。星號 * 字元可用作萬用字元，任何未被偽造的處理程序命令將會被實際執行。你可以使用 Process Facade 的 result 方法為這些命令建構 stub/fake 結果：

```
Process::fake([
    'cat *' => Process::result(
        output: 'Test "cat" output',
    ),
    'ls *' => Process::result(
        output: 'Test "ls" output',
    ),
]);
```

如果不需要自訂偽造處理程序的退出碼或錯誤輸出，你可以更方便地將偽造處理程序結果指定為簡單字串：

```
Process::fake([
    'cat *' => 'Test "cat" output',
    'ls *' => 'Test "ls" output',
]);
```

31.5.3 偽造處理程序序列

如果你測試的程式碼呼叫了多個相同命令的處理程序，你可能希望為每個處理程序呼叫分配不同的偽造處理程序結果。你可以使用 Process Facade 的 sequence 方法來實現這一點：

```
Process::fake([
    'ls *' => Process::sequence()
                ->push(Process::result('First invocation'))
                ->push(Process::result('Second invocation')),
]);
```

31.5.4 偽造非同步處理程序的生命週期

到目前為止，我們主要討論了偽造使用 run 方法同步呼叫的處理程序。但是，如果你正在嘗試測試與通過 start 呼叫的非同步處理程序互動的程式碼，則可能需要更複雜的方法來描述偽造處理程序。

例如，讓我們想像以下使用非同步處理程序互動的路由：

```
use Illuminate\Support\Facades\Log;
use Illuminate\Support\Facades\Route;

Route::get('/import', function () {
    $process = Process::start('bash import.sh');

    while ($process->running()) {
        Log::info($process->latestOutput());
```

```
        Log::info($process->latestErrorOutput());
    }

    return 'Done';
});
```

為了正確地偽造這個處理程序，我們需要能夠描述 `running` 方法應返回 `true` 的次數。此外，我們可能想要指定多行順序返回的輸出。為了實現這一點，我們可以使用 `Process` Facade 的 `describe` 方法：

```
Process::fake([
    'bash import.sh' => Process::describe()
            ->output('First line of standard output')
            ->errorOutput('First line of error output')
            ->output('Second line of standard output')
            ->exitCode(0)
            ->iterations(3),
]);
```

讓我們深入研究上面的例子。使用 `output` 和 `errorOutput` 方法，我們可以指定順序返回的多行輸出。`exitCode` 方法可用於指定偽造處理程序的最終退出碼。最後，`iterations` 方法可用於指定 `running` 方法應返回 `true` 的次數。

31.5.5 可用的斷言

如前所述，Laravel 為你的功能測試提供了幾個處理程序斷言。我們將在下面討論每個斷言。

31.5.5.1 assertRan

斷言已經執行了給定的處理程序：

```
use Illuminate\Support\Facades\Process;

Process::assertRan('ls -la');
```

`assertRan` 方法還接受一個閉包，該閉包將接收一個處理程序實例和一個處理程序結果，使你可以檢查處理程序的組態選項。如果此閉包返回 `true`，則斷言將“通過”：

```
Process::assertRan(fn ($process, $result) =>
    $process->command === 'ls -la' &&
    $process->path === __DIR__ &&
    $process->timeout === 60
);
```

傳遞給 `assertRan` 閉包的 `$process` 是 `Illuminate\Process\PendingProcess` 的實例，而 $result 是 `Illuminate\Contracts\Process\ProcessResult` 的實例。

31.5.5.2 assertDidntRun

斷言給定的處理程序沒有被呼叫：

```
use Illuminate\Support\Facades\Process;

Process::assertDidntRun('ls -la');
```

與 `assertRan` 方法類似，`assertDidntRun` 方法也接受一個閉包，該閉包將接收一個處理程序實例和一個處理程序結果，允許你檢查處理程序的組態選項。如果此閉包返回 `true`，則斷言將“失敗”：

```
Process::assertDidntRun(fn (PendingProcess $process, ProcessResult $result) =>
    $process->command === 'ls -la'
);
```

31.5.5.3 assertRanTimes

斷言給定的處理程序被呼叫了指定的次數：

```
use Illuminate\Support\Facades\Process;

Process::assertRanTimes('ls -la', times: 3);
```

`assertRanTimes` 方法也接受一個閉包，該閉包將接收一個處理程序實例和一個處理程序結果，允許你檢查處理程序的組態選項。如果此閉包返回 `true` 並且處理程序被呼叫了指定的次數，則斷言將“通過”：

```
Process::assertRanTimes(function (PendingProcess $process, ProcessResult $result) {
    return $process->command === 'ls -la';
}, times: 3);
```

31.5.6 防止運行未被偽造的處理程序

如果你想確保在單個測試或完整的測試套件中，所有被呼叫的處理程序都已經被偽造，你可以呼叫 `preventStrayProcesses` 方法。呼叫此方法後，任何沒有相應的偽造結果的處理程序都將引發異常，而不是啟動實際處理程序：

```
use Illuminate\Support\Facades\Process;

Process::preventStrayProcesses();

Process::fake([
    'ls *' => 'Test output...',
]);

// 返回假響應...
Process::run('ls -la');

// 拋出一個異常...
Process::run('bash import.sh');
```

32 佇列

32.1 簡介

在建構 Web 應用程式時，你可能需要執行一些任務，例如解析和儲存上傳的 CSV 檔案，這些任務在典型的 Web 請求期間需要很長時間才能執行。 值得慶幸的是，Laravel 允許你輕鬆建立可以在後台處理的佇列任務。通過將時間密集型任務移至佇列，你的應用程式可以以極快的速度響應 Web 請求，並為你的客戶提供更好的使用者體驗。

Laravel 佇列為各種不同的佇列驅動提供統一的佇列 API，例如 Amazon SQS，Redis，甚至關聯式資料庫。

Laravel 佇列的組態選項儲存在 `config/queue.php` 檔案中。 在這個檔案中，你可以找到框架中包含的每個佇列驅動的連接組態，包括資料庫， Amazon SQS, Redis， 和 Beanstalkd 驅動，以及一個會立即執行作業的同步驅動（用於本地開發）。還包括一個用於丟棄排隊任務的 `null` 佇列驅動。

> **技巧**
> Laravel 提供了 Horizon ，適用於 Redis 驅動佇列。 Horizon 是一個擁有漂亮儀表盤的組態系統。如需瞭解更多資訊請查看完整的 Horizon 文件。

32.1.1 連接 Vs. 驅動

在開始使用 Laravel 佇列之前，理解「連接」和「佇列」之間的區別非常重要。 在 `config/queue.php` 組態檔案中，有一個 `connections` 連接選項。 此選項定義連接某個驅動（如 Amazon SQS、Beanstalk 或 Redis）。然而，任何給定的佇列連接都可能有多個「佇列」，這些「佇列」可能被認為是不同的堆疊或成堆的排隊任務。

請注意， `queue` 組態檔案中的每個連接組態示例都包含一個 `queue` 屬性。

這是將任務傳送到給定連接時將被分配到的默認佇列。換句話說，如果你沒有顯式地定義任務應該被傳送到哪個佇列，那麼該任務將被放置在連接組態的 `queue` 屬性中定義的佇列上：

```
use App\Jobs\ProcessPodcast;

// 這個任務將被推送到默認佇列...

ProcessPodcast::dispatch();

// 這個任務將被推送到「emails」佇列...

ProcessPodcast::dispatch()->onQueue('emails');
```

有些應用程式可能不需要將任務推到多個佇列中，而是傾向於使用一個簡單的佇列。然而，如果希望對任務的處理方式進行優先順序排序或分段時，將任務推送到多個佇列就顯得特別有用，因為 Laravel 佇列工作程序允許你指定哪些佇列應該按優先順序處理。例如，如果你將任務推送到一個 `high` 佇列，你可能會運行一個賦予它們更高處理優先順序的 worker：

```
php artisan queue:work --queue=high,default
```

32.1.2 驅動程式說明和先決條件

32.1.2.1 資料庫

要使用 `database` 佇列驅動程式，你需要一個資料庫表來保存任務。要生成建立此表的遷移，請運行 `queue:table` Artisan 命令。一旦遷移已經建立，你可以使用 `migrate` 命令遷移你的資料庫：

```
php artisan queue:table

php artisan migrate
```

最後，請不要忘記通過修改 `.env` 檔案中的 `QUEUE_CONNECTION` 變數從而將 `database` 作為你的應用佇列驅動程式:

```
QUEUE_CONNECTION=database
```

32.1.2.2 Redis

要使用 `redis` 佇列驅動程式，需要在 `config/database.php` 組態檔案中組態一個 redis 資料庫連接。

Redis 叢集

如果你的 Redis 佇列當中使用了 Redis 叢集，那麼你的佇列名稱就必須包含一個 key hash tag。這是為了確保一個給定佇列的所有 Redis 鍵都被放在同一個雜湊插槽：

```
'redis' => [
     'driver' => 'redis',
     'connection' => 'default',
     'queue' => '{default}',
     'retry_after' => 90,
],
```

阻塞

在使用 Redis 佇列時，你可以使用 block_for 組態選項來指定在遍歷 worker 循環和重新輪詢 Redis 資料庫之前，驅動程式需要等待多長時間才能使任務變得可用。

根據你的佇列負載調整此值要比連續輪詢 Redis 資料庫中的新任務更加有效。例如，你可以將值設定為 5 以指示驅動程式在等待任務變得可用時應該阻塞 5 秒：

```
'redis' => [
     'driver' => 'redis',
     'connection' => 'default',
     'queue' => 'default',
     'retry_after' => 90,
     'block_for' => 5,
],
```

> **注意** 將 block_for 設定為 0 將導致佇列 workers 一直阻塞，直到某一個任務變得可用。這還能防止在下一個任務被處理之前處理諸如 SIGTERM 之類的訊號。

32.1.2.3 其他驅動的先決條件

列出的佇列驅動需要如下的依賴，這些依賴可通過 Composer 包管理器進行安裝：

- Amazon SQS: `aws/aws-sdk-php ~3.0`
- Beanstalkd: `pda/pheanstalk ~4.0`
- Redis: `predis/predis ~1.0` or phpredis PHP extension

32.2 建立任務

32.2.1 生成任務類

默認情況下，應用程式的所有的可排隊任務都被儲存在了 app/Jobs 目錄中。如果 app/Jobs 目錄不存在，當你運行 make:job Artisan 命令時，將會自動建立該目錄：

```
php artisan make:job ProcessPodcast
```

生成的類將會實現 Illuminate 介面， 告訴 Laravel ，該任務應該推入佇列以非同步的方式運行。

> **技巧** 你可以使用 stub publishing 來自訂任務 stub 。

32.2.2 任務類結構

任務類非常簡單，通常只包含一個 `handle` 方法，在佇列處理任務時將會呼叫它。讓我們看一個任務類的示例。在這個例子中，我們假設我們管理一個 podcast 服務，並且需要在上傳的 podcast 檔案發佈之前對其進行處理：

```
<?php

namespace App\Jobs;

use App\Models\Podcast;
use App\Services\AudioProcessor;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;

class ProcessPodcast implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    /**
     * 建立一個新的任務實例
     */
    public function __construct(
        public Podcast $podcast,
    ) {}

    /**
     * 運行任務
     */
    public function handle(AudioProcessor $processor): void
    {
        // 處理上傳的 podcast...
    }
}
```

在本示例中，請注意，我們能夠將一個 Eloquent model 直接傳遞到已排隊任務的建構函式中。由於任務所使用的 `SerializesModels` ，在任務處理時，Eloquent 模型及其載入的關係將被優雅地序列化和反序列化。

如果你的佇列任務在其建構函式中接受一個 Eloquent 模型，那麼只有模型的識別碼才會被序列化到佇列中。當實際處理任務時，佇列系統將自動重新從資料庫中獲取完整的模型實例及其載入的關係。這種用於模型序列化的方式允許將更小的作業有效負載傳送給你的佇列驅動程式。

32.2.2.1 `handle` 方法依賴注入

當任務由佇列處理時，將呼叫 `handle` 方法。注意，我們可以對任務的 `handle` 方法進行類型提示依賴。Laravel 服務容器 會自動注入這些依賴項。

如果你想完全控制容器如何將依賴注入 `handle` 方法，你可以使用容器的 `bindMethod` 方法。 `bindMethod` 方法接受一個可接收任務和容器的回呼。在回呼中，你可以在任何你想用的地方隨意呼叫 `handle` 方法。 通常， 你應該從你的 `App\Providers\AppServiceProvider` 服務提供者 中來呼叫該方法:

```
use App\Jobs\ProcessPodcast;
use App\Services\AudioProcessor;
use Illuminate\Contracts\Foundation\Application;

$this->app->bindMethod([ProcessPodcast::class, 'handle'], function (ProcessPodcast $job,
Application $app) {
    return $job->handle($app->make(AudioProcessor::class));
});
```

> **注意** 二進制資料，例如原始圖像內容，應該在傳遞到佇列任務之前通過 `base64_encode` 函數傳遞。否則，在將任務放入佇列時，可能無法正確地序列化為 JSON。

32.2.2.2 佇列關係

因為載入的關係也會被序列化，所以處理序列化任務的字串有時會變得相當大。為了防止該關係被序列化，可以在設定屬性值時對模型呼叫 `withoutRelations` 方法。此方法將返回沒有載入關係的模型實例：

```
/**
 * 建立新的任務實例
 */
public function __construct(Podcast $podcast)
{
    $this->podcast = $podcast->withoutRelations();
}
```

此外，當反序列化任務並從資料庫中重新檢索模型關係時，它們將被完整檢索。反序列化任務時，將不會應用在任務排隊過程中序列化模型之前應用的任何先前關係約束。因此，如果你希望使用給定關係的子集，則應在排隊任務中重新限制該關係。

32.2.3 唯一任務

> 注意：唯一任務需要支援 locks 的快取驅動程式。 目前，`memcached`、`redis`、`dynamodb`、`database`、`file` 和 `array` 快取驅動支援原子鎖。 此外，獨特的任務約束不適用於批次內的任務。

有時，你可能希望確保在任何時間點佇列中只有一個特定任務的實例。你可以通過在你的工作類上實現 `ShouldBeUnique` 介面來做到這一點。這個介面不需要你在你的類上定義任何額外的方法：

```
<?php

use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUnique;

class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
{
    ...
}
```

以上示例中，`UpdateSearchIndex` 任務是唯一的。因此，如果任務的另一個實例已經在佇列中並且尚未完成處理，則不會分派該任務。

在某些情況下，你可能想要定義一個使任務唯一的特定「鍵」，或者你可能想要指定一個超時時間，超過該時間任務不再保持唯一。為此，你可以在任務類上定義 `uniqueId` 和 `uniqueFor` 屬性或方法：

```
<?php

use App\Product;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUnique;

class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
{
    /**
     * 產品實例
     *
     * @var \App\Product
     */
    public $product;

    /**
     * 任務的唯一鎖將被釋放的秒數
     *
     * @var int
     */
    public $uniqueFor = 3600;

    /**
     * 任務的唯一 ID
     */
    public function uniqueId(): string
    {
        return $this->product->id;
    }
}
```

以上示例中， `UpdateSearchIndex` 任務中的 product ID 是唯一的。因此，在現有任務完成處理之前，任何具有相同 product ID 的任務都將被忽略。此外，如果現有任務在一小時內沒有得到處理，則釋放唯一鎖，並將具有相同唯一鍵的另一個任務分派到該佇列。

> **注意** 如果你的應用程式從多個 web 伺服器或容器分派任務，你應該確保你的所有伺服器都與同一個中央快取伺服器通訊，以便 Laravel 能夠準確確定任務是否唯一。

32.2.3.1 在任務處理開始前保證唯一

默認情況下，在任務完成處理或所有重試嘗試均失敗後，唯一任務將被「解鎖」。但是，在某些情況下，你可能希望任務在處理之前立即解鎖。為此，你的任務類可以實現 `ShouldBeUniqueUntilProcessing` 介面，而不是實現 `ShouldBeUnique` 介面：

```
<?php

use App\Product;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUniqueUntilProcessing;

class UpdateSearchIndex implements ShouldQueue, ShouldBeUniqueUntilProcessing
{
    // ...
}
```

32.2.3.2 唯一任務鎖

在底層實現中，當分發 `ShouldBeUnique` 任務時，Laravel 嘗試使用 `uniqueId` 鍵獲取一個 鎖 。如果未獲取到鎖，則不會分派任務。當任務完成處理或所有重試嘗試失敗時，將釋放此鎖。默認情況下，Laravel 將使用默

認的快取驅動程式來獲取此鎖。但是，如果你希望使用其他驅動程式來獲取鎖，則可以定義一個 uniqueVia 方法，該方法返回一個快取驅動對象：

```
use Illuminate\Contracts\Cache\Repository;
use Illuminate\Support\Facades\Cache;

class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
{
    ...

    /**
     * 獲取唯一任務鎖的快取驅動程式
     */
    public function uniqueVia(): Repository
    {
        return Cache::driver('redis');
    }
}
```

技巧：如果只需要限制任務的並行處理，請改用 WithoutOverlapping 任務中介軟體。

32.3 任務中介軟體

任務中介軟體允許你圍繞排隊任務的執行封裝自訂邏輯，從而減少了任務本身的樣板程式碼。例如，看下面的 handle 方法，它利用了 Laravel 的 Redis 速率限制特性，允許每 5 秒只處理一個任務：

```
use Illuminate\Support\Facades\Redis;

/**
 * 執行任務
 */
public function handle(): void
{
    Redis::throttle('key')->block(0)->allow(1)->every(5)->then(function () {
        info('取得了鎖...');

        // 處理任務...
    }, function () {
        // 無法獲取鎖...

        return $this->release(5);
    });
}
```

雖然這段程式碼是有效的， 但是 handle 方法的結構卻變得雜亂，因為它摻雜了 Redis 速率限制邏輯。此外，其他任務需要使用速率限制的時候，只能將限制邏輯複製一次。

我們可以定義一個處理速率限制的任務中介軟體，而不是在 handle 方法中定義速率限制。Laravel 沒有任務中介軟體的默認位置，所以你可以將任務中介軟體放置在你喜歡的任何位置。在本例中，我們將把中介軟體放在 app/Jobs/Middleware 目錄：

```
<?php

namespace App\Jobs\Middleware;

use Closure;
use Illuminate\Support\Facades\Redis;

class RateLimited
{
    /**
     * 處理佇列任務
     *
```

```
     * @param  \Closure(object): void  $next
     */
    public function handle(object $job, Closure $next): void
    {
        Redis::throttle('key')
                ->block(0)->allow(1)->every(5)
                ->then(function () use (object $job, Closure $next) {
                    // 已獲得鎖...

                    $next($job);
                }, function () use ($job) {
                    // 沒有獲取到鎖...

                    $job->release(5);
                });
    }
}
```

正如你看到的，類似於 路由中介軟體，任務中介軟體接收正在處理佇列任務以及一個回呼來繼續處理佇列任務。

在任務中介軟體被建立以後，他們可能被關聯到通過從任務的 `middleware` 方法返回的任務。這個方法並不存在於 `make:job` Artisan 命令搭建的任務中，所以你需要將它新增到你自己的任務類的定義中：

```
use App\Jobs\Middleware\RateLimited;

/**
 * 獲取一個可以被傳遞通過的中介軟體任務
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [new RateLimited];
}
```

技巧 任務中介軟體也可以分配其他可佇列處理的監聽事件當中，比如郵件，通知等。

32.3.1 訪問限制

儘管我們剛剛演示了如何編寫自己的訪問限制的任務中介軟體，但 Laravel 實際上內建了一個訪問限制中介軟體，你可以利用它來限制任務。與 路由限流器 一樣，任務訪問限制器是使用 `RateLimiter` facade 的 `for` 方法定義的。

例如，你可能希望允許使用者每小時備份一次資料，但不對高級客戶施加此類限制。為此，可以在 `RateLimiter` 的 `boot` 方法中定義 `AppServiceProvider`：

```
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Support\Facades\RateLimiter;

/**
 * 註冊應用程式服務
 */
public function boot(): void
{
    RateLimiter::for('backups', function (object $job) {
        return $job->user->vipCustomer()
                    ? Limit::none()
                    : Limit::perHour(1)->by($job->user->id);
    });
}
```

在上面的例子中，我們定義了一個小時訪問限制；但是，你可以使用 `perMinute` 方法輕鬆定義基於分鐘的訪

問限制。此外，你可以將任何值傳遞給訪問限制的 by 方法，但是，這個值通常用於按客戶來區分不同的訪問限制：

```
return Limit::perMinute(50)->by($job->user->id);
```

定義速率限制後，你可以使用 Illuminate\Queue\Middleware\RateLimited 中介軟體將速率限制器附加到備份任務。 每次任務超過速率限制時，此中介軟體都會根據速率限制持續時間以適當的延遲將任務釋放回佇列。

```
use Illuminate\Queue\Middleware\RateLimited;

/**
 * 獲取任務時，應該通過的中介軟體
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [new RateLimited('backups')];
}
```

將速率受限的任務釋放回佇列仍然會增加任務的 「嘗試」總數。你可能希望相應地調整你的任務類上的 tries 和 maxExceptions 屬性。或者，你可能希望使用 retryUntil 方法 來定義不再嘗試任務之前的時間量。

如果你不想在速率限制時重試任務，你可以使用 dontRelease 方法：

```
/**
 * 獲取任務時，應該通過的中介軟體
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [(new RateLimited('backups'))->dontRelease()];
}
```

> **技巧** 如果你使用 Redis，你可以使用 Illuminate 中介軟體，它針對 Redis 進行了微調，比基本的限速中介軟體更高效。

32.3.2 防止任務重疊

Laravel 包含一個 Illuminate\Queue\Middleware\WithoutOverlapping 中介軟體，允許你根據任意鍵防止任務重疊。當排隊的任務正在修改一次只能由一個任務修改的資源時，這會很有幫助。

例如，假設你有一個更新使用者信用評分的排隊任務，並且你希望防止同一使用者 ID 的信用評分更新任務重疊。為此，你可以從任務的 middleware 方法返回 WithoutOverlapping 中介軟體：

```
use Illuminate\Queue\Middleware\WithoutOverlapping;

/**
 * 獲取任務時，應該通過的中介軟體
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [new WithoutOverlapping($this->user->id)];
}
```

任何重疊的任務都將被釋放回佇列。你還可以指定再次嘗試釋放的任務之前必須經過的秒數：

```
/**
 * 獲取任務時，應該通過的中介軟體
```

```
     *
     * @return array<int, object>
     */
    public function middleware(): array
    {
        return [(new WithoutOverlapping($this->order->id))->releaseAfter(60)];
    }
```

如果你想立即刪除任何重疊的任務，你可以使用 `dontRelease` 方法，這樣它們就不會被重試：

```
    /**
     * 獲取任務時，應該通過的中介軟體。
     *
     * @return array<int, object>
     */
    public function middleware(): array
    {
        return [(new WithoutOverlapping($this->order->id))->dontRelease()];
    }
```

`WithoutOverlapping` 中介軟體由 Laravel 的原子鎖特性提供支援。有時，你的任務可能會以未釋放鎖的方式意外失敗或超時。因此，你可以使用 expireAfter 方法顯式定義鎖定過期時間。例如，下面的示例將指示 Laravel 在任務開始處理三分鐘後釋放 WithoutOverlapping 鎖：

```
    /**
     * 獲取任務時，應該通過的中介軟體。
     *
     * @return array<int, object>
     */
    public function middleware(): array
    {
        return [(new WithoutOverlapping($this->order->id))->expireAfter(180)];
    }
```

> **注意** `WithoutOverlapping` 中介軟體需要支援 locks 的快取驅動程式。目前，`memcached`、`redis`、`dynamodb`、`database`、`file` 和 `array` 快取驅動支援原子鎖。

32.3.2.1 跨任務類別共享鎖

默認情況下，`WithoutOverlapping` 中介軟體只會阻止同一類的重疊任務。 因此，儘管兩個不同的任務類可能使用相同的鎖，但不會阻止它們重疊。 但是，你可以使用 `shared` 方法指示 Laravel 跨任務類應用鎖：

```
use Illuminate\Queue\Middleware\WithoutOverlapping;

class ProviderIsDown
{
    // ...


    public function middleware(): array
    {
        return [
            (new WithoutOverlapping("status:{$this->provider}"))->shared(),
        ];
    }
}

class ProviderIsUp
{
    // ...


    public function middleware(): array
    {
        return [
```

```
            (new WithoutOverlapping("status:{$this->provider}"))->shared(),
        ];
    }
}
```

32.3.3 節流限制異常

Laravel 包含一個 `Illuminate\Queue\Middleware\ThrottlesExceptions` 中介軟體，允許你限制異常。一旦任務拋出給定數量的異常，所有進一步執行該任務的嘗試都會延遲，直到經過指定的時間間隔。該中介軟體對於與不穩定的第三方服務互動的任務特別有用。

例如，讓我們想像一個佇列任務與開始拋出異常的第三方 API 互動。要限制異常，你可以從任務的 `middleware` 方法返回 `ThrottlesExceptions` 中介軟體。通常，此中介軟體應與實現 基於時間的嘗試 的任務配對：

```
use DateTime;
use Illuminate\Queue\Middleware\ThrottlesExceptions;

/**
 * 獲取任務時，應該通過的中介軟體。
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [new ThrottlesExceptions(10, 5)];
}

/**
 * 確定任務應該超時的時間。
 */
public function retryUntil(): DateTime
{
    return now()->addMinutes(5);
}
```

中介軟體接受的第一個建構函式參數是任務在被限制之前可以拋出的異常數，而第二個建構函式參數是在任務被限制後再次嘗試之前應該經過的分鐘數。在上面的程式碼示例中，如果任務在 5 分鐘內拋出 10 個異常，我們將等待 5 分鐘，然後再次嘗試該任務。

當任務拋出異常但尚未達到異常閾值時，通常會立即重試該任務。但是，你可以通過在將中介軟體附加到任務時呼叫 `backoff` 方法來指定此類任務應延遲的分鐘數：

```
use Illuminate\Queue\Middleware\ThrottlesExceptions;

/**
 * 獲取任務時，應該通過的中介軟體。
 *
 * @return array<int, object>
 */
public function middleware(): array
{
    return [(new ThrottlesExceptions(10, 5))->backoff(5)];
}
```

在內部，這個中介軟體使用 Laravel 的快取系統來實現速率限制，並利用任務的類名作為快取 「鍵」。 在將中介軟體附加到任務時，你可以通過呼叫 `by` 方法來覆蓋此鍵。 如果你有多個任務與同一個第三方服務互動並且你希望它們共享一個共同的節流 「桶」，這可能會很有用：

```
use Illuminate\Queue\Middleware\ThrottlesExceptions;

/**
 * 獲取任務時，應該通過的中介軟體。
 *
```

```
 * @return array<int, object>
 */
public function middleware(): array
{
    return [(new ThrottlesExceptions(10, 10))->by('key')];
}
```

技巧

如果你使用 Redis，你可以使用 `Illuminate\Queue\Middleware\ThrottlesExceptionsWithRedis` 中介軟體，它針對 Redis 進行了微調，比基本的異常節流中介軟體更高效。

32.4 調度任務

一旦你寫好了你的任務類，你可以使用任務本身的 `dispatch` 方法來調度它。傳遞給 `dispatch` 方法的參數將被提供給任務的建構函式：

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * 儲存一個新的播客。
     */
    public function store(Request $request): RedirectResponse
    {
        $podcast = Podcast::create(/* ... */);

        // ...

        ProcessPodcast::dispatch($podcast);

        return redirect('/podcasts');
    }
}
```

如果你想有條件地分派任務，你可以使用 `dispatchIf` 和 `dispatchUnless` 方法：

```
ProcessPodcast::dispatchIf($accountActive, $podcast);

ProcessPodcast::dispatchUnless($accountSuspended, $podcast);
```

在新的 Laravel 應用程式中，`sync` 是默認的佇列驅動程式。 該驅動程式會在當前請求的前台同步執行任務，這在本地開發時通常會很方便。 如果你想在後台處理排隊任務，你可以在應用程式的 `config/queue.php` 組態檔案中指定一個不同的佇列驅動程式。

32.4.1 延遲調度

如果你想指定任務不應立即可供佇列工作人員處理，你可以在調度任務時使用 `delay` 方法。例如，讓我們指定一個任務在分派後 10 分鐘內不能用於處理

```
<?php

namespace App\Http\Controllers;
```

```
use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * 儲存一個新的播客
     */
    public function store(Request $request): RedirectResponse
    {
        $podcast = Podcast::create(/* ... */);

        // ...

        ProcessPodcast::dispatch($podcast)
                    ->delay(now()->addMinutes(10));

        return redirect('/podcasts');
    }
}
```

注意

Amazon SQS 佇列服務的最大延遲時間為 15 分鐘。

32.4.1.1 響應傳送到瀏覽器後調度

或者，`dispatchAfterResponse` 方法延遲調度任務，直到 HTTP 響應傳送到使用者的瀏覽器之後。 即使排隊的任務仍在執行，這仍將允許使用者開始使用應用程式。這通常應該只用於需要大約一秒鐘的工作，例如傳送電子郵件。由於它們是在當前 HTTP 請求中處理的，因此以這種方式分派的任務不需要運行佇列工作者來處理它們：

```
use App\Jobs\SendNotification;

SendNotification::dispatchAfterResponse();
```

你也可以 `dispatch` 一個閉包並將 `afterResponse` 方法連結到 `dispatch` 幫助器以在 HTTP 響應傳送到瀏覽器後執行一個閉包

```
use App\Mail\WelcomeMessage;
use Illuminate\Support\Facades\Mail;

dispatch(function () {
    Mail::to('taylor@example.com')->send(new WelcomeMessage);
})->afterResponse();
```

32.4.2 同步調度

如果你想立即（同步）調度任務，你可以使用 `dispatchSync` 方法。使用此方法時，任務不會排隊，會在當前處理程序內立即執行：

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
```

```
class PodcastController extends Controller
{
    /**
     * 儲存一個新的播客。
     */
    public function store(Request $request): RedirectResponse
    {
        $podcast = Podcast::create(/* ... */);

        // 建立播客

        ProcessPodcast::dispatchSync($podcast);

        return redirect('/podcasts');
    }
}
```

32.4.3 任務 & 資料庫事務

雖然在資料庫事務中分派任務非常好，但你應該特別注意確保你的任務實際上能夠成功執行。在事務中調度任務時，任務可能會在父事務提交之前由工作人員處理。發生這種情況時，你在資料庫事務期間對模型或資料庫記錄所做的任何更新可能尚未反映在資料庫中。此外，在事務中建立的任何模型或資料庫記錄可能不存在於資料庫中。

值得慶幸的是，Laravel 提供了幾種解決這個問題的方法。首先，你可以在佇列連接的組態陣列中設定 `after_commit` 連接選項：

```
'redis' => [
    'driver' => 'redis',
    // ...
    'after_commit' => true,
],
```

當 `after_commit` 選項為 true 時，你可以在資料庫事務中分發任務；Laravel 會等到所有打開的資料庫事務都已提交，然後才會開始分發任務。當然，如果當前沒有打開的資料庫事務，任務將被立即調度。

如果事務因事務期間發生異常而回滾，則在該事務期間分發的已分發任務將被丟棄。

> **技巧**
> 將 `after_commit` 組態選項設定為 `true` 還會導致所有排隊的事件監聽器、郵件、通知和廣播事件在所有打開的資料庫事務提交後才被調度。

32.4.3.1 內聯指定提交調度

如果你沒有將 `after_commit` 佇列連接組態選項設定為 `true`，你可能需要在所有打開的資料庫事務提交後才調度特定的任務。為此，你可以將 `afterCommit` 方法放到你的調度操作上：

```
use App\Jobs\ProcessPodcast;

ProcessPodcast::dispatch($podcast)->afterCommit();
```

同樣，如果 `after_commit` 組態選項設定為 `true`，則可以指示應立即調度特定作業，而無需等待任何打開的資料庫事務提交：

```
ProcessPodcast::dispatch($podcast)->beforeCommit();
```

32.4.4 任務鏈

任務鏈允許你指定一組應在主任務成功執行後按順序運行的排隊任務。如果序列中的一個任務失敗，其餘的任務將不會運行。要執行一個排隊的任務鏈，你可以使用 Bus facade 提供的 chain 方法：

```
use App\Jobs\OptimizePodcast;
use App\Jobs\ProcessPodcast;
use App\Jobs\ReleasePodcast;
use Illuminate\Support\Facades\Bus;

Bus::chain([
    new ProcessPodcast,
    new OptimizePodcast,
    new ReleasePodcast,
])->dispatch();
```

除了連結任務類實例之外，你還可以連結閉包：

```
Bus::chain([
    new ProcessPodcast,
    new OptimizePodcast,
    function () {
        Podcast::update(/* ... */);
    },
])->dispatch();
```

> **注意**
> 在任務中使用 $this->delete() 方法刪除任務不會阻止鏈式任務的處理。只有當鏈中的任務失敗時，鏈才會停止執行。

32.4.4.1 鏈式連接 & 佇列

如果要指定連結任務應使用的連接和佇列，可以使用 onConnection 和 onQueue 方法。這些方法指定應使用的佇列連接和佇列名稱，除非為排隊任務顯式分配了不同的連接 / 佇列：

```
Bus::chain([
    new ProcessPodcast,
    new OptimizePodcast,
    new ReleasePodcast,
])->onConnection('redis')->onQueue('podcasts')->dispatch();
```

32.4.4.2 鏈故障

連結任務時，你可以使用 catch 方法指定一個閉包，如果鏈中的任務失敗，則應呼叫該閉包。給定的回呼將接收導致任務失敗的 Throwable 實例：

```
use Illuminate\Support\Facades\Bus;
use Throwable;

Bus::chain([
    new ProcessPodcast,
    new OptimizePodcast,
    new ReleasePodcast,
])->catch(function (Throwable $e) {
    // 鏈中的任務失敗了...
})->dispatch();
```

> **注意**
> 由於鏈式回呼由 Laravel 佇列稍後序列化並執行，因此你不應在鏈式回呼中使用 $this 變數。

32.4.5 自訂佇列 & 連接

32.4.5.1 分派到特定佇列

通過將任務推送到不同的佇列，你可以對排隊的任務進行「分類」，甚至可以優先考慮分配給各個佇列的工作人員數量。請記住，這不會將任務推送到佇列組態檔案定義的不同佇列「連接」，而只會推送到單個連接中的特定佇列。要指定佇列，請在調度任務時使用 onQueue 方法：

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
     /**
      * 儲存一個播客。
      */
     public function store(Request $request): RedirectResponse
     {
          $podcast = Podcast::create(/* ... */);

          // 建立播客...

          ProcessPodcast::dispatch($podcast)->onQueue('processing');

          return redirect('/podcasts');
     }
}
```

或者，你可以通過在任務的建構函式中呼叫 onQueue 方法來指定任務的佇列：

```
<?php

namespace App\Jobs;

 use Illuminate\Bus\Queueable;
 use Illuminate\Contracts\Queue\ShouldQueue;
 use Illuminate\Foundation\Bus\Dispatchable;
 use Illuminate\Queue\InteractsWithQueue;
 use Illuminate\Queue\SerializesModels;

class ProcessPodcast implements ShouldQueue
{
     use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

     /**
      * 建立一個新的任務實例
      */
     public function __construct()
     {
          $this->onQueue('processing');
     }
}
```

32.4.5.2 調度到特定連接

如果你的應用程式與多個佇列連接互動，你可以使用 onConnection 方法指定將任務推送到哪個連接：

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * 儲存新的播客
     */
    public function store(Request $request): RedirectResponse
    {
        $podcast = Podcast::create(/* ... */);

        // 建立播客...

        ProcessPodcast::dispatch($podcast)->onConnection('sqs');

        return redirect('/podcasts');
    }
}
```

你可以將 `onConnection` 和 `onQueue` 方法連結在一起，以指定任務的連接和佇列：

```
ProcessPodcast::dispatch($podcast)
                ->onConnection('sqs')
                ->onQueue('processing');
```

或者，你可以通過在任務的建構函式中呼叫 `onConnection` 方法來指定任務的連接

```
<?php

namespace App\Jobs;

 use Illuminate\Bus\Queueable;
 use Illuminate\Contracts\Queue\ShouldQueue;
 use Illuminate\Foundation\Bus\Dispatchable;
 use Illuminate\Queue\InteractsWithQueue;
 use Illuminate\Queue\SerializesModels;

class ProcessPodcast implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    /**
     * 建立一個新的任務實例。
     */
    public function __construct()
    {
        $this->onConnection('sqs');
    }
}
```

32.4.6 指定最大任務嘗試 / 超時值

32.4.6.1 最大嘗試次數

如果你的一個佇列任務遇到了錯誤，你可能不希望無限制的重試。因此 Laravel 提供了各種方法來指定一個任務可以嘗試多少次或多長時間。

指定任務可嘗試的最大次數的其中一個方法是，通過 Artisan 命令列上的 `--tries` 開關。這將適用於調度作業的所有任務，除非正在處理的任務指定了最大嘗試次數。

```
php artisan queue:work --tries=3
```

如果一個任務超過其最大嘗試次數，將被視為「失敗」的任務。有關處理失敗任務的更多資訊，可以參考 處理失敗佇列。如果將 `--tries=0` 提供給 `queue:work` 命令，任務將無限期地重試。

你可以採取更細化的方法來定義任務類本身的最大嘗試次數。如果在任務上指定了最大嘗試次數，它將優先於命令列上提供的 `--tries` 開關設定的值：

```
<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
{
    /**
     * 任務可嘗試的次數。
     *
     * @var int
     */
    public $tries = 5;
}
```

32.4.6.2 基於時間的嘗試

除了定義任務失敗前嘗試的次數之外，還可以定義任務應該超時的時間。這允許在給定的時間範圍內嘗試任意次數的任務。要定義任務超時的時間，請在任務類中新增 `retryUntil` 方法。這個方法應返回一個 `DateTime` 實例：

```
use DateTime;

/**
 * 確定任務應該超時的時間。
 */
public function retryUntil(): DateTime
{
    return now()->addMinutes(10);
}
```

> **技巧**
> 你也可以在 佇列事件監聽器 上定義一個 `tries` 屬性或 `retryUntil` 方法。

32.4.6.3 最大嘗試

有時你可能希望指定一個任務可能會嘗試多次，但如果重試由給定數量的未處理異常觸發（而不是直接由 `release` 方法釋放），則應該失敗。為此，你可以在任務類上定義一個 `maxExceptions` 屬性：

```
<?php

namespace App\Jobs;

use Illuminate\Support\Facades\Redis;

class ProcessPodcast implements ShouldQueue
{
    /**
     * 可以嘗試任務的次數
     *
     * @var int
     */
    public $tries = 25;
```

```
    /**
     * 失敗前允許的最大未處理異常數
     *
     * @var int
     */
    public $maxExceptions = 3;

    /**
     * 執行
     */
    public function handle(): void
    {
        Redis::throttle('key')->allow(10)->every(60)->then(function () {
            // 獲得鎖，處理播客...
        }, function () {
            // 無法獲取鎖...
            return $this->release(10);
        });
    }
}
```

在此示例中，如果應用程式無法獲得 Redis 鎖，則該任務將在 10 秒後被釋放，並將繼續重試最多 25 次。但是，如果任務拋出三個未處理的異常，則任務將失敗。

32.4.6.4 超時

> **注意**
> 必須安裝 `pcntl` PHP 擴展以指定任務超時。

通常，你大致知道你預計排隊的任務需要多長時間。出於這個原因，Laravel 允許你指定一個「超時」值。 如果任務的處理時間超過超時值指定的秒數，則處理該任務的任務處理程序將退出並出現錯誤。 通常，任務程序將由在你的伺服器上組態的處理程序管理器自動重新啟動。

同樣，任務可以運行的最大秒數可以使用 Artisan 命令列上的 `--timeout` 開關來指定：

```
php artisan queue:work --timeout=30
```

如果任務因不斷超時而超過其最大嘗試次數，則它將被標記為失敗。

你也可以定義允許任務在任務類本身上運行的最大秒數。如果在任務上指定了超時，它將優先於在命令列上指定的任何超時:

```
<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
{
    /**
     * 在超時之前任務可以運行的秒數.
     *
     * @var int
     */
    public $timeout = 120;
}
```

有些時候，諸如 socket 或在 HTTP 連接之類的 IO 阻止處理程序可能不會遵守你指定的超時。因此，在使用這些功能時，也應始終嘗試使用其 API 指定超時。例如，在使用 Guzzle 時，應始終指定連接並請求的超時時間。

32.4.6.5 超時失敗

如果你希望在超時時將任務標記為 failed，可以在任務類上定義 `$failOnTimeout` 屬性：

```
/**
 * 標示是否應在超時時標記為失敗.
 *
 * @var bool
 */
public $failOnTimeout = true;
```

32.4.7 錯誤處理

如果在處理任務時拋出異常，任務將自動釋放回佇列，以便再次嘗試。 任務將繼續發佈，直到嘗試達到你的應用程式允許的最大次數為止。最大嘗試次數由 `queue:work` Artisan 命令中使用的 `--tries` 開關定義。或者，可以在任務類本身上定義最大嘗試次數。有關運行佇列處理器的更多資訊 可以在下面找到。

32.4.7.1 手動發佈

有時你可能希望手動將任務發佈回佇列，以便稍後再次嘗試。你可以通過呼叫 `release` 方法來完成此操作：

```
/**
 * 執行任務。
 */
public function handle(): void
{
    // ...

    $this->release();
}
```

默認情況下，`release` 方法會將任務發佈回佇列以供立即處理。但是，通過向 `release` 方法傳遞一個整數，你可以指示佇列在給定的秒數過去之前不使任務可用於處理：

```
$this->release(10);
```

32.4.7.2 手動使任務失敗

有時，你可能需要手動將任務標記為 「failed」 。為此，你可以呼叫 `fail` 方法：

```
/**
 * 執行任務。
 */
public function handle(): void
{
    // ...

    $this->fail();
}
```

如果你捕獲了一個異常，你想直接將你的任務標記為失敗，你可以將異常傳遞給 `fail` 方法。 或者，為方便起見，你可以傳遞一個字串來表示錯誤異常資訊：

```
$this->fail($exception);

$this->fail('Something went wrong.');
```

> **技巧**
> 有關失敗任務的更多資訊，請查看 處理任務失敗的文件.

32.5 任務批處理

Laravel 的任務批處理功能允許你輕鬆地執行一批任務，然後在這批任務執行完畢後執行一些操作。 在開始之前，你應該建立一個資料庫遷移以建構一個表來包含有關你的任務批次的元資訊，例如它們的完成百分比。

可以使用 queue:batches-table Artisan 命令來生成此遷移：

```
php artisan queue:batches-table

php artisan migrate
```

32.5.1 定義可批處理任務

要定義可批處理的任務，你應該像往常一樣建立可排隊的任務； 但是，你應該將 Illuminate\Bus\Batchable 特性新增到任務類中。 此 trait 提供對 batch 方法的訪問，該方法可用於檢索任務正在執行的當前批次：

```
<?php

namespace App\Jobs;

use Illuminate\Bus\Batchable;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;

class ImportCsv implements ShouldQueue
{
    use Batchable, Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    /**
     * 執行任務。
     */
    public function handle(): void
    {
        if ($this->batch()->cancelled()) {
            // 確定批次是否已被取消...

            return;
        }

        // 匯入 CSV 檔案的一部分...
    }
}
```

32.5.2 調度批次

要調度一批任務，你應該使用 Bus 門面的 batch 方法。 當然，批處理主要在與完成回呼結合使用時有用。 因此，你可以使用 then、catch 和 finally 方法來定義批處理的完成回呼。 這些回呼中的每一個在被呼叫時都會收到一個 Illuminate\Bus\Batch 實例。 在這個例子中，我們假設我們正在排隊一批任務，每個任務處理 CSV 檔案中給定數量的行：

```
use App\Jobs\ImportCsv;
use Illuminate\Bus\Batch;
use Illuminate\Support\Facades\Bus;
use Throwable;

$batch = Bus::batch([
    new ImportCsv(1, 100),
    new ImportCsv(101, 200),
    new ImportCsv(201, 300),
    new ImportCsv(301, 400),
    new ImportCsv(401, 500),
])->then(function (Batch $batch) {
    // 所有任務均已成功完成...
})->catch(function (Batch $batch, Throwable $e) {
```

```
    // 檢測到第一批任務失敗...
})->finally(function (Batch $batch) {
    // 批處理已完成執行...
})->dispatch();

return $batch->id;
```

批次的 ID 可以通過 `$batch->id` 屬性訪問，可用於 查詢 Laravel 命令匯流排 以獲取有關批次分派後的資訊。

> **注意**
> 由於批處理回呼是由 Laravel 佇列序列化並在稍後執行的，因此你不應在回呼中使用 `$this` 變數。

32.5.2.1 命名批次

Laravel Horizon 和 Laravel Telescope 等工具如果命名了批次，可能會為批次提供更使用者友好的偵錯資訊。要為批處理分配任意名稱，你可以在定義批處理時呼叫 `name` 方法：

```
$batch = Bus::batch([
    // ...
])->then(function (Batch $batch) {
    // 所有任務均已成功完成...
})->name('Import CSV')->dispatch();
```

32.5.2.2 批處理連接 & 佇列

如果你想指定應用於批處理任務的連接和佇列，你可以使用 `onConnection` 和 `onQueue` 方法。 所有批處理任務必須在相同的連接和佇列中執行：

```
$batch = Bus::batch([
    // ...
])->then(function (Batch $batch) {
    // 所有任務均已成功完成...
})->onConnection('redis')->onQueue('imports')->dispatch();
```

32.5.2.3 批內連結

你可以通過將連結的任務放在陣列中來在批處理中定義一組 連結的任務。 例如，我們可以平行執行兩個任務鏈，並在兩個任務鏈都完成處理後執行回呼：

```
use App\Jobs\ReleasePodcast;
use App\Jobs\SendPodcastReleaseNotification;
use Illuminate\Bus\Batch;
use Illuminate\Support\Facades\Bus;

Bus::batch([
    [
        new ReleasePodcast(1),
        new SendPodcastReleaseNotification(1),
    ],
    [
        new ReleasePodcast(2),
        new SendPodcastReleaseNotification(2),
    ],
])->then(function (Batch $batch) {
    // ...
})->dispatch();
```

32.5.3 批次新增任務

有些時候，批次向批處理中新增任務可能很有用。當你需要批次處理數千個任務時，這種模式非常好用，而這些任務在 Web 請求期間可能需要很長時間才能調度。因此，你可能希望調度初始批次的「載入器」任務，這些

任務與更多工相結合：

```
$batch = Bus::batch([
    new LoadImportBatch,
    new LoadImportBatch,
    new LoadImportBatch,
])->then(function (Batch $batch) {
    // 所有任務都成功完成...
})->name('Import Contacts')->dispatch();
```

在這個例子中，我們將使用 `LoadImportBatch` 實例為批處理新增其他任務。為了實現這個功能，我們可以對批處理實例使用 `add` 方法，該方法可以通過 `batch` 實例訪問：

```
use App\Jobs\ImportContacts;
use Illuminate\Support\Collection;

/**
 * 執行任務。
 */
public function handle(): void
{
    if ($this->batch()->cancelled()) {
        return;
    }

    $this->batch()->add(Collection::times(1000, function () {
        return new ImportContacts;
    }));
}
```

> **注意**
> 你只能將任務新增到當前任務所屬的批處理中。

32.5.4 校驗批處理

為批處理完成後提供回呼的 `Illuminate\Bus\Batch` 實例中具有多種屬性和方法，可以幫助你與指定的批處理業務進行互動和檢查：

```
// 批處理的 UUID...
$batch->id;

// 批處理的名稱（如果已經設定的話）...
$batch->name;

// 分配給批處理的任務數量...
$batch->totalJobs;

// 佇列還沒處理的任務數量...
$batch->pendingJobs;

// 失敗的任務數量...
$batch->failedJobs;

// 到目前為止已經處理的任務數量...
$batch->processedJobs();

// 批處理已經完成的百分比（0-100）...
$batch->progress();

// 批處理是否已經完成執行...
$batch->finished();

// 取消批處理的運行...
$batch->cancel();
```

```
// 批處理是否已經取消...
$batch->cancelled();
```

32.5.4.1 從路由返回批次

所有 `Illuminate\Bus\Batch` 實例都是 JSON 可序列化的，這意味著你可以直接從應用程式的一個路由返回它們，以檢索包含有關批處理的資訊的 JSON 有效負載，包括其完成進度。這樣可以方便地在應用程式的 UI 中顯示有關批處理完成進度的資訊。

要通過 ID 檢索批次，你可以使用 `Bus` 外觀的 `findBatch` 方法：

```
use Illuminate\Support\Facades\Bus;
use Illuminate\Support\Facades\Route;

Route::get('/batch/{batchId}', function (string $batchId) {
    return Bus::findBatch($batchId);
});
```

32.5.5 取消批次

有時你可能需要取消給定批處理的執行。這可以通過呼叫 `Illuminate\Bus\Batch` 實例的 `cancel` 方法來完成：

```
/**
 * 執行任務。
 */
public function handle(): void
{
    if ($this->user->exceedsImportLimit()) {
        return $this->batch()->cancel();
    }

    if ($this->batch()->cancelled()) {
        return;
    }
}
```

正如你在前面的示例中可能已經注意到的那樣，批處理任務通常應在繼續執行之前確定其相應的批處理是否已被取消。 但是，為了方便起見，你可以將 `SkipIfBatchCancelled` 中介軟體 分配給作業。 顧名思義，如果相應的批次已被取消，此中介軟體將指示 Laravel 不處理該作業：

```
use Illuminate\Queue\Middleware\SkipIfBatchCancelled;

/**
 * 獲取任務應通過的中介軟體。
 */
public function middleware(): array
{
    return [new SkipIfBatchCancelled];
}
```

32.5.6 批處理失敗

當批處理任務失敗時，將呼叫 `catch` 回呼（如果已分配）。此回呼僅針對批處理中失敗的第一個任務呼叫。

32.5.6.1 允許失敗

當批處理中的某個任務失敗時，Laravel 會自動將該批處理標記為「已取消」。如果你願意，你可以停用此行為，以便任務失敗不會自動將批處理標記為已取消。這可以通過在調度批處理時呼叫 `allowFailures` 方法來完成：

```
$batch = Bus::batch([
    // ...
])->then(function (Batch $batch) {
    // 所有任務均已成功完成...
})->allowFailures()->dispatch();
```

32.5.6.2 重試失敗的批處理任務

為方便起見，Laravel 提供了一個 `queue:retry-batch` Artisan 命令，允許你輕鬆重試給定批次的所有失敗任務。 `queue:retry-batch` 命令接受應該重試失敗任務的批處理的 UUID：

```
php artisan queue:retry-batch 32dbc76c-4f82-4749-b610-a639fe0099b5
```

32.5.7 修剪批次

如果不進行修剪，`job_batches` 表可以非常快速地積累記錄。為了緩解這種情況，你應該 schedule `queue:prune-batches` Artisan 命令每天運行：

```
$schedule->command('queue:prune-batches')->daily();
```

默認情況下，將修剪所有超過 24 小時的已完成批次。你可以在呼叫命令時使用 `hours` 選項來確定保留批處理資料的時間。例如，以下命令將刪除 48 小時前完成的所有批次：

```
$schedule->command('queue:prune-batches --hours=48')->daily();
```

有時，你的 `jobs_batches` 表可能會累積從未成功完成的批次的批次記錄，例如任務失敗且該任務從未成功重試的批次。 你可以使用 `unfinished` 選項指示 `queue:prune-batches` 命令修剪這些未完成的批處理記錄：

```
$schedule->command('queue:prune-batches --hours=48 --unfinished=72')->daily();
```

同樣，你的 `jobs_batches` 表也可能會累積已取消批次的批次記錄。 你可以使用 `cancelled` 選項指示 `queue:prune-batches` 命令修剪這些已取消的批記錄：

```
$schedule->command('queue:prune-batches --hours=48 --cancelled=72')->daily();
```

32.6 佇列閉包

除了將任務類分派到佇列之外，你還可以分派一個閉包。這對於需要在當前請求週期之外執行的快速、簡單的任務非常有用。當向佇列分派閉包時，閉包的程式碼內容是加密簽名的，因此它不能在傳輸過程中被修改：

```
$podcast = App\Podcast::find(1);

dispatch(function () use ($podcast) {
    $podcast->publish();
});
```

使用 `catch` 方法，你可以提供一個閉包，如果排隊的閉包在耗盡所有佇列的組態的重試次數 後未能成功完成，則應執行該閉包：

```
use Throwable;

dispatch(function () use ($podcast) {
    $podcast->publish();
})->catch(function (Throwable $e) {
    // 這個任務失敗了...
});
```

> **注意**
> 由於 `catch` 回呼由 Laravel 佇列稍後序列化並執行，因此你不應在 `catch` 回呼中使用 `$this` 變數。

32.7 運行佇列工作者

32.7.1 queue:work 命令

Laravel 包含一個 Artisan 命令，該命令將啟動佇列處理程序並在新任務被推送到佇列時處理它們。 你可以使用 queue:work Artisan 命令運行任務處理程序。請注意，一旦 queue:work 命令啟動，它將繼續運行，直到手動停止或關閉終端：

```
php artisan queue:work
```

> **技巧**
> 要保持 queue:work 處理程序在後台永久運行，你應該使用 Supervisor 等處理程序監視器來確保佇列工作處理程序不會停止運行。

如果你希望處理的任務 ID 包含在命令的輸出中，則可以在呼叫 queue:work 命令時包含 -v 標誌：

```
php artisan queue:work -v
```

請記住，佇列任務工作者是長期存在的處理程序，並將啟動的應用程式狀態儲存在記憶體中。 因此，他們在啟動後不會注意到你的程式碼庫中的更改。 因此，在你的部署過程中，請務必重新啟動你的任務佇列處理程序。 此外，請記住，你的應用程式建立或修改的任何靜態狀態都不會在任務啟動之間自動重設。

或者，你可以運行 queue:listen 命令。 使用 queue:listen 命令時，當你想要重新載入更新後的程式碼或重設應用程式狀態時，無需手動重啟 worker； 但是，此命令的效率明顯低於 queue:work 命令：

```
php artisan queue:listen
```

32.7.1.1 運行多個佇列處理程序

要將多個 worker 分配到一個佇列並同時處理任務，你應該簡單地啟動多個 queue:work 處理程序。 這可以通過終端中的多個選項卡在本地完成，也可以使用流程管理器的組態設定在生產環境中完成。 使用 Supervisor 時，你可以使用 numprocs 組態值。

32.7.1.2 指定連接 & 佇列

你還可以指定工作人員應使用哪個佇列連接。 傳遞給 work 命令的連接名稱應對應於 config/queue.php 組態檔案中定義的連接之一：

```
php artisan queue:work redis
```

默認情況下，queue:work 命令只處理給定連接上默認佇列的任務。 但是，你可以通過僅處理給定連接的特定佇列來進一步自訂你的佇列工作者。 例如，如果你的所有電子郵件都在你的 redis 佇列連接上的 emails 佇列中處理，你可以發出以下命令來啟動只處理該佇列的工作程序：

```
php artisan queue:work redis --queue=emails
```

32.7.1.3 Processing A Specified Number Of Jobs

--once 選項可用於指定處理程序僅處理佇列中的單個任務

```
php artisan queue:work --once
```

--max-jobs 選項可用於指示 worker 處理給定數量的任務然後退出。 此選項在與 Supervisor 結合使用時可能很有用，這樣你的工作人員在處理給定數量的任務後會自動重新啟動，釋放他們可能積累的任何記憶體：

```
php artisan queue:work --max-jobs=1000
```

32.7.1.4 處理所有排隊的任務然後退出

`--stop-when-empty` 選項可用於指定處理程序處理所有作業，然後正常退出。如果你希望在佇列為空後關閉容器，則此選項在處理 Docker 容器中的 Laravel 佇列時很有用

```
php artisan queue:work --stop-when-empty
```

32.7.1.5 在給定的秒數內處理任務

`--max-time` 選項可用於指示處理程序給定的秒數內處理作業，然後退出。 當與 Supervisor 結合使用時，此選項可能很有用，這樣你的工作人員在處理作業給定時間後會自動重新啟動，釋放他們可能積累的任何記憶體：

```
# 處理處理程序一小時，然後退出...
php artisan queue:work --max-time=3600
```

32.7.1.6 處理程序睡眠時間

當佇列中有任務可用時，處理程序將繼續處理作業，而不會在它們之間產生延遲。但是，`sleep` 選項決定了如果沒有可用的新任務，處理程序將 `sleep` 多少秒。 睡眠時，處理程序不會處理任何新的作業 - 任務將在處理程序再次喚醒後處理。

```
php artisan queue:work --sleep=3
```

32.7.1.7 資源注意事項

守護處理程序佇列在處理每個任務之前不會 `reboot` 框架。因此，你應該在每個任務完成後釋放所有繁重的資源。例如，如果你正在使用 GD 庫進行圖像處理，你應該在處理完圖像後使用 `imagedestroy` 釋放記憶體。

32.7.2 佇列優先順序

有時你可能希望優先處理佇列的處理方式。例如，在 `config/queue.php` 組態檔案中，你可以將 `redis` 連接的默認 `queue` 設定為 `low`。 但是，有時你可能希望將作業推送到 `high` 優先順序佇列，如下所示：

```
dispatch((new Job)->onQueue('high'));
```

要啟動一個處理程序，在繼續處理 `low` 佇列上的任何任務之前驗證所有 `high` 佇列任務是否已處理，請將佇列名稱的逗號分隔列表傳遞給 `work` 命令：

```
php artisan queue:work --queue=high,low
```

32.7.3 佇列處理程序 & 部署

由於佇列任務是長期存在的處理程序，如果不重新啟動，他們不會注意到程式碼的更改。因此，使用佇列任務部署應用程式的最簡單方法是在部署過程中重新啟動任務。你可以通過發出 `queue:restart` 命令優雅地重新啟動所有處理程序：

```
php artisan queue:restart
```

此命令將指示所有佇列處理程序在處理完當前任務後正常退出，以免丟失現有任務。由於佇列任務將在執行 `queue:restart` 命令時退出，你應該運行諸如 Supervisor 之類的處理程序管理器來自動重新啟動佇列任務。

> **注意** 佇列使用 cache 來儲存重啟訊號，因此你應該在使用此功能之前驗證是否為你的應用程式正確組態了快取驅動程式。

32.7.4 任務到期 & 超時

32.7.4.1 任務到期

在 `config/queue.php` 組態檔案中，每個佇列連接都定義了一個 `retry_after` 選項。該選項指定佇列連接在重試正在處理的作業之前應該等待多少秒。例如，如果 `retry_after` 的值設定為 `90`，如果作業已經處理了 90 秒而沒有被釋放或刪除，則該作業將被釋放回佇列。通常，你應該將 `retry_after` 值設定為作業完成處理所需的最大秒數。

> **警告** 唯一不包含 `retry_after` 值的佇列連接是 Amazon SQS。SQS 將根據 AWS 控制台內管理的 默認可見性超時 重試作業。

32.7.4.2 處理程序超時

`queue:work` Artisan 命令公開了一個`--timeout` 選項。默認情況下，`--timeout` 值為 60 秒。如果任務的處理時間超過超時值指定的秒數，則處理該任務的處理程序將退出並出現錯誤。通常，工作程序將由 你的伺服器上組態的處理程序管理器 自動重新啟動：

```
php artisan queue:work --timeout=60
```

`retry_after` 組態選項和 `--timeout` CLI 選項是不同的，但它們協同工作以確保任務不會丟失並且任務僅成功處理一次。 > **警告** > `--timeout` 值應始終比 `retry_after` 組態值至少短幾秒鐘。 這將確保處理凍結任務的處理程序始終在重試任務之前終止。 如果你的 `--timeout` 選項比你的 `retry_after` 組態值長，你的任務可能會被處理兩次。

32.8 Supervisor 組態

在生產中，你需要一種方法來保持 `queue:work` 處理程序運行。 `queue:work` 處理程序可能會因多種原因停止運行，例如超過 worker 超時或執行 `queue:restart` 命令。 出於這個原因，你需要組態一個處理程序監視器，它可以檢測你的 `queue:work` 處理程序何時退出並自動重新啟動它們。此外，處理程序監視器可以讓你指定要同時運行多少個 `queue:work` 處理程序。Supervisor 是 Linux 環境中常用的處理程序監視器，我們將在下面的文件中討論如何組態它。

32.8.1.1 安裝 Supervisor

Supervisor 是 Linux 作業系統的處理程序監視器，如果它們失敗，它將自動重新啟動你的 `queue:work` 處理程序。要在 Ubuntu 上安裝 Supervisor，你可以使用以下命令：

```
sudo apt-get install supervisor
```

> **注意** 如果你自己組態和管理 Supervisor 聽起來很費力，請考慮使用 Laravel Forge，它會自動為你的生產 Laravel 項目安裝和組態 Supervisor。

32.8.1.2 組態 Supervisor

Supervisor 組態檔案通常儲存在 `/etc/supervisor/conf.d` 目錄中。在這個目錄中，你可以建立任意數量的組態檔案來指示 Supervisor 應該如何監控你的處理程序。例如，讓我們建立一個啟動和監控 `queue:work` 處理程序的 `laravel-worker.conf` 檔案：

```
[program:laravel-worker]
process_name=%(program_name)s_%(process_num)02d
command=php /home/forge/app.com/artisan queue:work sqs --sleep=3 --tries=3 --max-
```

```
time=3600
autostart=true
autorestart=true
stopasgroup=true
killasgroup=true
user=forge
numprocs=8
redirect_stderr=true
stdout_logfile=/home/forge/app.com/worker.log
stopwaitsecs=3600
```

在這個例子中，`numprocs` 指令將指示 Supervisor 運行 8 個 `queue:work` 處理程序並監控所有處理程序，如果它們失敗則自動重新啟動它們。你應該更改組態的「命令」指令以反映你所需的佇列連接和任務選項。

> **警告** 你應該確保 `stopwaitsecs` 的值大於執行階段間最長的作業所消耗的秒數。否則，Supervisor 可能會在作業完成處理之前將其終止。

32.8.1.3 開始 Supervisor

建立組態檔案後，你可以使用以下命令更新 Supervisor 組態並啟動處理程序：

```
sudo supervisorctl reread

sudo supervisorctl update

sudo supervisorctl start laravel-worker:*
```

有關 Supervisor 的更多資訊，請參閱 Supervisor 文件。

32.9 處理失敗的任務

有時，你佇列的任務會失敗。別擔心，事情並不總是按計畫進行！ Laravel 提供了一種方便的方法來 指一個任務應該嘗試的最大次數。在非同步任務超過此嘗試次數後，它將被插入到 `failed_jobs` 資料庫表中。 失敗的 同步調度的任務 不儲存在此表中，它們的異常由應用程式立即處理。

建立 `failed_jobs` 表的遷移通常已經存在於新的 Laravel 應用程式中。但是，如果你的應用程式不包含此表的遷移，你可以使用 `queue:failed-table` 命令來建立遷移：

```
php artisan queue:failed-table

php artisan migrate
```

運行 queue worker 處理程序時，你可以使用 `queue:work` 命令上的 `--tries` 開關指定任務應嘗試的最大次數。如果你沒有為 `--tries` 選項指定值，則作業將僅嘗試一次或與任務類的 `$tries` 屬性指定的次數相同：

```
php artisan queue:work redis --tries=3
```

使用 `--backoff` 選項，你可以指定 Laravel 在重試遇到異常的任務之前應該等待多少秒。默認情況下，任務會立即釋放回佇列，以便可以再次嘗試：

```
php artisan queue:work redis --tries=3 --backoff=3
```

如果你想組態 Laravel 在重試每個任務遇到異常的任務之前應該等待多少秒，你可以通過在你的任務類上定義一個 `backoff` 屬性來實現：

```
/**
 * 重試任務前等待的秒數
 *
 * @var int
 */
public $backoff = 3;
```

如果你需要更複雜的邏輯來確定任務的退避時間，你可以在你的任務類上定義一個 `backoff` 方法：

```
/**
* 計算重試任務之前要等待的秒數
*/
public function backoff(): int
{
    return 3;
}
```

你可以通過從 backoff 方法返回一組退避值來輕鬆組態 “exponential” 退避。在此示例中，第一次重試的重試延遲為 1 秒，第二次重試為 5 秒，第三次重試為 10 秒：

```
/**
* 計算重試任務之前要等待的秒數
*
* @return array<int, int>
*/
public function backoff(): array
{
    return [1, 5, 10];
}
```

32.9.1 任務失敗後清理

當特定任務失敗時，你可能希望向使用者傳送警報或恢復該任務部分完成的任何操作。為此，你可以在任務類上定義一個 failed 方法。導致作業失敗的 Throwable 實例將被傳遞給 failed 方法：

```
<?php

namespace App\Jobs;

use App\Models\Podcast;
use App\Services\AudioProcessor;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;
use Throwable;

class ProcessPodcast implements ShouldQueue
{
    use InteractsWithQueue, Queueable, SerializesModels;

    /**
     * 建立新任務實例
     */
    public function __construct(
        public Podcast $podcast,
    ) {}

    /**
     * 執行任務
     */
    public function handle(AudioProcessor $processor): void
    {
        // 處理上傳的播客...
    }

    /**
     * 處理失敗作業
     */
    public function failed(Throwable $exception): void
    {
        // 向使用者傳送失敗通知等...
    }
}
```

注意
在呼叫 failed 方法之前實例化任務的新實例；因此，在 handle 方法中可能發生的任何類屬性修改都將丟失。

32.9.2 重試失敗的任務

要查看已插入到你的 failed_jobs 資料庫表中的所有失敗任務，你可以使用 queue:failed Artisan 命令：

```
php artisan queue:failed
```

queue:failed 命令將列出任務 ID、連接、佇列、失敗時間和有關任務的其他資訊。任務 ID 可用於重試失敗的任務。例如，要重試 ID 為 ce7bb17c-cdd8-41f0-a8ec-7b4fef4e5ece 的失敗任務，請發出以下命令：

```
php artisan queue:retry ce7bb17c-cdd8-41f0-a8ec-7b4fef4e5ece
```

如有必要，可以向命令傳遞多個 ID:

```
php artisan queue:retry ce7bb17c-cdd8-41f0-a8ec-7b4fef4e5ece 91401d2c-0784-4f43-824c-34f94a33c24d
```

還可以重試指定佇列的所有失敗任務：

```
php artisan queue:retry --queue=name
```

重試所有失敗任務，可以執行 queue:retry 命令，並將 all 作為 ID 傳遞：

```
php artisan queue:retry all
```

如果要刪除指定的失敗任務，可以使用 queue:forget 命令：

```
php artisan queue:forget 91401d2c-0784-4f43-824c-34f94a33c24d
```

技巧
使用 Horizon 時，應該使用 Horizon:forget 命令來刪除失敗任務，而不是 queue:forget 命令。

刪除 failed_jobs 表中所有失敗任務，可以使用 queue:flush 命令:

```
php artisan queue:flush
```

32.9.3 忽略缺失的模型

向任務中注入 Eloquent 模型時，模型會在注入佇列之前自動序列化，並在處理任務時從資料庫中重新檢索。但是，如果在任務等待消費時刪除了模型，則任務可能會失敗，拋出 ModelNotFoundException 異常。

為方便起見，可以把將任務的 deleteWhenMissingModels 屬性設定為 true，這樣會自動刪除缺少模型的任務。當此屬性設定為 true 時，Laravel 會放棄該任務，並且不會引發異常：

```
/**
 * 如果任務的模型不存在，則刪除該任務
 *
 * @var bool
 */
public $deleteWhenMissingModels = true;
```

32.9.4 刪除失敗的任務

你可以通過呼叫 queue:prune-failed Artisan 命令刪除應用程式的 failed_jobs 表中的所有記錄：

```
php artisan queue:prune-failed
```

默認情況下，將刪除所有超過 24 小時的失敗任務記錄，如果為命令提供 --hours 選項，則僅保留在過去 N

小時內插入的失敗任務記錄。例如，以下命令將刪除超過 48 小時前插入的所有失敗任務記錄：

```
php artisan queue:prune-failed --hours=48
```

32.9.5 在 DynamoDB 中儲存失敗的任務

Laravel 還支援將失敗的任務記錄儲存在 DynamoDB 而不是關聯式資料庫表中。但是，你必須建立一個 DynamoDB 表來儲存所有失敗的任務記錄。通常，此表應命名為 `failed_jobs`，但你應根據應用程式的 `queue` 組態檔案中的 `queue.failed.table` 組態值命名該表。

`failed_jobs` 表應該有一個名為 `application` 的字串主分區鍵和一個名為 uuid 的字串主排序鍵。鍵的 `application` 部分將包含應用程式的名稱，該名稱由應用程式的 `app` 組態檔案中的 `name` 組態值定義。由於應用程式名稱是 DynamoDB 表鍵的一部分，因此你可以使用同一個表來儲存多個 Laravel 應用程式的失敗任務。

此外，請確保你安裝了 AWS 開發工具包，以便你的 Laravel 應用程式可以與 Amazon DynamoDB 通訊：

```
composer require aws/aws-sdk-php
```

接下來，`queue.failed.driver` 組態選項的值設定為 `dynamodb`。此外，你應該在失敗的作業組態陣列中定義 `key`、`secret` 和 `region` 組態選項。 這些選項將用於向 AWS 進行身份驗證。 當使用 `dynamodb` 驅動程式時，`queue.failed.database` 組態選項不是必須的：

```
'failed' => [
    'driver' => env('QUEUE_FAILED_DRIVER', 'dynamodb'),
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
    'table' => 'failed_jobs',
],
```

32.9.6 停用失敗的任務儲存

你可以通過將 `queue.failed.driver` 組態選項的值設定為 `null` 來指示 Laravel 丟棄失敗的任務而不儲存它們。通過 `QUEUE_FAILED_DRIVER` 環境變數來完成：

```
QUEUE_FAILED_DRIVER=null
```

32.9.7 失敗的任務事件

如果你想註冊一個在作業失敗時呼叫的事件監聽器，你可以使用 `Queue` facade 的 failing 方法。例如，我們可以從 Laravel 中包含的 `AppServiceProvider` 的 `boot` 方法為這個事件附加一個閉包：

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Queue;
use Illuminate\Support\ServiceProvider;
use Illuminate\Queue\Events\JobFailed;

class AppServiceProvider extends ServiceProvider
{
    /**
     * 註冊任何應用程式服務
     */
    public function register(): void
    {
        // ...
    }

    /**
```

```
     * 引導任何應用程式服務
     */
    public function boot(): void
    {
        Queue::failing(function (JobFailed $event) {
            // $event->connectionName
            // $event->job
            // $event->exception
        });
    }
}
```

32.10 從佇列中清除任務

> **技巧** 使用 Horizon 時，應使用 `horizon:clear` 命令從佇列中清除作業，而不是使用 `queue:clear` 命令。

如果你想從默認連接的默認佇列中刪除所有任務，你可以使用 `queue:clear` Artisan 命令來執行此操作：

```
php artisan queue:clear
```

你還可以提供 `connection` 參數和 `queue` 選項以從特定連接和佇列中刪除任務：

```
php artisan queue:clear redis --queue=emails
```

> **注意** 從佇列中清除任務僅適用於 SQS、Redis 和資料庫佇列驅動程式。 此外，SQS 消息刪除過程最多需要 60 秒，因此在你清除佇列後 60 秒內傳送到 SQS 佇列的任務也可能會被刪除。

32.11 監控你的佇列

如果你的佇列突然湧入了大量的任務，它會導致佇列任務繁重，從而增加了任務的完成時間，想你所想，Laravel 可以在佇列執行超過設定的閾值時候提醒你。

在開始之前， 你需要通過 `queue:monitor` 命令組態它 每分鐘執行一次。這個命令可以設定任務的名稱，以及你想要設定的任務閾值：

```
php artisan queue:monitor redis:default,redis:deployments --max=100
```

當你的任務超過設定閾值時候，僅通過這個方法還不足以觸發通知，此時會觸發一個 `Illuminate\Queue\Events\QueueBusy` 事件。你可以在你的應用 `EventServiceProvider` 來監聽這個事件，從而將監聽結果通知給你的開發團隊：

```
use App\Notifications\QueueHasLongWaitTime;
use Illuminate\Queue\Events\QueueBusy;
use Illuminate\Support\Facades\Event;
use Illuminate\Support\Facades\Notification;

/**
 * 為你的應用程式註冊其他更多事件
 */
public function boot(): void
{
    Event::listen(function (QueueBusy $event) {
        Notification::route('mail', 'dev@example.com')
                ->notify(new QueueHasLongWaitTime(
                    $event->connection,
                    $event->queue,
                    $event->size
                ));
    });
```

```
}
```

32.12 測試

當測試調度任務的程式碼時，你可能希望指示 Laravel 不要實際執行任務本身，因為任務的程式碼可以直接和獨立於調度它的程式碼進行測試。 當然，要測試任務本身，你可以實例化一個任務實例並在測試中直接呼叫 `handle` 方法。

你可以使用 `Queue` facade 的 `fake` 方法來防止排隊的任務實際被推送到佇列中。 在呼叫 `Queue` facade 的 `fake` 方法後，你可以斷言應用程式試圖將任務推送到佇列中：

```
<?php

namespace Tests\Feature;

use App\Jobs\AnotherJob;
use App\Jobs\FinalJob;
use App\Jobs\ShipOrder;
use Illuminate\Support\Facades\Queue;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_orders_can_be_shipped(): void
    {
        Queue::fake();

        // 執行訂單發貨...

        // 斷言沒有任務被推送......
        Queue::assertNothingPushed();

        // 斷言一個任務被推送到一個給定的佇列...
        Queue::assertPushedOn('queue-name', ShipOrder::class);

        // 斷言任務被推了兩次...
        Queue::assertPushed(ShipOrder::class, 2);

        // 斷言任務沒有被推送...
        Queue::assertNotPushed(AnotherJob::class);

        // 斷言閉包被推送到佇列中...
        Queue::assertClosurePushed();
    }
}
```

你可以將閉包傳遞給 `assertPushed` 或 `assertNotPushed` 方法，以斷言已推送通過給定「真實性測試」的任務。 如果至少有一項任務被推送並通過了給定的真值測試，則斷言將成功：

```
Queue::assertPushed(function (ShipOrder $job) use ($order) {
    return $job->order->id === $order->id;
});
```

32.12.1 偽造任務的一個子集

如果你只需要偽造特定的任務，同時允許你的其他任務正常執行，你可以將應該偽造的任務的類名傳遞給 fake 方法：

```
public function test_orders_can_be_shipped(): void
{
    Queue::fake([
        ShipOrder::class,
    ]);
```

```
    // 執行訂單發貨...

    // 斷言任務被推了兩次......
    Queue::assertPushed(ShipOrder::class, 2);
}
```

你可以使用 except 方法偽造除一組指定任務之外的所有任務：

```
Queue::fake()->except([
    ShipOrder::class,
]);
```

32.12.2 測試任務鏈

要測試任務鏈，你需要利用 Bus 外觀的偽造功能。 Bus 門面的 assertChained 方法可用於斷言 任務鏈 已被分派。 assertChained 方法接受一個鏈式任務陣列作為它的第一個參數：

```
use App\Jobs\RecordShipment;
use App\Jobs\ShipOrder;
use App\Jobs\UpdateInventory;
use Illuminate\Support\Facades\Bus;

Bus::fake();

// ...

Bus::assertChained([
    ShipOrder::class,
    RecordShipment::class,
    UpdateInventory::class
]);
```

正如你在上面的示例中看到的，鏈式任務陣列可能是任務類名稱的陣列。 但是，你也可以提供一組實際的任務實例。 這樣做時，Laravel 將確保任務實例屬於同一類，並且與你的應用程式調度的鏈式任務具有相同的屬性值：

```
Bus::assertChained([
    new ShipOrder,
    new RecordShipment,
    new UpdateInventory,
]);
```

你可以使用 assertDispatchedWithoutChain 方法來斷言一個任務是在沒有任務鏈的情況下被推送的：

```
Bus::assertDispatchedWithoutChain(ShipOrder::class);
```

32.12.3 測試任務批處理

Bus 門面的 assertBatched 方法可用於斷言 批處理任務 已分派。 給 assertBatched 方法的閉包接收一個 Illuminate\Bus\PendingBatch 的實例，它可用於檢查批處理中的任務：

```
use Illuminate\Bus\PendingBatch;
use Illuminate\Support\Facades\Bus;

Bus::fake();

// ...

Bus::assertBatched(function (PendingBatch $batch) {
    return $batch->name == 'import-csv' &&
           $batch->jobs->count() === 10;
});
```

32.12.3.1 測試任務 / 批處理互動

此外，你可能偶爾需要測試單個任務與其基礎批處理的互動。 例如，你可能需要測試任務是否取消了對其批次的進一步處理。 為此，你需要通過 `withFakeBatch` 方法為任務分配一個假批次。 `withFakeBatch` 方法返回一個包含任務實例和假批次的元組：

```
[$job, $batch] = (new ShipOrder)->withFakeBatch();

$job->handle();

$this->assertTrue($batch->cancelled());
$this->assertEmpty($batch->added);
```

32.13 任務事件

使用 `Queue` facade 上的 `before` 和 `after` 方法，你可以指定要在處理排隊任務之前或之後執行的回呼。 這些回呼是為儀表板執行額外日誌記錄或增量統計的絕佳機會。 通常，你應該從 服務提供者 的 `boot` 方法中呼叫這些方法。 例如，我們可以使用 Laravel 自帶的 `AppServiceProvider`：

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Queue;
use Illuminate\Support\ServiceProvider;
use Illuminate\Queue\Events\JobProcessed;
use Illuminate\Queue\Events\JobProcessing;

class AppServiceProvider extends ServiceProvider
{
    /**
     * 註冊任何應用程式服務。
     */
    public function register(): void
    {
        // ...
    }

    /**
     * 引導任何應用程式服務。
     */
    public function boot(): void
    {
        Queue::before(function (JobProcessing $event) {
            // $event->connectionName
            // $event->job
            // $event->job->payload()
        });

        Queue::after(function (JobProcessed $event) {
            // $event->connectionName
            // $event->job
            // $event->job->payload()
        });
    }
}
```

通過使用 `Queue` facade 的 `looping` 方法 ，你可以在 worker 嘗試從佇列獲取任務之前執行指定的回呼。例如，你可以註冊一個閉包，用以回滾之前失敗任務打開的任何事務：

```
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Facades\Queue;

Queue::looping(function () {
```

```
    while (DB::transactionLevel() > 0) {
        DB::rollBack();
    }
});
```

33 限流

33.1 簡介

Laravel 包含了一個開箱即用的，基於 快取 實現的限流器，提供了一個簡單的方法來限制指定時間內的任何操作。

> **技巧** 瞭解更多關於如何限制 HTTP 請求，請參考 請求頻率限制中介軟體。

33.1.1 快取組態

通常情況下，限流器使用你默認的快取驅動，由 `cache` 組態檔案中的 `default` 鍵定義。你也可以通過在你的應用程式的 `cache` 組態檔案中定義一個 `limiter` 來指定限流器應該使用哪一個快取來驅動：

```
'default' => 'memcached',

'limiter' => 'redis',
```

33.2 基本用法

可以通過 `Illuminate\Support\Facades\RateLimiter` 來操作限流器。限流器提供的最簡單的方法是 `attempt` 方法，它將一個給定的回呼函數執行次數限制在一個給定的秒數內。

當回呼函數執行次數超過限制時， `attempt` 方法返回 `false` ；否則， `attempt` 方法將返回回呼的結果或 `true` 。 `attempt` 方法接受的第一個參數是一個速率限制器 「key」 ，它可以是你選擇的任何字串，代表被限制速率的動作：

```
use Illuminate\Support\Facades\RateLimiter;

$executed = RateLimiter::attempt(
     'send-message:'.$user->id,
     $perMinute = 5,
     function() {
          // 傳送消息...
     }
);

if (! $executed) {
  return 'Too many messages sent!';
}
```

33.2.1 手動組態嘗試次數

如果您想手動與限流器互動，可以使用多種方法。例如，您可以呼叫 `tooManyAttempts` 方法來確定給定的限流器是否超過了每分鐘允許的最大嘗試次數：

```
use Illuminate\Support\Facades\RateLimiter;

if (RateLimiter::tooManyAttempts('send-message:'.$user->id, $perMinute = 5)) {
     return 'Too many attempts!';
}
```

或者，您可以使用 `remaining` 方法檢索給定金鑰的剩餘嘗試次數。如果給定的金鑰還有重試次數，您可以呼

叫 hit 方法來增加總嘗試次數：

```
use Illuminate\Support\Facades\RateLimiter;

if (RateLimiter::remaining('send-message:'.$user->id, $perMinute = 5)) {
     RateLimiter::hit('send-message:'.$user->id);

     // 傳送消息...
}
```

33.2.1.1 確定限流器可用性

當一個鍵沒有更多的嘗試次數時，availableIn 方法返回在嘗試可用之前需等待的剩餘秒數：

```
use Illuminate\Support\Facades\RateLimiter;

if (RateLimiter::tooManyAttempts('send-message:'.$user->id, $perMinute = 5)) {
     $seconds = RateLimiter::availableIn('send-message:'.$user->id);

     return 'You may try again in '.$seconds.' seconds.';
}
```

33.2.2 清除嘗試次數

您可以使用 clear 方法重設給定速率限制鍵的嘗試次數。例如，當接收者讀取給定消息時，您可以重設嘗試次數：

```
use App\Models\Message;
use Illuminate\Support\Facades\RateLimiter;

/**
 * 標記消息為已讀。
 */
public function read(Message $message): Message
{
     $message->markAsRead();

     RateLimiter::clear('send-message:'.$message->user_id);

     return $message;
}
```

34 任務調度

34.1 簡介

過去，你可能需要在伺服器上為每一個調度任務去建立 Cron 條目。因為這些任務的調度不是通過程式碼控制的，你要查看或新增任務調度都需要通過 SSH 遠端登錄到伺服器上去操作，所以這種方式很快會讓人變得痛苦不堪。

Laravel 的命令列調度器允許你在 Laravel 中清晰明了地定義命令調度。在使用這個任務調度器時，你只需要在你的伺服器上建立單個 Cron 入口。你的任務調度在 `app/Console/Kernel.php` 的 `schedule` 方法中進行定義。為了幫助你更好的入門，這個方法中有個簡單的例子。

34.2 定義調度

你可以在 `App\Console\Kernel` 類的 `schedule` 方法中定義所有的調度任務。在開始之前，我們來看一個例子：我們計畫每天午夜執行一個閉包，這個閉包會執行一次資料庫語句去清空一張表：

```
<?php

namespace App\Console;

use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;
use Illuminate\Support\Facades\DB;

class Kernel extends ConsoleKernel
{
    /**
     * 定義應用中的命令調度
     */
    protected function schedule(Schedule $schedule): void
    {
        $schedule->call(function () {
            DB::table('recent_users')->delete();
        })->daily();
    }
}
```

除了呼叫閉包這種方式來調度外，你還可以呼叫 可呼叫對象。 可呼叫對像是簡單的 PHP 類，包含一個 `__invoke` 方法：

```
$schedule->call(new DeleteRecentUsers)->daily();
```

如果你想查看任務計畫的概述及其下次計畫執行階段間，你可以使用 `schedule:list` Artisan 命令：

```
php artisan schedule:list
```

34.2.1 Artisan 命令調度

調度方式不僅有呼叫閉包，還有呼叫 Artisan commands 和作業系統命令。例如，你可以給 command 方法傳遞命令名稱或類來調度一個 `Artisan` 命令：

當使用命令類名調度 `Artisan` 命令時，你可以通過一個陣列傳遞附加的命令列參數，且這些參數需要在命令觸發時提供：

```
use App\Console\Commands\SendEmailsCommand;
```

```
$schedule->command('emails:send Taylor --force')->daily();

$schedule->command(SendEmailsCommand::class, ['Taylor', '--force'])->daily();
```

34.2.2 佇列任務調度

`job` 方法可以用來調度 queued job。此方法提供了一種快捷方式來調度任務，而無需使用 `call` 方法建立閉包來調度任務：

```
use App\Jobs\Heartbeat;

$schedule->job(new Heartbeat)->everyFiveMinutes();
```

`job` 方法提供了可選的第二，三參數，分別指定任務將被放置的佇列名稱及連接：

```
use App\Jobs\Heartbeat;

// 分發任務到「heartbeats」佇列及「sqs」連接...
$schedule->job(new Heartbeat, 'heartbeats', 'sqs')->everyFiveMinutes();
```

34.2.3 Shell 命令調度

`exec` 方法可傳送命令到作業系統：

```
$schedule->exec('node /home/forge/script.js')->daily();
```

34.2.4 調度頻率選項

我們已經看到了幾個如何設定任務在指定時間間隔運行的例子。不僅如此，你還有更多的任務調度頻率可選：

方法	描述
`->cron('* * * * *');`	自訂 Cron 計畫執行任務
`->everySecond();`	每秒鐘執行一次任務
`->everyTwoSeconds();`	每 2 秒鐘執行一次任務
`->everyFiveSeconds();`	每 5 秒鐘執行一次任務
`->everyTenSeconds();`	每 10 秒鐘執行一次任務
`->everyFifteenSeconds();`	每 15 秒鐘執行一次任務
`->everyTwentySeconds();`	每 20 秒鐘執行一次任務
`->everyThirtySeconds();`	每 30 秒鐘執行一次任務
`->everyMinute();`	每分鐘執行一次任務
`->everyTwoMinutes();`	每兩分鐘執行一次任務
`->everyThreeMinutes();`	每三分鐘執行一次任務
`->everyFourMinutes();`	每四分鐘執行一次任務
`->everyFiveMinutes();`	每五分鐘執行一次任務
`->everyTenMinutes();`	每十分鐘執行一次任務
`->everyFifteenMinutes();`	每十五分鐘執行一次任務
`->everyThirtyMinutes();`	每三十分鐘執行一次任務
`->hourly();`	每小時執行一次任務
`->hourlyAt(17);`	每小時第十七分鐘時執行一次任務
`->everyTwoHours();`	每兩小時執行一次任務
`->everyThreeHours();`	每三小時執行一次任務
`->everyFourHours();`	每四小時執行一次任務
`->everySixHours();`	每六小時執行一次任務
`->daily();`	每天 00:00 執行一次任務
`->dailyAt('13:00');`	每天 13:00 執行一次任務

方法	描述
`->twiceDaily(1, 13);`	每天 01:00 和 13:00 各執行一次任務
`->twiceDailyAt(1, 13, 15);`	每天 1:15 和 13:15 各執行一次任務
`->weekly();`	每週日 00:00 執行一次任務
`->weeklyOn(1, '8:00');`	每週一 08:00 執行一次任務
`->monthly();`	每月第一天 00:00 執行一次任務
`->monthlyOn(4, '15:00');`	每月第四天 15:00 執行一次任務
`->twiceMonthly(1, 16, '13:00');`	每月第一天和第十六天的 13:00 各執行一次任務
`->lastDayOfMonth('15:00');`	每月最後一天 15:00 執行一次任務
`->quarterly();`	每季度第一天 00:00 執行一次任務
`->quarterlyOn(4, '14:00');`	每季度第四天 14:00 運行一次任務
`->yearly();`	每年第一天 00:00 執行一次任務
`->yearlyOn(6, 1, '17:00');`	每年六月第一天 17:00 執行一次任務
`->timezone('America/New_York');`	為任務設定時區

這些方法與額外的約束條件相結合後，可用於建立在一週的特定時間運行甚至更精細的工作排程。例如，在每週一執行命令：

```
// 在每週一 13:00 執行...
$schedule->call(function () {
    // ...
})->weekly()->mondays()->at('13:00');

// 在每個工作日 8:00 到 17:00 之間的每小時週期執行...
$schedule->command('foo')
          ->weekdays()
          ->hourly()
          ->timezone('America/Chicago')
          ->between('8:00', '17:00');
```

下方列出了額外的約束條件：

方法	描述
`->weekdays();`	工作日執行
`->weekends();`	週末執行
`->sundays();`	週日執行
`->mondays();`	週一執行
`->tuesdays();`	週二執行
`->wednesdays();`	週三執行
`->thursdays();`	週四執行
`->fridays();`	週五執行
`->saturdays();`	週六執行
`->days(array|mixed);`	每週的指定日期執行
`->between($startTime, $endTime);`	`$startTime` 和 `$endTime` 區間執行
`->unlessBetween($startTime, $endTime);`	不在 `$startTime` 和 `$endTime` 區間執行
`->when(Closure);`	閉包返回為真時執行
`->environments($env);`	特定環境中執行

34.2.4.1 周幾（Day）限制

`days` 方法可以用於限制任務在每週的指定日期執行。舉個例子，你可以在讓一個命令每週日和每週三每小時執行一次：

```
$schedule->command('emails:send')
                 ->hourly()
                 ->days([0, 3]);
```

不僅如此，你還可以使用 `Illuminate\Console\Scheduling\Schedule` 類中的常數來設定任務在指定

日期運行：

```
use Illuminate\Console\Scheduling\Schedule;

$schedule->command('emails:send')
                 ->hourly()
                 ->days([Schedule::SUNDAY, Schedule::WEDNESDAY]);
```

34.2.4.2 時間範圍限制

between 方法可用於限制任務在一天中的某個時間段執行：

```
$schedule->command('emails:send')
                     ->hourly()
                     ->between('7:00', '22:00');
```

同樣， unlessBetween 方法也可用於限制任務不在一天中的某個時間段執行：

```
$schedule->command('emails:send')
                     ->hourly()
                     ->unlessBetween('23:00', '4:00');
```

34.2.4.3 真值檢測限制

when 方法可根據閉包返回結果來執行任務。換言之，若給定的閉包返回 true，若無其他限制條件阻止，任務就會一直執行：

```
$schedule->command('emails:send')->daily()->when(function () {
    return true;
});
```

skip 可看作是 when 的逆方法。若 skip 方法返回 true，任務將不會執行：

```
$schedule->command('emails:send')->daily()->skip(function () {
    return true;
});
```

當鏈式呼叫 when 方法時，僅當所有 when 都返回 true 時，任務才會執行。

34.2.4.4 環境限制

environments 方法可限制任務在指定環境中執行（由 APP_ENV 環境變數 定義）：

```
$schedule->command('emails:send')
             ->daily()
             ->environments(['staging', 'production']);
```

34.2.5 時區

timezone 方法可指定在某一時區的時間執行工作排程：

```
$schedule->command('report:generate')
          ->timezone('America/New_York')
          ->at('2:00')
```

若想給所有工作排程分配相同的時區，那麼需要在 app/Console/Kernel.php 類中定義 scheduleTimezone 方法。該方法會返回一個默認時區，最終分配給所有工作排程：

```
use DateTimeZone;

/**
 * 獲取計畫事件默認使用的時區
 */
protected function scheduleTimezone(): DateTimeZone|string|null
{
```

```
    return 'America/Chicago';
}
```

> **注意** 請記住，有些時區會使用夏令時。當夏令時發生調整時，你的任務可能會執行兩次，甚至根本不會執行。因此，我們建議儘可能避免使用時區來安排工作排程。

34.2.6 避免任務重複

默認情況下，即使之前的任務實例還在執行，調度內的任務也會執行。為避免這種情況的發生，你可以使用 `withoutOverlapping` 方法：

```
$schedule->command('emails:send')->withoutOverlapping();
```

在此例中，若 `emails:send` Artisan 命令 還未運行，那它將會每分鐘執行一次。如果你的任務執行時間非常不確定，導致你無法精準預測任務的執行時間，那 `withoutOverlapping` 方法會特別有用。

如有需要，你可以在 `withoutOverlapping` 鎖過期之前，指定它的過期分鐘數。默認情況下，這個鎖會在 24 小時後過期：

```
$schedule->command('emails:send')->withoutOverlapping(10);
```

上面這種場景中，`withoutOverlapping` 方法使用應用程式的 快取 獲取鎖。如有必要，可以使用 `schedule:clear cache` Artisan 命令清除這些快取鎖。這通常只有在任務由於意外的伺服器問題而卡住時才需要。

34.2.7 任務只運行在一台伺服器上

> **注意** 要使用此功能，你的應用程式必須使用 `database`, `memcached`, `dynamodb`, 或 `redis` 快取驅動程式作為應用程式的默認快取驅動程式。此外，所有伺服器必須和同一個中央快取伺服器通訊。

如果你的應用運行在多台伺服器上，可能需要限制調度任務只在某台伺服器上運行。 例如，假設你有一個每個星期五晚上生成新報告的調度任務，如果任務調度器運行在三台伺服器上，調度任務會在三台伺服器上運行並且生成三次報告，不夠優雅！

要指示任務應僅在一台伺服器上運行，請在定義工作排程時使用 `onOneServer` 方法。第一台獲取到該任務的伺服器會給任務上一把原子鎖以阻止其他伺服器同時運行該任務:

```
$schedule->command('report:generate')
                 ->fridays()
                 ->at('17:00')
                 ->onOneServer();
```

34.2.7.1 命名單伺服器作業

有時，你可能需要使用不同的參數調度相同的作業，同時使其仍然在單個伺服器上運行作業。為此，你可以使用 `name` 方法為每個作業定義一個唯一的名字：

```
$schedule->job(new CheckUptime('https://laravel.com'))
             ->name('check_uptime:laravel.com')
             ->everyFiveMinutes()
             ->onOneServer();

$schedule->job(new CheckUptime('https://vapor.laravel.com'))
             ->name('check_uptime:vapor.laravel.com')
             ->everyFiveMinutes()
             ->onOneServer();
```

如果你使用閉包來定義單伺服器作業，則必須為他們定義一個名字

```
$schedule->call(fn () => User::resetApiRequestCount())
     ->name('reset-api-request-count')
     ->daily()
     ->onOneServer();
```

34.2.8 後台任務

默認情況下，同時運行多個任務將根據它們在 `schedule` 方法中定義的順序執行。如果你有一些長時間運行的任務，將會導致後續任務比預期時間更晚啟動。 如果你想在背景執行任務，以便它們可以同時運行，則可以使用 `runInBackground` 方法:

```
$schedule->command('analytics:report')
          ->daily()
          ->runInBackground();
```

注意 `runInBackground` 方法只有在通過 `command` 和 `exec` 方法調度任務時才可以使用

34.2.9 維護模式

當應用處於 維護模式 時，Laravel 的佇列任務將不會運行。因為我們不想調度任務干擾到伺服器上可能還未完成的維護項目。不過，如果你想強制任務在維護模式下運行，你可以使用 `evenInMaintenanceMode` 方法：

```
$schedule->command('emails:send')->evenInMaintenanceMode();
```

34.3 運行調度程序

現在，我們已經學會了如何定義工作排程，接下來讓我們討論如何真正在伺服器上運行它們。`schedule:run` Artisan 命令將評估你的所有工作排程，並根據伺服器的當前時間決定它們是否運行。

因此，當使用 Laravel 的調度程序時，我們只需要向伺服器新增一個 cron 組態項，該項每分鐘運行一次 `schedule:run` 命令。如果你不知道如何向伺服器新增 cron 組態項，請考慮使用 Laravel Forge 之類的服務來為你管理 cron 組態項：

```
* * * * * cd /path-to-your-project && php artisan schedule:run >> /dev/null 2>&1
```

34.3.1 本地運行調度程序

通常，你不會直接將 cron 組態項新增到本地開發電腦。你反而可以使用 `schedule:work` Artisan 命令。該命令將在前景執行，並每分鐘呼叫一次調度程序，直到你終止該命令為止：

```
php artisan schedule:work
```

34.4 任務輸出

Laravel 調度器提供了一些簡便方法來處理調度任務生成的輸出。首先，你可以使用 `sendOutputTo` 方法將輸出傳送到檔案中以便後續檢查：

```
$schedule->command('emails:send')
          ->daily()
          ->sendOutputTo($filePath);
```

如果希望將輸出追加到指定檔案，可使用 `appendOutputTo` 方法：

```
$schedule->command('emails:send')
          ->daily()
          ->appendOutputTo($filePath);
```

使用 `emailOutputTo` 方法，你可以將輸出傳送到指定信箱。在傳送郵件之前，你需要先組態 Laravel 的 郵件服務:

```
$schedule->command('report:generate')
          ->daily()
          ->sendOutputTo($filePath)
          ->emailOutputTo('taylor@example.com');
```

如果你只想在命令執行失敗時將輸出傳送到信箱，可使用 `emailOutputOnFailure` 方法：

```
$schedule->command('report:generate')
          ->daily()
          ->emailOutputOnFailure('taylor@example.com');
```

> **注意** `emailOutputTo`, `emailOutputOnFailure`, `sendOutputTo` 和 `appendOutputTo` 是 `command` 和 `exec` 獨有的方法。

34.5 任務鉤子

使用 `before` 和 `after` 方法，你可以決定在調度任務執行前或者執行後來運行程式碼：

```
$schedule->command('emails:send')
          ->daily()
          ->before(function () {
              // 任務即將執行。。。
          })
          ->after(function () {
              // 任務已經執行。。。
          });
```

使用 `onSuccess` 和 `onFailure` 方法，你可以決定在調度任務成功或者失敗運行程式碼。失敗表示 Artisan 或系統命令以非零退出碼終止：

```
$schedule->command('emails:send')
          ->daily()
          ->onSuccess(function () {
              // 任務執行成功。。。
          })
          ->onFailure(function () {
              // 任務執行失敗。。。
          });
```

如果你的命令有輸出，你可以使用 `after`, `onSuccess` 或 `onFailure` 鉤子並傳入類型為 `Illuminate\Support\Stringable` 的`$output` 參數的閉包來訪問任務輸出：

```
use Illuminate\Support\Stringable;

$schedule->command('emails:send')
          ->daily()
          ->onSuccess(function (Stringable $output) {
              // The task succeeded...
          })
          ->onFailure(function (Stringable $output) {
              // The task failed...
          });
```

34.5.1.1 Pinging 網址

使用 `pingBefore` 和 `thenPing` 方法，你可以在任務完成之前或完成之後來 ping 指定的 URL。當前方法在通知外部服務，如 Envoyer，工作排程在將要執行或已完成時會很有用：

```
$schedule->command('emails:send')
          ->daily()
```

```
            ->pingBefore($url)
            ->thenPing($url);
```

只有當條件為 `true` 時，才可以使用 `pingBeforeIf` 和 `thenPingIf` 方法來 ping 指定 URL ：

```
$schedule->command('emails:send')
            ->daily()
            ->pingBeforeIf($condition, $url)
            ->thenPingIf($condition, $url);
```

當任務成功或失敗時，可使用 `pingOnSuccess` 和 `pingOnFailure` 方法來 ping 給定 URL。失敗表示 Artisan 或系統命令以非零退出碼終止：

```
$schedule->command('emails:send')
            ->daily()
            ->pingOnSuccess($successUrl)
            ->pingOnFailure($failureUrl);
```

所有 ping 方法都依賴 Guzzle HTTP 庫。通常，Guzzle 已在所有新的 Laravel 項目中默認安裝，不過，若意外將 Guzzle 刪除，則可以使用 Composer 包管理器將 Guzzle 手動安裝到項目中：

```
composer require guzzlehttp/guzzle
```

34.6 事件

如果需要，你可以監聽調度程序調度的 事件。通常，事件偵聽器對應將在你的應用程式的 `App\Providers\EventServiceProvider` 類中定義：

```
/**
 * 應用的事件監聽器對應
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Console\Events\ScheduledTaskStarting' => [
        'App\Listeners\LogScheduledTaskStarting',
    ],

    'Illuminate\Console\Events\ScheduledTaskFinished' => [
        'App\Listeners\LogScheduledTaskFinished',
    ],

    'Illuminate\Console\Events\ScheduledBackgroundTaskFinished' => [
        'App\Listeners\LogScheduledBackgroundTaskFinished',
    ],

    'Illuminate\Console\Events\ScheduledTaskSkipped' => [
        'App\Listeners\LogScheduledTaskSkipped',
    ],

    'Illuminate\Console\Events\ScheduledTaskFailed' => [
        'App\Listeners\LogScheduledTaskFailed',
    ],
];
```