Laravel 10
官方文件

目錄

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樁，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',
    ],
];
```

35 使用者認證

35.1 簡介

許多 Web 應用程式為其使用者提供了一種通過應用程式進行身份驗證和「登錄」的方法。在 Web 應用程式中實現此功能可能是一項複雜且具有潛在風險的工作。因此，Laravel 致力於為你提供所需的工具，以快速、安全、輕鬆地實現身份驗證。

Laravel 的身份驗證工具的核心是由「看守器」和「提供器」組成的。 看守器定義如何對每個請求的使用者進行身份驗證。例如，Laravel 附帶了一個 `session` 守衛，它使用 session 和 cookie 來維護狀態。

提供器定義如何從持久儲存中檢索使用者。 Laravel 支援使用 Eloquent 和資料庫查詢建構器檢索使用者。不僅如此，你甚至可以根據應用程式的需要自由定製其他提供程序。

應用程式的身份驗證組態檔案位於 `config/auth.php`. 這個檔案包含幾個記載了的選項，用於調整 Laravel 身份驗證服務的行為。

> **注意**
> 看守器和提供器不應與「角色」和「權限」混淆。要瞭解有關通過權限授權使用者操作的更多資訊，請參閱 使用者授權 文件。

35.1.1 入門套件

想要快速入門？在新的 Laravel 應用程式中安裝 Laravel 入門套件。遷移資料庫後，將瀏覽器導航到 `/register` 或分配給應用程式的任何其他 URL。這個入門套件將負責建構你的整個身份驗證系統！

即使你在最終的 Laravel 應用程式中選擇不使用入門套件，安裝 Laravel Breeze 入門套件也是學習如何在實際的 Laravel 項目中實現所有 Laravel 身份驗證功能的絕佳機會。 由於 Laravel Breeze 為你建立身份驗證 controller 、路由和檢視，因此你可以查看這些檔案中的原始碼，進而瞭解如何實現 Laravel 的身份驗證功能。

35.1.2 資料庫注意事項

默認情況下，Laravel 在 `app/Models` 目錄中包含一個 `App\Models\User` Eloquent 模型。 此模型可與默認的 Eloquent 身份驗證驅動程式一起使用。如果你的應用程式未使用 Eloquent，則可以使用 Laravel 查詢建構器的 `database` 身份驗證提供程序。

為 `App\Models\User` 模型建構資料庫架構時，請確保密碼列的長度至少為 60 個字元。當然，新的 Laravel 應用程式中包含的 `users` 表遷移檔案已經建立了一個超過此長度的列。

此外，你應該驗證你的 `users` (或等效) 表是否包含一個可為空的字串 `remember_token` 列，該列包含 100 個字元。 此列將用於為在登錄到應用程式時選擇「記住我」選項的使用者儲存令牌。同樣，新的 Laravel 應用程式中包含的默認 `users` 表遷移檔案已經包含此列。

35.1.3 生態系統概述

Laravel 提供了幾個與身份驗證相關的包。在繼續之前，我們將回顧 Laravel 中的通用身份驗證生態系統，並討論每個包的預期用途。

首先，考慮身份驗證是如何工作的。使用 web 瀏覽器時，使用者將通過登錄表單提供他們的使用者名稱和密碼。如果這些憑據正確，應用程式將在使用者的 session 中儲存有關已通過身份驗證的使用者的資訊。發給瀏覽器的 cookie 包含 session ID，以便應用程式的後續請求可以將使用者與正確的 session 相關聯。在接收到 session 的 cookie 之後，應用程式將基於 session ID 檢索 session 資料，注意認證資訊已經儲存在 session 中，並且將使用者視為「已認證」。

當遠端服務需要通過身份驗證才能訪問 API 時，我們通常不用 cookie 進行身份驗證，因為沒有 web 瀏覽器。相反，遠端服務會在每個請求時向 API 傳送一個 token。應用程式可以對照有效 API 令牌表來驗證傳入 token ，並「驗證」與該 API 令牌相關聯的使用者正在執行的請求。

35.1.3.1 Laravel 內建的瀏覽器認證服務

Laravel 包括內建的身份驗證和 session 服務，這些服務通常通過 `Auth` 和 `Session` facade 使用。 這些特性為從 web 瀏覽器發起的請求提供基於 cookie 的身份驗證。它們提供的方法允許你驗證使用者的憑據並對使用者進行身份驗證。此外，這些服務會自動將正確的身份驗證資料儲存在使用者的 session 中，並行布使用者的 session cookie 。本文件中包含對如何使用這些服務的討論。

應用入門套件

如本文件中所述，你可以手動與這些身份驗證服務進行互動，以建構應用程式自己的身份驗證層。不過，為了幫助你更快地入門，我們發佈了 免費軟體包，為整個身份驗證層提供強大的現代化腳手架。這些軟體包分別是 Laravel Breeze，Laravel Jetstream，和 Laravel Fortify。

Laravel Breeze 是 Laravel 所有身份驗證功能的簡單、最小實現，包括登錄、註冊、密碼重設、電子郵件驗證和密碼確認。 Laravel Breeze 的檢視層由簡單的 Blade 範本 組成，樣式為 Tailwind CSS。要開始使用，請查看 Laravel 的 應用入門套件 文件。

Laravel Fortify 是 Laravel 的無 header 身份驗證後端，它實現了本文件中的許多功能，包括基於 cookie 的身份驗證以及其他功能，如雙因素身份驗證和電子郵件驗證。Fortify 為 Laravel Jetstream 提供身份驗證後端，或者可以單獨與 Laravel Sanctum 結合使用，為需要使用 Laravel 進行身份驗證的 SPA 提供身份驗證。

Laravel Jetstream 是一個強大的應用入門套件，它使用 Tailwind CSS，Livewire 和 / 或 Inertia 提供美觀的現代 UI，同時整合和擴展了 Laravel Fortify 的認證服務。Laravel Jetstream 提供了雙因素身份驗證、團隊支援、瀏覽器 session 管理、個人資料管理等功能，並內建了 Laravel Sanctum 的整合以支援 API 令牌身份驗證。接下來我們將討論 Laravel 的 API 身份驗證產品。

35.1.3.2 Laravel 的 API 認證服務

Laravel 提供了兩個可選的包來幫助你管理 API 令牌和驗證使用 API 令牌發出的請求：Passport 和 Sanctum。請注意，這些庫和 Laravel 內建的基於 Cookie 的身份驗證庫並不是互斥的。這些庫主要關注 API 令牌身份驗證，而內建的身份驗證服務則關注基於 Cookie 的瀏覽器身份驗證。許多應用程式將同時使用 Laravel 內建的基於 Cookie 的身份驗證服務和一個 Laravel 的 API 身份驗證包。

Passport

Passport 是一個 OAuth2 身份驗證提供程序，提供各種 OAuth2 「授權類型」，允許你發佈各種類型的令牌。總的來說，這是一個強大而複雜的 API 身份驗證包。但是，大多數應用程式不需要 OAuth2 規範提供的複雜特性，這可能會讓使用者和開發人員感到困惑。此外，開發人員一直對如何使用 Passport 等 OAuth2 身份驗證提供程序對 SPA 應用程式或移動應用程式進行身份驗證感到困惑。

Sanctum

為了應對 OAuth2 的複雜性和開發人員的困惑，我們著手建構一個更簡單、更精簡的身份驗證包，旨在處理通過令牌進行的第一方 Web 請求和 API 請求。 Laravel Sanctum 發佈後，這一目標就實現了。對於除 API 外還提供第一方 web UI 的應用程式，或由單頁應用程式（SPA）提供支援的應用程式，或是提供移動客戶端的應用程

式，Sanctum 是首選推薦的身份驗證包。

Laravel Sanctum 是一個混合了 web 和 API 的身份驗證包，它讓我們管理應用程式的整個身份驗證過程成為可能，因為當基於 Sanctum 的應用程式收到請求時，Sanctum 將首先確定請求是否包含引用已驗證 session 的 session cookie。Sanctum 通過呼叫我們前面討論過的 Laravel 的內建身份驗證服務來實現這一點。如果請求沒有通過 session cookie 進行身份驗證，Sanctum 將檢查請求中的 API 令牌。如果存在 API 令牌，則 Sanctum 將使用該令牌對請求進行身份驗證。要瞭解有關此過程的更多資訊，請參閱 Sanctum 的 工作原理 文件。

Laravel Sanctum 是我們選擇與 Laravel Jetstream 應用程式入門套件一起使用的 API 包，因為我們認為它最適合大多數 web 應用程式的身份驗證需求。

35.1.3.3 彙總 & 選擇你的解決方案

總之，如果你的應用程式將使用瀏覽器訪問，並且你正在建構一個單頁面的 Laravel 應用程式，那麼你的應用程式可以使用 Laravel 的內建身份驗證服務。

接下來，如果你的應用程式提供將由第三方使用的 API ，你可以在 Passport 或 Sanctum 之間進行選擇，為你的應用程式提供 API 令牌身份驗證。一般來說，儘可能選擇 Sanctum，因為它是 API 認證、SPA 認證和移動認證的簡單、完整的解決方案，包括對「scopes」或「abilities」的支援。

如果你正在建構一個將由 Laravel 後端支援的單頁面應用程式（SPA），那麼應該使用 Laravel Sanctum。在使用 Sanctum 時，你需要 手動實現自己的後端驗證路由 或使用 Laravel Fortify 作為無 header 身份驗證後端服務，為註冊、密碼重設、電子郵件驗證等功能提供路由和 controller 。

當應用程式確定必須使用 OAuth2 規範提供的所有特性時，可以選擇 Passport。

而且，如果你想快速入門，我們很高興推薦 Laravel Breeze 作為啟動新 Laravel 應用程式的快速方法，該應用程式已經使用了我們首選的 Laravel 內建身份驗證服務和 Laravel Sanctum 身份驗證技術堆疊。

35.2 身份驗證快速入門

> **警告**
> 文件的這一部分討論了通過 Laravel 應用入門套件 對使用者進行身份驗證，其中包括可幫助你快速入門的 UI 腳手架。如果你想直接與 Laravel 的身份驗證系統整合，請查看 手動驗證使用者 上的文件。

35.2.1 安裝入門套件

首先，你應該 安裝 Laravel 應用入門套件。我們當前的入門套件 Laravel Breeze 和 Laravel Jetstream 提供了設計精美的起點，可將身份驗證納入你的全新 Laravel 應用程式。

Laravel Breeze 是 Laravel 所有身份驗證功能的最簡單的實現，包括登錄、註冊、密碼重設、電子郵件驗證和密碼確認。 Laravel Breeze 的檢視層由簡單的 Blade templates 和 Tailwind CSS 組成。Breeze 還使用 Vue 或 React 提供了基於 Inertia 的腳手架選項。

Laravel Jetstream 是一個更強大的應用入門套件，它支援使用 Livewire 或 Inertia and Vue 來建構你的應用程式。 此外，Jetstream 還提供可選的雙因素身份驗證支援、團隊、組態檔案管理、瀏覽器 session 管理、通過 Laravel Sanctum 的 API 支援、帳戶刪除等。

35.2.2 獲取已認證的使用者資訊

在安裝身份驗證入門套件並允許使用者註冊應用程式並對其進行身份驗證之後，你通常需要與當前通過身份驗

證的使用者進行互動。在處理傳入請求時，你可以通過 Auth facade 的 user 方法訪問通過身份驗證的使用者：

```
use Illuminate\Support\Facades\Auth;

// 獲取當前的認證使用者資訊...
$user = Auth::user();

// 獲取當前的認證使用者 ID...
$id = Auth::id();
```

或者，一旦使用者通過身份驗證，你就可以通過 Illuminate\Http\Request 實例訪問通過身份驗證的使用者。請記住，使用類型提示的類將自動注入到 controller 方法中。通過對 Illuminate\Http\Request 對象進行類型提示，你可以通過 Request 的 user 方法從應用程式中的任何 controller 方法方便地訪問通過身份驗證的使用者：

```
<?php

namespace App\Http\Controllers;

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

class FlightController extends Controller
{
    /**
     * 更新現有航班的航班資訊。
     */
    public function update(Request $request): RedirectResponse
    {
        $user = $request->user();

        // ...

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

35.2.2.1 確定當前使用者是否已通過身份驗證

要確定發出傳入 HTTP 請求的使用者是否通過身份驗證，你可以在 Auth facade 上使用 check 方法。如果使用者通過身份驗證，此方法將返回 true ：

```
use Illuminate\Support\Facades\Auth;

if (Auth::check()) {
    // 該使用者已登錄...
}
```

> **注意**
> 儘管可以使用 check 方法確定使用者是否已通過身份驗證，但在允許使用者訪問某些路由 / controller 之前，你通常會使用中介軟體驗證使用者是否已通過身份驗證。要瞭解更多資訊，請查看有關 路由保護 的文件。

35.2.3 路由保護

路由中介軟體 可用於僅允許通過身份驗證的使用者訪問給定路由。Laravel 附帶了一個 auth 中介軟體，它引用了 Illuminate\Auth\Middleware\Authenticate 類。由於此中介軟體已在應用程式的 HTTP 核心中註冊，因此你只需將中介軟體附加到路由定義即可：

```
Route::get('/flights', function () {
    // 只有經過身份驗證的使用者才能訪問此路由...
```

```
})->middleware('auth');
```

35.2.3.1 給未認證的使用者設定重新導向

當 `auth` 中介軟體檢測到未經身份驗證的使用者時，它將使用者重新導向到 `login` 命名路由。你可以通過更新應用程式的 `app/Http/Middleware/Authenticate.php` 檔案中的 `redirectTo` 方法來修改此行為：

```
use Illuminate\Http\Request;

/**
 * 獲取使用者應重新導向到的路徑。
 */
protected function redirectTo(Request $request): string
{
    return route('login');
}
```

35.2.3.2 指定看守器

將 `auth` 中介軟體附加到路由時，你還可以指定應該使用哪個「guard」來驗證使用者。指定的 guard 應與 `auth.php` 組態檔案的 `guards` 陣列中的一個鍵相對應：

```
Route::get('/flights', function () {
    // 只有經過身份驗證的使用者才能訪問此路由...
})->middleware('auth:admin');
```

35.2.4 登錄限流

如果你使用的是 Laravel Breeze 或 Laravel Jetstream 入門套件，那麼在嘗試登錄的時候將自動應用速率限制。默認情況下，如果使用者在多次嘗試後未能提供正確的憑據，他們將在一分鐘內無法登錄。該限制對與使用者的使用者名稱 / 電子郵件地址及其 IP 地址是唯一的。

> **注意**
> 如果你想對應用程式中的其他路由進行速率限制，請查看 速率限制 文件。

35.3 手動驗證使用者

你並非一定要使用 Laravel 的 應用入門套件 附帶的身份驗證腳手架。如果你選擇不使用這個腳手架，則需要直接使用 Laravel 身份驗證類來管理使用者身份驗證。別擔心，這也很容易！

我們將通過 `Auth` facade 訪問 Laravel 的身份驗證服務，因此我們需要確保在類的頂部匯入 `Auth` facade 。接下來，讓我們看看 `attempt` 方法。 `attempt` 方法通常用於處理來自應用程式「登錄」表單的身份驗證嘗試。如果身份驗證成功，你應該重新生成使用者的 session 以防止 session fixation:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Http\RedirectResponse;
use Illuminate\Support\Facades\Auth;

class LoginController extends Controller
{
    /**
     * 處理身份驗證嘗試。
     */
    public function authenticate(Request $request): RedirectResponse
```

```
    {
        $credentials = $request->validate([
            'email' => ['required', 'email'],
            'password' => ['required'],
        ]);

        if (Auth::attempt($credentials)) {
            $request->session()->regenerate();

            return redirect()->intended('dashboard');
        }

        return back()->withErrors([
            'email' => 'The provided credentials do not match our records.',
        ])->onlyInput('email');
    }
}
```

`attempt` 方法接受一個鍵 / 值對陣列作為它的第一個參數。陣列中的值將用於在資料庫表中尋找使用者。因此，在上面的示例中，將通過 `email` 列的值檢索使用者。如果找到使用者，則資料庫中儲存的 hash 密碼將與通過陣列傳遞給該方法的 `password` 值進行比較。你不應該對傳入請求的 `password` 值進行 hash 處理，因為框架會在將該值與資料庫中的 hash 密碼進行比較之前自動對該值進行 hash 處理。如果兩個 hash 密碼匹配，將為使用者啟動一個通過身份驗證的 session。

請記住，Laravel 的身份驗證服務將根據身份驗證 guard 的「provider」組態，從資料庫檢索使用者。在默認的 `config/auth.php` 組態檔案中，指定了 Eloquent 為使用者提供程序，並指示它在檢索使用者時使用 `App\Models\User` 模型。你可以根據應用程式的需要在組態檔案中更改這些值。

如果身份驗證成功，`attempt` 方法將返回 `true` 。否則，將返回 `false`。

Laravel 的重新導向器提供的 `intended` 方法會將使用者重新導向到他們在被身份驗證中介軟體攔截之前嘗試訪問的 URL。如果預期的目的地不可用，可以為該方法提供回退 URI。

35.3.1.1 指定附加條件

如果你願意，除了使用者的電子郵件和密碼之外，你還可以向身份驗證查詢中新增額外的查詢條件。為了實現這一點，我們可以簡單地將查詢條件新增到傳遞給 `attempt` 方法的陣列中。例如，我們可以驗證使用者是否標記為「active」：

```
if (Auth::attempt(['email' => $email, 'password' => $password, 'active' => 1])) {
    // 認證成功...
}
```

對於複雜的查詢條件，你可以在憑證陣列中提供閉包。此閉包將與查詢實例一起呼叫，允許你根據應用程式的需要自訂查詢：

```
use Illuminate\Database\Eloquent\Builder;

if (Auth::attempt([
    'email' => $email,
    'password' => $password,
    fn (Builder $query) => $query->has('activeSubscription'),
])) {
    // 認證成功...
}
```

> **警告**
> 在這些例子中，`email` 不是必需的選項，它只是作為一個例子。你應該使用與資料庫表中的「使用者名稱」對應的任何列名。

`attemptWhen` 方法接收一個閉包作為其第二個參數，可用於在實際驗證使用者之前對潛在使用者執行更廣泛

的檢查。閉包接收潛在使用者並應返回 `true` 或 `false` 以指示使用者是否可以通過身份驗證：

```
if (Auth::attemptWhen([
     'email' => $email,
     'password' => $password,
], function (User $user) {
     return $user->isNotBanned();
})) {
     // 認證成功...
}
```

35.3.1.2 訪問特定的看守器實例

通過 `Auth` facade 的 `guard` 方法，你可以指定在對使用者進行身份驗證時要使用哪個 guard 實例。這允許你使用完全不同的可驗證模型或使用者表來管理應用程式的不同部分的驗證。

傳遞給 `guard` 方法的 guard 名稱應該對應於 `auth.php` 組態檔案中 guards 的其中一個：

```
if (Auth::guard('admin')->attempt($credentials)) {
     // ...
}
```

35.3.2 記住使用者

許多 web 應用程式在其登錄表單上提供了「記住我」複選框。如果你希望在應用程式中提供「記住我」功能，你可以將布林值作為第二個參數傳遞給 `attempt` 方法。

當此值為 `true` 時，Laravel 將無限期地保持使用者身份驗證，或者直到使用者手動註銷。 你的 `users` 表必須包含字串 `remember_token` 列，該列將用於儲存「記住我」標記。新的 Laravel 應用程式中包含的 `users` 表遷移檔案已經包含此列：

```
use Illuminate\Support\Facades\Auth;

if (Auth::attempt(['email' => $email, 'password' => $password], $remember)) {
     // 正在為該使用者執行記住我操作...
}
```

如果你的應用程式提供「記住我」的功能，你可以使用 `viaRemember` 方法來確定當前通過身份驗證的使用者是否使用「記住我」cookie 進行了身份驗證：

```
use Illuminate\Support\Facades\Auth;

if (Auth::viaRemember()) {
     // ...
}
```

35.3.3 其他身份驗證方法

35.3.3.1 驗證使用者實例

如果你需要將現有使用者實例設定為當前通過身份驗證的使用者，你可以將該使用者實例傳遞給 `Auth` facade 的 `login` 方法。 給定的使用者實例必須是 `Illuminate\Contracts\Auth\Authenticatable` 契約 的實現。 Laravel 中包含的 `App\Models\User` 模型已經實現了此介面。當你已經有一個有效的使用者實例時（例如使用者直接向你的應用程式註冊之後），此身份驗證方法非常有用：

```
use Illuminate\Support\Facades\Auth;

Auth::login($user);
```

你可以將布林值作為第二個參數傳遞給 `login` 方法。此值指示通過身份驗證的 session 是否需要「記住我」功

能。請記住，這意味著 session 將無限期地進行身份驗證，或者直到使用者手動註銷應用程式為止：

```
Auth::login($user, $remember = true);
```

如果需要，你可以在呼叫 `login` 方法之前指定身份驗證看守器：

```
Auth::guard('admin')->login($user);
```

35.3.3.2 通過 ID 對使用者進行身份驗證

要使用資料庫記錄的主鍵對使用者進行身份驗證，你可以使用 `loginUsingId` 方法。此方法接受你要驗證的使用者的主鍵：

```
Auth::loginUsingId(1);
```

你可以將布林值作為第二個參數傳遞給 `loginUsingId` 方法。此值指示通過身份驗證的 session 是否需要「記住我」功能。請記住，這意味著 session 將無限期地進行身份驗證，或者直到使用者手動註銷應用程式為止：

```
Auth::loginUsingId(1, $remember = true);
```

35.3.3.3 只驗證一次

你可以使用 `once` 方法通過應用程式對單個請求的使用者進行身份驗證。呼叫此方法時不會使用 session 或 cookie：

```
if (Auth::once($credentials)) {
    // ...
}
```

35.4 HTTP Basic 使用者認證

HTTP Basic 使用者認證 提供了一種無需設定專用「登錄」頁面即可對應用程式使用者進行身份驗證的快速方法。首先，將 `auth.basic` 中介軟體 附加到路由。 `auth.basic` 中介軟體包含在 Laravel 框架中，因此你不需要定義它：

```
Route::get('/profile', function () {
    // 只有經過身份驗證的使用者才能訪問此路由...
})->middleware('auth.basic');
```

將中介軟體附加到路由後，當你在瀏覽器中訪問路由時，系統會自動提示你輸入憑據。默認情況下 `auth.basic` 中介軟體將假定 `users` 資料庫表中的 `email` 列是使用者的「使用者名稱」。

35.4.1.1 注意 FastCGI

如果你使用的是 PHP FastCGI 和 Apache 來為 Laravel 應用程式提供服務，那麼 HTTP Basic 身份驗證可能無法正常工作。要糾正這些問題，可以將以下行新增到應用程式的 `.htaccess` 檔案中：

```
RewriteCond %{HTTP:Authorization} ^(.+)$
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
```

35.4.2 無狀態 HTTP Basic 認證

你也可以在 session 中不設定使用者識別碼 cookie 的情況下使用 HTTP Basic 身份驗證。如果你選擇使用 HTTP 身份驗證來驗證對應用程式 API 的請求，這將非常有用。為此，定義一個中介軟體 呼叫 `onceBasic` 方法。如果 `onceBasic` 方法沒有返迴響應，則請求可能會進一步傳遞到應用程式中：

```
<?php

namespace App\Http\Middleware;
```

```
use Closure;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;
use Symfony\Component\HttpFoundation\Response;

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

}
```

然後，將中介軟體附加到路由中：

```
Route::get('/api/user', function () {
    // 只有經過身份驗證的使用者才能訪問此路由...
})->middleware(AuthenticateOnceWithBasicAuth::class);
```

35.5 退出登錄

要在應用程式中手動註銷使用者，可以使用 `Auth` facade 提供的 `logout` 方法。這將從使用者的 session 中刪除身份驗證資訊，以便後續請求不會得到身份驗證。

除了呼叫 `logout` 方法外，建議你將使用者的 session 置為過期，並重新生成其 CSRF token。註銷使用者後，通常會將使用者重新導向到應用程式的根目錄：

```
use Illuminate\Http\Request;
use Illuminate\Http\RedirectResponse;
use Illuminate\Support\Facades\Auth;

/**
 * 將使用者退出應用程式。
 */
public function logout(Request $request): RedirectResponse
{
    Auth::logout();

    $request->session()->invalidate();

    $request->session()->regenerateToken();

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

35.5.1 使其他裝置上的 session 失效

Laravel 還提供了這樣一種機制，可以使在其他裝置上處於活動狀態的使用者 session 無效和「註銷」，而不會使其當前裝置上的 session 失效。當使用者正在更改或更新其密碼，並且你希望在保持當前裝置身份驗證的同時使其他裝置上的 session 無效時，通常會使用此功能。

在開始之前，你應該確保 `Illuminate\Session\Middleware\AuthenticateSession` 中介軟體已經包含在應該接收 session 身份驗證的路由中。通常，你應該將此中介軟體放置在一個路由組定義中，以便它可以應用於大多數應用程式的路由。默認情況下， `AuthenticateSession` 中介軟體可以使用 `auth.session` 路由中介軟體別名，並附加到一個路由上，這個別名在你的應用程式的 HTTP 核心中定義：

```
Route::middleware(['auth', 'auth.session'])->group(function () {
    Route::get('/', function () {
        // ...
    });
});
```

然後，你可以使用 `Auth` facade 提供的 `logoutOtherDevices` 方法。此方法要求使用者確認其當前密碼，你的應用程式應通過輸入表單接受該密碼：

```
use Illuminate\Support\Facades\Auth;

Auth::logoutOtherDevices($currentPassword);
```

當呼叫 `logoutOtherDevices` 方法時，使用者的其他 session 將完全失效，這意味著他們將從之前驗證過的所有看守器中「註銷」。

35.6 密碼確認

在建構應用程式時，你可能偶爾會要求使用者在執行操作之前或在將使用者重新導向到應用程式的敏感區域之前確認其密碼。Laravel 包含內建的中介軟體，使這個過程變得輕而易舉。實現此功能你需要定義兩個路由：一個路由顯示請求使用者確認其密碼的檢視，另一個路由確認密碼有效並將使用者重新導向到其預期目的地。

> **注意**
> 以下文件討論了如何直接與 Laravel 的密碼確認功能整合。然而，如果你想更快地開始使用， Laravel 應用入門套件 包括對此功能的支援！

35.6.1 組態

確認密碼後，使用者在三個小時內不會被要求再次確認密碼。但是，你可以通過更改應用程式 `config/auth.php` 組態檔案中的 `password_timeout` 組態值來組態重新提示使用者輸入密碼之前的時長。

35.6.2 路由

35.6.2.1 密碼確認表單

首先，我們將定義一個路由以顯示請求使用者確認其密碼的檢視：

```
Route::get('/confirm-password', function () {
    return view('auth.confirm-password');
})->middleware('auth')->name('password.confirm');
```

如你所料，此路由返回的檢視應該有一個包含 `password` 欄位的表單。此外，可以隨意在檢視中包含說明使用者正在進入應用程式的受保護區域並且必須確認其密碼的文字。

35.6.2.2 確認密碼

接下來，我們將定義一個路由來處理來自「確認密碼」檢視的表單請求。此路由將負責驗證密碼並將使用者重新導向到其預期目的地：

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Facades\Redirect;

Route::post('/confirm-password', function (Request $request) {
```

```
    if (! Hash::check($request->password, $request->user()->password)) {
        return back()->withErrors([
            'password' => ['The provided password does not match our records.']
        ]);
    }

    $request->session()->passwordConfirmed();

    return redirect()->intended();
})->middleware(['auth', 'throttle:6,1']);
```

在繼續之前，讓我們更詳細地檢查一下這條路由。首先，請求的 `password` 欄位被確定為實際匹配經過身份驗證的使用者的密碼。如果密碼有效，我們需要通知 Laravel 的 session 使用者已經確認了他們的密碼。`passwordConfirmed` 方法將在使用者的 session 中設定一個時間戳，Laravel 可以使用它來確定使用者上次確認密碼的時間。最後，我們可以將使用者重新導向到他們想要的目的地。

35.6.3 保護路由

你應該確保為執行需要最近確認密碼的操作的路由被分配到 `password.confirm` 中介軟體。此中介軟體包含在 Laravel 的默認安裝中，並且會自動將使用者的預期目的地儲存在 session 中，以便使用者在確認密碼後可以重新導向到該位置。在 session 中儲存使用者的預期目的地之後，中介軟體將使用者重新導向到 `password.confirm` 的 命名路由：

```
Route::get('/settings', function () {
    // ...
})->middleware(['password.confirm']);

Route::post('/settings', function () {
    // ...
})->middleware(['password.confirm']);
```

35.7 新增自訂的看守器

你可以使用 `Auth` facade 上的 `extend` 方法定義你自己的身份驗證看守器。你應該在 服務提供者 中呼叫 `extend` 方法。 由於 Laravel 已經附帶了 `AuthServiceProvider`，因此我們可以將程式碼放置在該提供者中：

```
<?php

namespace App\Providers;

use App\Services\Auth\JwtGuard;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;
use Illuminate\Support\Facades\Auth;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * 註冊任意應用程式驗證 / 授權服務。
     */
    public function boot(): void
    {
        Auth::extend('jwt', function (Application $app, string $name, array $config) {
            // 返回 Illuminate\Contracts\Auth\Guard 的實例...

            return new JwtGuard(Auth::createUserProvider($config['provider']));
        });
    }
}
```

正如你在上面的示例中所看到的，傳遞給 extend 方法的回呼應該返回 Illuminate\Contracts\Auth\Guard 的實例。此介面包含一些方法，你需要實現這些方法來定義自訂看守器。定義自訂看守器後，你可以在 auth.php 組態檔案的 guards 組態中引用該看守器：

```
'guards' => [
    'api' => [
        'driver' => 'jwt',
        'provider' => 'users',
    ],
],
```

35.7.1 閉包請求看守器

實現基於 HTTP 請求的自訂身份驗證系統的最簡單方法是使用 Auth::viaRequest 方法。此方法允許你使用單個閉包快速定義身份驗證過程。

首先，請在 AuthServiceProvider 的 boot 方法中呼叫 Auth::viaRequest 方法。 VIASRequest 方法接受身份驗證驅動程式名稱作為其第一個參數。此名稱可以是描述你的自訂看守器的任何字串。傳遞給方法的第二個參數應該是一個閉包，該閉包接收傳入的 HTTP 請求並返回使用者實例，或者，如果身份驗證失敗返回 null:

```
use App\Models\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

/**
 * 註冊任意應用程式驗證 / 授權服務。
 */
public function boot(): void
{
    Auth::viaRequest('custom-token', function (Request $request) {
        return User::where('token', $request->token)->first();
    });
}
```

定義自訂身份驗證驅動程式後，你可以將其組態為 auth.php 組態檔案的 guards 組態中的驅動程式：

```
'guards' => [
    'api' => [
        'driver' => 'custom-token',
    ],
],
```

最後，你可以在將身份驗證中介軟體分配給路由時引用該看守器：

```
Route::middleware('auth:api')->group(function () {
    // ...
}
```

35.8 新增自訂的使用者提供器

如果你不使用傳統的關係型資料庫來儲存使用者，你將需要使用你自己的身份驗證使用者提供器來擴展 Laravel。 我們將在 Auth facade 上的 provider 方法來定義自訂使用者提供器。使用者提供器解析器應返回 Illuminate\Contracts\Auth\UserProvider 的實例：

```
<?php

namespace App\Providers;

use App\Extensions\MongoUserProvider;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;
use Illuminate\Support\Facades\Auth;
```

```
class AuthServiceProvider extends ServiceProvider
{
     /**
      * 註冊任意應用程式驗證 / 授權服務。
      */
     public function boot(): void
     {
          Auth::provider('mongo', function (Application $app, array $config) {
               // 返回 illighte\Contracts\Auth\UserProvider 的實例...

               return new MongoUserProvider($app->make('mongo.connection'));
          });
     }
}
```

使用 `provider` 方法註冊提供器後，你可以在 `auth.php` 組態檔案中切換到新的使用者提供器。 首先，定義一個使用新驅動程式的 `provider`：

```
'providers' => [
     'users' => [
          'driver' => 'mongo',
     ],
],
```

最後，你可以在 `guards` 組態中引用此提供器：

```
'guards' => [
     'web' => [
          'driver' => 'session',
          'provider' => 'users',
     ],
],
```

35.8.1 使用者提供器契約

`Illuminate\Contracts\Auth\UserProvider` 實現負責從持久性儲存系統（如 MySQL、MongoDB 等）中獲取 `Illuminate\Contracts\Auth\Authenticatable` 實現。這兩個介面可以保障 Laravel 身份驗證機制持續工作，無論使用者資料是如何儲存的，或者可以使用任意類型的類來表示經過身份驗證的使用者：

讓我們看一下 `Illuminate\Contracts\Auth\UserProvider` 契約：

```
<?php

namespace Illuminate\Contracts\Auth;

interface UserProvider
{
     public function retrieveById($identifier);
     public function retrieveByToken($identifier, $token);
     public function updateRememberToken(Authenticatable $user, $token);
     public function retrieveByCredentials(array $credentials);
     public function validateCredentials(Authenticatable $user, array $credentials);
}
```

`retrieveById` 函數通常接收表示使用者的主鍵，例如 MySQL 資料庫中的自動遞增 ID。方法應檢索並返回與 ID 匹配的 `Authenticatable` 實現。

`retrieveByToken` 函數通過使用者唯一的 `$identifier` 和 「記住我」的 `$token` 檢索使用者，通常儲存在資料庫列中，如 `remember_token`。與前面的方法一樣，此方法應返回具有匹配令牌值的 `Authenticatable` 實現。

`updateMemberToken` 方法使用新的 `$token` 更新 `$user` 實例的 `remember_token`。在成功的「記住我」身份驗證嘗試或使用者註銷時，會將新令牌分配給使用者。

當嘗試對應用程式進行身份驗證時， `retrieveByCredentials` 方法接收傳遞給 `Auth::attempt` 方法的憑據陣列。然後，該方法應該「查詢」底層的持久性儲存以尋找與這些憑據匹配的使用者。通常，此方法將運行帶有「where」條件的查詢，以搜尋「username」與 `$credentials['username']` 的值匹配的使用者記錄。該方法應返回 `Authenticatable` 的實現。 **此方法不應嘗試執行任何密碼驗證或身份驗證**。

`validateCredentials` 方法應將給定的 `$user` 與 `$credentials` 進行比較，以對使用者進行身份驗證。例如，此方法通常會使用 `Hash::check` 方法將 `$user->getAuthPassword()` 的值與 `$credentials['password']` 的值進行比較。此方法應返回 `true` 或 `false`，指示密碼是否有效。

35.8.2 使用者認證契約

現在我們已經探索了 `UserProvider` 上的每個方法，現在讓我們看看 `Authenticatable` 契約。請記住，`UserProvider` 應該從 `retrieveById`、`retrieveByToken` 和 `retrieveByCredentials` 方法返回此介面的實現：

```
<?php

namespace Illuminate\Contracts\Auth;

interface Authenticatable
{
     public function getAuthIdentifierName();
     public function getAuthIdentifier();
     public function getAuthPassword();
     public function getRememberToken();
     public function setRememberToken($value);
     public function getRememberTokenName();
}
```

這個介面很簡單。`getAuthIdentifierName` 方法應返回使用者的「主鍵」欄位的名稱，`getAuthIdentifier` 方法應返回使用者的「主鍵」。當使用 MySQL 後端時，這可能是分配給使用者記錄的自動遞增主鍵。`getAuthPassword` 方法應返回使用者的 hash 密碼。

此介面允許身份驗證系統與任何「使用者」類一起工作，而不管你使用的是哪個 ORM 或儲存抽象層。默認情況下，Laravel 在實現此介面的 `app/Models` 目錄中包含一個 `App\Models\User` 類。

35.9 事件

在身份驗證過程中，Laravel 調度各種 事件 。你可以在 `EventServiceProvider` 中將監聽器附加到這些事件上：

```
/**
 * 應用事件監聽對應
 *
 * @var array
 */
protected $listen = [
     'Illuminate\Auth\Events\Registered' => [
          'App\Listeners\LogRegisteredUser',
     ],

     'Illuminate\Auth\Events\Attempting' => [
          'App\Listeners\LogAuthenticationAttempt',
     ],

     'Illuminate\Auth\Events\Authenticated' => [
          'App\Listeners\LogAuthenticated',
     ],
```

```
    'Illuminate\Auth\Events\Login' => [
        'App\Listeners\LogSuccessfulLogin',
    ],

    'Illuminate\Auth\Events\Failed' => [
        'App\Listeners\LogFailedLogin',
    ],

    'Illuminate\Auth\Events\Validated' => [
        'App\Listeners\LogValidated',
    ],

    'Illuminate\Auth\Events\Verified' => [
        'App\Listeners\LogVerified',
    ],

    'Illuminate\Auth\Events\Logout' => [
        'App\Listeners\LogSuccessfulLogout',
    ],

    'Illuminate\Auth\Events\CurrentDeviceLogout' => [
        'App\Listeners\LogCurrentDeviceLogout',
    ],

    'Illuminate\Auth\Events\OtherDeviceLogout' => [
        'App\Listeners\LogOtherDeviceLogout',
    ],

    'Illuminate\Auth\Events\Lockout' => [
        'App\Listeners\LogLockout',
    ],

    'Illuminate\Auth\Events\PasswordReset' => [
        'App\Listeners\LogPasswordReset',
    ],
];
```

36 使用者授權

36.1 簡介

除了提供內建的 authentication（身份驗證）服務外，Laravel 還提供了一種可以很簡單就進行使用的方法，來對使用者與資源的授權關係進行管理。 它很安全，即使使用者已經通過了「身份驗證（authentication)」, 使用者也可能無權對應用程式中重要的模型或資料庫記錄進行刪除或更改。簡單、條理化的系統性，是 Laravel 對授權管理的特性。

Laravel 主要提供了兩種授權操作的方法: 攔截器和策略。 可以把攔截器（gates）和策略（policies）想像成路由和 controller 。攔截器（Gates）提供了一種輕便的基於閉包涵數的授權方法，像是路由。而策略（policies)，就像是一個 controller ，對特定模型或資源，進行分組管理的邏輯規則。 在本文件中，我們將首先探討攔截器（gates），然後研究策略（policies)。

你在建構應用程式時，不用為是僅僅使用攔截器（gates）或是僅僅使用策略（policies）而擔心，並不需要在兩者中進行唯一選擇。大多數的應用程式都同時包含兩個方法，並且同時使用兩者，能夠更好的進行工作。攔截器（gates），更適用於沒有與任何模型或資源有關的授權操作，例如查看管理員儀表盤。與之相反，當你希望為特定的模型或資源進行授權管理時，應該使用策略（policies) 方法。

36.2 攔截器 (Gates)

36.2.1 編寫攔截器（Gates）

> **注意**
> 通過理解攔截器（Gates），是一個很好的學習 Laravel 授權特性的基礎知識的方法。同時，考慮到 Laravel 應用程式的健壯性，應該結合使用策略 policies 來組織授權規則。

攔截器（Gates）是用來確定使用者是否有權執行給定操作的閉包涵數。默認條件下，攔截器（Gates）的使用，是在 `App\Providers\AuthServiceProvider` 類中的 `boot` 函數里來規定 `Gate` 規則。攔截器（Gates）始終接收使用者實例為其第一個參數，並且可以選擇性的接收其他參數，例如相關的 Eloquent 模型。

在下面的例子中，我們將定義一個攔截器（Gates)，並通過呼叫 `App\Models\Post` 類，來實現結合使用者的 POST 請求，命中給定的規則。攔截器（Gates）將通過比較使用者的 `id`，和 POST 請求中的 `user_id` 來實現這個目標：

```
use App\Models\Post;
use App\Models\User;
use Illuminate\Support\Facades\Gate;

/**
 * 註冊任何需要身份驗證、授權服務的行為
 */
public function boot(): void
{
    Gate::define('update-post', function (User $user, Post $post)
    {
        return $user->id === $post->user_id;
    });
}
```

像是在 controller 中操作一樣，也可以直接使用類，進行回呼陣列，完成攔截器（Gates）的定義：

```
use App\Policies\PostPolicy;
use Illuminate\Support\Facades\Gate;

/**
 * 註冊任何需要身份驗證、授權服務的行為
 */
public function boot(): void
{
    Gate::define('update-post', [PostPolicy::class, 'update']);
}
```

36.2.2 授權動作

如果需要通過攔截器（Gates）來對行為進行授權控制，你可以通過呼叫 `Gate` 中的 `allows` 或 `denies` 方法。請注意，在使用過程中，你不需要將已經通過身份驗證的使用者資訊傳遞給這些方法。 Laravel 將會自動把使用者資訊傳遞給攔截器（Gates）。以下是一個典型的，在 controller 中使用攔截器（Gates）進行行為授權控制的例子：

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Gate;

class PostController extends Controller
{
    /**
     * 更新給定的帖子
     */
    public function update(Request $request, Post $post): RedirectResponse
    {
        if (! Gate::allows('update-post', $post)) {
            abort(403);
        }

        // 更新帖子...

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

如果你需要判斷某個使用者，是否有權執行某個行為，你可以在 `Gate` 門面中，使用 `forUser` 方法：

```
if (Gate::forUser($user)->allows('update-post', $post)) {
    // 這個使用者可以提交 update...
}

if (Gate::forUser($user)->denies('update-post', $post)) {
    // 這個使用者不可以提交 update...
}
```

你還可以通過 any 或 none 方法來一次性授權多個行為:

```
if (Gate::any(['update-post', 'delete-post'], $post)) {
    // 使用者可以提交 update 或 delete...
}

if (Gate::none(['update-post', 'delete-post'], $post)) {
    // 使用者不可以提交 update 和 delete...
}
```

36.2.2.1 未通過授權時的拋出異常

`Illuminate\Auth\Access\AuthorizationException` 中準備了 HTTP 的 403 響應。你可以使用 `Gate` 門面中的 `authorize` 方法，來規定如果使用者進行了未授權的行為時，觸發 `AuthorizationException` 實例 ，該實例會自動轉換返回為 HTTP 的 403 響應:

```
Gate::authorize('update-post', $post);

// 行為已獲授權...
```

36.2.2.2 上下文的值傳遞

能夠用於攔截器（Gates）的授權方法，(`allows`，`denies`，`check`，`any`，`none`，`authorize`，`can`，`cannot`) 和在前端進行的授權方法 Blade 指令 (`@can`，`@cannot`，`@canany`) 在第 2 個參數中，可以接收陣列。這些陣列元素作為參數傳遞給攔截器（Gates），在做出授權決策時可用於其他上下文:

```
use App\Models\Category;
use App\Models\User;
use Illuminate\Support\Facades\Gate;

Gate::define('create-post', function (User $user, Category $category, bool $pinned) {
    if (! $user->canPublishToGroup($category->group)) {
        return false;
    } elseif ($pinned && ! $user->canPinPosts()) {
        return false;
    }

    return true;
});

if (Gate::check('create-post', [$category, $pinned])) {
    // 使用者可以請求 create...
}
```

36.2.3 攔截器響應

到目前為止，我們只學習了攔截器（Gates）中返回布林值的簡單操作。但是，有時你需要的返回可能更複雜，比如錯誤消息。所以，你可以嘗試使用 `Illuminate\Auth\Access\Response` 來建構你的攔截器（Gates）：

```
use App\Models\User;
use Illuminate\Auth\Access\Response;
use Illuminate\Support\Facades\Gate;

Gate::define('edit-settings', function (User $user) {
    return $user->isAdmin
                ? Response::allow()
                : Response::deny('You must be an administrator.');
});
```

你希望從攔截器（Gates）中返迴響應時，使用 `Gate::allows` 方法，將僅返回一個簡單的布林值；同時，你還可以使用 `Gate::inspect` 方法來返回攔截器（Gates）中的所有響應值：

```
$response = Gate::inspect('edit-settings');

if ($response->allowed()) {
    // 行為進行授權...
} else {
    echo $response->message();
}
```

在使用 `Gate::authorize` 方法時，如果操作未被授權，仍然會觸發 `AuthorizationException`, 使用者驗證（authorization）響應提供的錯誤消息，將傳遞給 HTTP 響應：

```
Gate::authorize('edit-settings');

// 行為進行授權...
```

36.2.3.1 自訂 HTTP 響應狀態

當一個操作通過 Gate 被拒絕時，返回一個 403HTTP 響應；然而，有時返回一個可選的 HTTP 狀態程式碼是有用的。你可以使用 `Illuminate\Auth\Access\Response` 類上的 `denyWithStatus` 靜態建構函式自訂授權檢查失敗返回的 HTTP 狀態程式碼：

```
use App\Models\User;
use Illuminate\Auth\Access\Response;
use Illuminate\Support\Facades\Gate;

Gate::define('edit-settings', function (User $user) {
    return $user->isAdmin
                ? Response::allow()
                : Response::denyWithStatus(404);
});
```

由於通過 `404` 響應隱藏資源是 Web 應用程式的常見模式，為了方便起見，提供了 `denyAsNotFound` 方法:

```
use App\Models\User;
use Illuminate\Auth\Access\Response;
use Illuminate\Support\Facades\Gate;

Gate::define('edit-settings', function (User $user) {
    return $user->isAdmin
                ? Response::allow()
                : Response::denyAsNotFound();
});
```

36.2.4 攔截 Gate 檢查

有時，你可能希望將所有能力授予特定使用者。你可以使用 `before` 方法定義一個閉包，在所有其他授權檢查之前運行:

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

Gate::before(function (User $user, string $ability) {
    if ($user->isAdministrator()) {
        return true;
    }
});
```

如果 `before` 返回的是非 null 結果，則該返回將會被視為最終的檢查結果。

你還可以使用 `after` 方法，來定義在所有授權攔截規則執行後，再次進行授權攔截規則判定：

```
use App\Models\User;

Gate::after(function (User $user, string $ability, bool|null $result, mixed $arguments)
{
    if ($user->isAdministrator()) {
        return true;
    }
});
```

類似於 `before` 方法，如果 `after` 閉包返回非空結果，則該結果將被視為授權檢查的結果。

36.2.5 內聯授權

有時，你可能希望確定當前經過身份驗證的使用者是否有權執行給定操作，而無需編寫與該操作對應的專用攔

截器。Laravel 允許你通過 Gate::allowIf 和 Gate::denyIf 方法執行這些類型的「內聯」授權檢查：

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

Gate::allowIf(fn (User $user) => $user->isAdministrator());

Gate::denyIf(fn (User $user) => $user->banned());
```

如果該操作未授權或當前沒有使用者經過身份驗證，Laravel 將自動拋出 Illuminate\Auth\Access\AuthorizationException 異常。AuthorizationException 的實例會被 Laravel 的異常處理程序自動轉換為 403 HTTP 響應：

36.3 生成策略

36.3.1 註冊策略

策略是圍繞特定模型或資源組織授權邏輯的類。例如，如果你的應用程式是部落格，可能有一個 App\Models\Post 模型和一個相應的 App\Policies\PostPolicy 來授權使用者操作，例如建立或更新帖子。

你可以使用 make:policyArtisan 命令生成策略。生成的策略將放置在 app/Policies 目錄中。如果應用程式中不存在此目錄，Laravel 將自動建立：

```
php artisan make:policy PostPolicy
```

make:policy 命令將生成一個空的策略類。如果要生成一個包含與查看、建立、更新和刪除資源相關的示例策略方法的類，可以在執行命令時提供一個--model 選項：

```
php artisan make:policy PostPolicy --model=Post
```

36.3.2 註冊策略

建立了策略類之後，還需要對其進行註冊。註冊策略是告知 Laravel 在授權針對給定模型類型的操作時使用哪個策略。

新的 Laravel 應用程式中包含的 App\Providers\AuthServiceProvider 包含一個 policies 屬性，它將 Eloquent 模型對應到其相應的策略。 註冊策略將指示 Laravel 在授權針對給定 Eloquent 模型的操作時使用哪個策略：

```
<?php

namespace App\Providers;

use App\Models\Post;
use App\Policies\PostPolicy;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;
use Illuminate\Support\Facades\Gate;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * 應用程式的策略對應。
     *
     * @var array
     */
    protected $policies = [
        Post::class => PostPolicy::class,
    ];

    /**
```

```
     * 註冊任何應用程式身份驗證/授權服務。
     */
    public function boot(): void
    {
        // ...
    }
}
```

36.3.2.1 策略自動發現

只要模型和策略遵循標準的 Laravel 命名約定，Laravel 就可以自動發現策略，而不是手動註冊模型策略。具體來說，策略必須位於包含模型的目錄或其上方的「Policies」目錄中。 因此，例如，模型可以放置在 `app/Models` 目錄中，而策略可以放置在 `app/Policies` 目錄中。在這種情況下，Laravel 將檢查 `app/Models/Policies` 然後 `app/Policies` 中的策略。此外，策略名稱必須與模型名稱匹配並具有「策略」後綴。 因此，`User` 模型將對應於 `UserPolicy` 策略類。

如果要自訂策略的發現邏輯，可以使用 `Gate::guessPolicyNamesUsing` 方法註冊自訂策略發現回呼。通常，應該從應用程式的 `AuthServiceProvider` 的 `boot` 方法呼叫此方法：

```
use Illuminate\Support\Facades\Gate;

Gate::guessPolicyNamesUsing(function (string $modelClass) {
    // 返回給定模型的策略類的名稱…
});
```

> **注意**
> 在 `AuthServiceProvider` 中顯式對應的任何策略將優先於任何可能自動發現的策略。

36.4 編寫策略

36.4.1 策略方法

註冊策略類後，可以為其授權的每個操作新增方法。例如，讓我們在 `PostPolicy` 上定義一個 `update` 方法，該方法確定給定的 `App\Models\User` 是否可以更新給定的 `App\Models\Post` 實例。

該 `update` 方法將接收一個 `User` 和一個 `Post` 實例作為其參數，並應返回 `true` 或 `false`，指示使用者是否有權更新給定的 `Post`。因此，在本例中，我們將驗證使用者的 `id` 是否與 Post 上的 `user_id` 匹配：

```
<?php

namespace App\Policies;

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

class PostPolicy
{
    /**
     * 確定使用者是否可以更新給定的帖子
     */
    public function update(User $user, Post $post): bool
    {
        return $user->id === $post->user_id;
    }
}
```

你可以繼續根據需要為策略授權的各種操作定義其他方法。例如，你可以定義 `view` 或 `delete` 方法來授權各種與 `Post` 相關的操作，但請記住，你可以自由地為策略方法命名任何你喜歡的名稱。

如果你在 Artisan 控制台生成策略時使用了`--model`選項，它將包含用於`viewAny`、`view`、`create`、`update`、`delete`、`restore`和`forceDelete`操作。

> **技巧**
> 所有策略都通過 Laravel 服務容器解析，允許你在策略的建構函式中鍵入任何需要的依賴項，以自動注入它們。

36.4.2 策略響應

到目前為止，我們只檢查了返回簡單布林值的策略方法。但是，有時你可能希望返回更詳細的響應，包括錯誤消息。為此，你可以從你的策略方法返回一個`Illuminate\Auth\Access\Response`實例：

```
use App\Models\Post;
use App\Models\User;
use Illuminate\Auth\Access\Response;

/**
 * 確定使用者是否可以更新給定的帖子。
 */
public function update(User $user, Post $post): Response
{
    return $user->id === $post->user_id
                ? Response::allow()
                : Response::deny('你不擁有這個帖子。');
}
```

當從你的策略返回授權響應時，`Gate::allows`方法仍將返回一個簡單的布林值；但是，你可以使用`Gate::inspect`方法來獲取返回的完整授權響應：

```
use Illuminate\Support\Facades\Gate;

$response = Gate::inspect('update', $post);

if ($response->allowed()) {
    // 操作已被授權…
} else {
    echo $response->message();
}
```

當使用`Gate::authorize`方法時，如果操作未被授權，該方法會拋出`AuthorizationException`，授權響應提供的錯誤消息將傳播到 HTTP 響應：

```
Gate::authorize('update', $post);

// 該操作已授權通過...
```

36.4.2.1 自訂 HTTP 響應狀態

當一個操作通過策略方法被拒絕時，返回一個`403`HTTP 響應;然而，有時返回一個可選的 HTTP 狀態程式碼是有用的。你可以使用`Illuminate\Auth\Access\Response`類上的`denyWithStatus`靜態建構函式自訂授權檢查失敗返回的 HTTP 狀態程式碼:

```
use App\Models\Post;
use App\Models\User;
use Illuminate\Auth\Access\Response;

/**
 * 確定使用者是否可以更新給定的帖子。
 */
public function update(User $user, Post $post): Response
{
    return $user->id === $post->user_id
                ? Response::allow()
```

```
                : Response::denyWithStatus(404);
}
```

由於通過 404 響應隱藏資源是 Web 應用程式的常見模式，為了方便起見，提供了 denyAsNotFound 方法:

```
use App\Models\Post;
use App\Models\User;
use Illuminate\Auth\Access\Response;

/**
 * 確定使用者是否可以更新給定的帖子。
 */
public function update(User $user, Post $post): Response
{
    return $user->id === $post->user_id
                ? Response::allow()
                : Response::denyAsNotFound();
}
```

36.4.3 無需傳遞模型的方法

一些策略方法只接收當前經過身份驗證的使用者實例，最常見的情況是給 create 方法做授權。例如，如果你正在建立一個部落格，你可能希望確定一個使用者是否被授權建立任何文章，在這種情況下，你的策略方法應該只期望接收一個使用者實例：

```
/**
 * 確定給定使用者是否可以建立檔案
 */
public function create(User $user): bool
{
    return $user->role == 'writer';
}
```

36.4.4 Guest 使用者

默認情況下，如果傳入的 HTTP 請求不是經過身份驗證的使用者發起的，那麼所有的攔截器（gates）和策略（policies）會自動返回 false。但是，你可以通過聲明一個「optional」類型提示或為使用者參數定義提供一個 null 預設值，從而允許這些授權檢查通過你的攔截器（gates）和策略（policies）：

```
<?php

namespace App\Policies;

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

class PostPolicy
{
    /**
     * 確定使用者是否可以更新給定的文章
     */
    public function update(?User $user, Post $post): bool
    {
        return $user?->id === $post->user_id;
    }
}
```

36.4.5 策略過濾器

對於某些使用者，你可能希望給他授權給定策略中的所有操作。為了實現這一點，你可以在策略上定義一個 before 方法。該 before 方法將在策略上的所有方法之前執行，這樣就使你有機會在實際呼叫預期的策略方

法之前就已經授權了操作。該功能常用於授權應用程式管理員來執行任何操作：

```
use App\Models\User;

/**
 * 執行預先授權檢查
 */
public function before(User $user, string $ability): bool|null
{
    if ($user->isAdministrator()) {
        return true;
    }

    return null;
}
```

如果你想拒絕特定類型使用者的所有授權檢查，那麼你可以從 `before` 方法返回 `false`。如果返回 `null`，則授權檢查將通過策略方法進行。

> **注意**
> 如果策略類中不包含名稱與被檢查能力的名稱相匹配的方法，則不會呼叫策略類的 `before` 方法。

36.5 使用策略進行授權操作

36.5.1 通過使用者模型

Laravel 應用程式中的 `App\Models\User` 型提供了兩個用於授權操作的方法：`can` 和 `cannot`。`can` 和 `cannot` 方法接收你希望授權的操作名稱和相關模型。例如，讓我們確定一個使用者是否被授權更新給定的 `App\Models\Post` 模型，這通常在 controller 方法中實現：

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PostController extends Controller
{
    /**
     * 更新給定的帖子。
     */
    public function update(Request $request, Post $post): RedirectResponse
    {
        if ($request->user()->cannot('update', $post)) {
            abort(403);
        }

        // 更新帖子...

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

如果為給定模型註冊了策略，該 `can` 方法將自動呼叫適當的策略並返回布林值；如果沒有為模型註冊策略，該 `can` 方法將嘗試呼叫基於 Gate 的閉包，該閉包將匹配給定的操作名稱。

36.5.1.1 不需要指定模型的操作

請記住，某些操作可能對應著「不需要模型實例」的策略方法，比如 `create`。在這些情況下，你可以將類名傳遞給 `can` 方法，類名將用於確定在授權操作時使用哪個策略：

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PostController extends Controller
{
    /**
     * 建立一個帖子。
     */
    public function store(Request $request): RedirectResponse
    {
        if ($request->user()->cannot('create', Post::class)) {
            abort(403);
        }

        // 建立帖子…

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

36.5.2 通過 controller 輔助函數

除了給 `App\Models\User` 模型提供了有用方法，Laravel 還給任何 controller 提供了一個有用的 `authorize` 方法，這些 controller 要繼承（`extends`）`App\Http\Controllers\Controller` 基類。

與 `can` 方法一樣，`authorize` 方法接收你希望授權的操作名稱和相關模型，如果該操作未被授權，該方法將拋出 `Illuminate\Auth\Access\AuthorizationException` 異常，Laravel 的異常處理程序將自動將該異常轉換為一個帶有 403 狀態碼的 HTTP 響應：

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class PostController extends Controller
{
    /**
     * 更新指定的部落格文章
     *
     * @throws \Illuminate\Auth\Access\AuthorizationException
     */
    public function update(Request $request, Post $post): RedirectResponse
    {
        $this->authorize('update', $post);

        // 當前使用者可以更新部落格文章…

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

```
}
```

36.5.2.1 不需要指定模型的操作

如前所述，一些策略方法 如 `create` 不需要模型實例，在這些情況下，你應該給 `authorize` 方法傳遞一個類名，該類名將用來確定在授權操作時使用哪個策略：

```
use App\Models\Post;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

/**
 * 建立一個新的部落格文章。
 *
 * @throws \Illuminate\Auth\Access\AuthorizationException
 */
public function create(Request $request): RedirectResponse
{
    $this->authorize('create', Post::class);

    // 當前使用者可以建立部落格帖子…

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

36.5.2.2 授權資源 controller

如果你正在使用資源 controller，你可以在 controller 的構造方法中使用 `authorizeResource` 方法，該方法將把適當的 `can` 中介軟體定義附加到資源 controller 的方法上。

該 `authorizeResource` 方法的第一個參數是模型的類名，第二個參數是包含模型 ID 的 路由/請求參數的名稱。你應該確保你的資源 controller 是使用 `--model` 標誌建立的，這樣它才具有所需的方法簽名和類型提示。

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\Request;

class PostController extends Controller
{
    /**
     * 建立 controller 實例
     */
    public function __construct()
    {
        $this->authorizeResource(Post::class, 'post');
    }
}
```

以下 controller 方法將對應到其相應的策略方法。當請求被路由到給定的 controller 方法時，會在 controller 方法執行之前自動呼叫相應的策略方法：

controller 方法	策略方法
index	viewAny
show	view
create	create
store	create
edit	update
update	update
destroy	delete

技巧

你可以使用帶有 make:policy 帶有 --model 選項的命令，快速的為給定模型生成一個策略類：php artisan make:policy PostPolicy --model=Post。

36.5.3 通過中介軟體

Laravel 包含一個中介軟體，可以在傳入的請求到達路由或 controller 之前對操作進行授權。默認情況下，Illuminate\Auth\Middleware\Authorize 中介軟體會在 App\Http\Kernel 中的 can 鍵中被指定。讓我們來看一個使用 can 中介軟體授權使用者更新部落格文章的例子：

```
use App\Models\Post;

Route::put('/post/{post}', function (Post $post) {
    // 當前使用者可以更新帖子…
})->middleware('can:update,post');
```

在這個例子中，我們給 can 中介軟體傳遞了兩個參數。第一個是我們希望授權操作的名稱，第二個是我們希望傳遞給策略方法的路由參數。在這個例子中，當我們使用了隱式模型繫結後，一個 App\Models\Post 模型就將被傳遞給對應的策略方法。如果使用者沒有被授權執行給定操作的權限，那麼中介軟體將會返回一個帶有 403 狀態碼的 HTTP 響應。

為了方便起見，你也可以使用 can 方法將 can 中介軟體繫結到你的路由上：

```
use App\Models\Post;

Route::put('/post/{post}', function (Post $post) {
    // 當前使用者可以更新文章...
})->can('update', 'post');
```

36.5.3.1 不需要指定模型的操作

同樣的，一些策略方法不需要模型實例，比如 create。在這些情況下，你可以給中介軟體傳遞一個類名。這個類名將用來確定在授權操作時使用哪個策略：

```
Route::post('/post', function () {
    // 當前使用者可以建立文章...
})->middleware('can:create,App\Models\Post');
```

在一個中介軟體中定義整個類名會變得難以維護。因此，你也可以選擇使用 can 方法將 can 中介軟體繫結到你的路由上：

```
use App\Models\Post;

Route::post('/post', function () {
    // 當前使用者可以建立文章
})->can('create', Post::class);
```

36.5.4 通過 Blade 範本

當編寫 Blade 範本時，你可能希望只展示給使用者有權限操作的資料。例如，你可能希望當使用者具有更新文章的權限時才展示更新部落格文章的表單。在這種情況下，你可以使用@can 和@cannot 指令：

```
@can('update', $post)
    <!-- 當前使用者可更新的文章... -->
@elsecan('create', App\Models\Post::class)
    <!-- 當前使用者可建立新文章... -->
@else
    <!-- ... -->
@endcan

@cannot('update', $post)
    <!-- 當前使用者不可更新的文章... -->
```

```
@elsecannot('create', App\Models\Post::class)
    <!-- 當前使用者不可建立新文章... -->
@endcannot
```

這些指令是編寫@if 和@unless 語句的快捷方式。上面的@can 和@cannot 語句相當於下面的語句：

```
@if (Auth::user()->can('update', $post))
    <!-- 當前使用者可更新的文章... -->
@endif

@unless (Auth::user()->can('update', $post))
    <!-- 當前使用者不可更新的文章... -->
@endunless
```

你還可以確定一個使用者是否被授權從給定的運算元組中執行任何操作，要做到這一點，可以使用@canany 指令：

```
@canany(['update', 'view', 'delete'], $post)
    <!-- 當前使用者可以更新、查看、刪除文章... -->
@elsecanany(['create'], \App\Models\Post::class)
    <!-- 當前使用者可以建立新文章... -->
@endcanany
```

36.5.4.1 不需要執行模型的操作

像大多數其他授權方法一樣，如果操作不需要模型實例，你可以給@can 和@cannot 指令傳遞一個類名：

```
@can('create', App\Models\Post::class)
    <!-- 當前使用者可以建立文章... -->
@endcan

@cannot('create', App\Models\Post::class)
    <!-- 當前使用者不能建立文章... -->
@endcannot
```

36.5.5 提供額外的上下文

在使用策略授權操作時，可以將陣列作為第二個參數傳遞給授權函數和輔助函數。陣列中的第一個元素用於確定應該呼叫哪個策略，其餘的陣列元素作為參數傳遞給策略方法，並可在作出授權決策時用於額外的上下文中。例如，考慮下面的 PostPolicy 方法定義，它包含一個額外的 $category 參數：

```
/**
 * 確認使用者是否可以更新給定的文章。
 */
public function update(User $user, Post $post, int $category): bool
{
    return $user->id === $post->user_id &&
           $user->canUpdateCategory($category);
}
```

當嘗試確認已驗證過的使用者是否可以更新給定的文章時，我們可以像這樣呼叫此策略方法：

```
/**
 * 更新給定的部落格文章
 *
 * @throws \Illuminate\Auth\Access\AuthorizationException
 */
public function update(Request $request, Post $post): RedirectResponse
{
    $this->authorize('update', [$post, $request->category]);

    // 當前使用者可以更新部落格文章...

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

37 Email 認證

37.1 簡介

很多 Web 應用會要求使用者在使用之前進行 Email 地址驗證。Laravel 不會強迫你在每個應用中重複實現它，而是提供了便捷的方法來傳送和校驗電子郵件的驗證請求。

> **技巧** 想快速上手嗎？你可以在全新的應用中安裝 Laravel 應用入門套件 。入門套件將幫助你搭建整個身份驗證系統，包括電子郵件驗證支援。

37.1.1 準備模型

在開始之前，需要檢查你的 `App\Models\User` 模型是否實現了 `Illuminate\Contracts\Auth\MustVerifyEmail` 契約：

```
<?php

namespace App\Models;

use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable implements MustVerifyEmail
{
    use Notifiable;

    // ...
}
```

一旦這一介面被新增到模型中，新註冊的使用者將自動收到一封包含電子郵件驗證連結的電子郵件。檢查你的 `App\Providers\EventServiceProvider` 可以看到，Laravel 已經為 `Illuminate\Auth\Events\Registered` 事件註冊了一個 `SendEmailVerificationNotification` 監聽器 。這個事件監聽器會通過郵件傳送驗證連結給使用者。 如果在應用中你沒有使用 入門套件 而是手動實現的註冊，你需要確保在使用者註冊成功後手動分發 `Illuminate\Auth\Events\Registered` 事件：

```
use Illuminate\Auth\Events\Registered;

event(new Registered($user));
```

37.1.2 資料庫準備

接下來，你的 `users` 表必須有一個 `email_verified_at` 欄位，用來儲存使用者信箱驗證的日期和時間。Laravel 框架自帶的 `users` 表以及默認包含了該欄位。因此，你只需運行資料庫遷移即可：

```
php artisan migrate
```

37.2 路由

為了實現完整的電子郵件驗證流程，你將需要定義三個路由。首先，需要定義一個路由向使用者顯示通知，告訴使用者應該點選註冊之後， Laravel 向他們傳送的驗證郵件中的連結。

其次，需要一個路由來處理使用者點選郵件中驗證連結時發來的請求。

第三，如果使用者沒有收到驗證郵件，則需要一路由來重新傳送驗證郵件。

37.2.1 信箱驗證通知

如上所述，應該定義一條路由，該路由將返回一個檢視，引導使用者點選註冊後 Laravel 傳送給他們郵件中的驗證連結。當使用者嘗試存取網站的其它頁面而沒有先完成信箱驗證時，將向使用者顯示此檢視。請注意，只要您的 `App\Models\User` 模型實現了 `MustVerifyEmail` 介面，就會自動將該連結發郵件給使用者：

```
Route::get('/email/verify', function () {
    return view('auth.verify-email');
})->middleware('auth')->name('verification.notice');
```

顯示信箱驗證的路由，應該命名為 `verification.notice`。組態這個命名路由很重要，因為如果使用者信箱驗證未通過，Laravel 自帶的 `verified` 中介軟體 將會自動重新導向到該命名路由上。

> **注意**
> 手動實現信箱驗證過程時，你需要自己定義驗證通知檢視。如果你希望包含所有必要的身份驗證和驗證檢視，請查看 Laravel 應用入門套件

37.2.2 Email 認證處理

接下來，我們需要定義一個路由，該路由將處理當使用者點選驗證連結時傳送的請求。該路由應命名為 `verification.verify` ，並新增了 `auth` 和 `signed` 中介軟體

```
use Illuminate\Foundation\Auth\EmailVerificationRequest;

Route::get('/email/verify/{id}/{hash}', function (EmailVerificationRequest $request) {
    $request->fulfill();

    return redirect('/home');
})->middleware(['auth', 'signed'])->name('verification.verify');
```

在繼續之前，讓我們仔細看一下這個路由。首先，您會注意到我們使用的是 `EmailVerificationRequest` 請求類型，而不是通常的 `Illuminate\Http\Request` 實例。 `EmailVerificationRequest` 是 Laravel 中包含的 表單請求。此請求將自動處理驗證請求的 id 和 hash 參數。

接下來，我們可以直接在請求上呼叫 `fulfill` 方法。該方法將在經過身份驗證的使用者上呼叫 `markEmailAsVerified` 方法，並會觸發 `Illuminate\Auth\Events\Verified` 事件。通過 `Illuminate\Foundation\Auth\User` 基類，`markEmailAsVerified` 方法可用於默認的 `App\Models\User` 模型。驗證使用者的電子郵件地址後，您可以將其重新導向到任意位置。

37.2.3 重新傳送 Email 認證郵件

有時候，使用者可能輸錯了電子郵件地址或者不小心刪除了驗證郵件。為瞭解決這種問題，您可能會想定義一個路由實現使用者重新傳送驗證郵件。您可以通過在 驗證通知檢視 中放置一個簡單的表單來實現此功能。

```
use Illuminate\Http\Request;

Route::post('/email/verification-notification', function (Request $request) {
    $request->user()->sendEmailVerificationNotification();

    return back()->with('message', 'Verification link sent!');
})->middleware(['auth', 'throttle:6,1'])->name('verification.send');
```

37.2.4 保護路由

路由中介軟體可用於僅允許經過驗證的使用者訪問給定路由。Laravel 附帶了一個 `verified` 中介軟體別名，它是 `Illuminate\Auth\Middleware\EnsureEmailIsVerified` 類的別名。由於該中介軟體已經在你的應用程式的 HTTP 核心中註冊，所以你只需要將中介軟體附加到路由定義即可。通常，此中介軟體與 `auth` 中介軟體配對使用。

```
Route::get('/profile', function () {
    // 僅經過驗證的使用者可以訪問此路由。。。
})->middleware(['auth', 'verified']);
```

如果未經驗證的使用者嘗試訪問已被分配了此中介軟體的路由，他們將自動重新導向到 `verification.notice` 命名路由。

37.3 自訂

37.3.1.1 驗證郵件自訂

雖然默認的電子郵件驗證通知應該能夠滿足大多數應用程式的要求，但 Laravel 允許你自訂如何建構電子郵件驗證郵件消息。

要開始自訂郵件驗證消息，你需要將一個閉包傳遞給 `Illuminate\Auth\Notifications\VerifyEmail` 通知提供的 `toMailUsing` 方法。該閉包將接收到通知的可通知模型實例以及使用者必須訪問以驗證其電子郵件地址的已簽名電子郵件驗證 URL。該閉包應返回 `Illuminate\Notifications\Messages\MailMessage` 的實例。通常，你應該從應用程式的 `App\Providers\AuthServiceProvider` 類的 `boot` 方法中呼叫 `toMailUsing` 方法：

```
use Illuminate\Auth\Notifications\VerifyEmail;
use Illuminate\Notifications\Messages\MailMessage;

/**
 * 註冊任何身份驗證/授權服務。
 */
public function boot(): void
{
    // ...

    VerifyEmail::toMailUsing(function (object $notifiable, string $url) {
        return (new MailMessage)
            ->subject('Verify Email Address')
            ->line('Click the button below to verify your email address.')
            ->action('Verify Email Address', $url);
    });
}
```

> 技巧：要瞭解更多有關郵件通知的資訊，請參閱 郵件通知文件。

37.4 事件

如果你是使用 Laravel 應用入門套件 的話，Laravel 在電子郵件驗證通過後會派發 事件 。如果你想接收到這個事件並進行手動處理的話，你應該在 `EventServiceProvider` 中註冊監聽器：

```
use App\Listeners\LogVerifiedUser;
use Illuminate\Auth\Events\Verified;

/**
 * 應用的事件監聽器
```

```
     *
     * @var array
     */
    protected $listen = [
        Verified::class => [
            LogVerifiedUser::class,
        ],
    ];
```

38 加密解密

38.1 簡介

Laravel 的加密服務提供了一個簡單、方便的介面，使用 OpenSSL 所提供的 AES-256 和 AES-128 加密和解密文字。所有 Laravel 加密的結果都會使用消息認證碼 (MAC) 進行簽名，因此一旦加密，其底層值就不能被修改或篡改。

38.2 組態

在使用 Laravel 的加密工具之前，你必須先設定 `config/app.php` 組態檔案中的 `key` 組態項。該組態項由環境變數 `APP_KEY` 設定。你應當使用 `php artisan key:generate` 命令來生成該變數的值，`key:generate` 命令將使用 PHP 的安全隨機位元組生成器為你的應用程式建構加密安全金鑰。通常情況下，在 Laravel 安裝 中會為你生成 APP_KEY 環境變數的值。

38.3 基本用法

38.3.1.1 加密一個值

你可以使用 `Crypt` 門面提供的 `encryptString` 方法來加密一個值。所有加密的值都使用 OpenSSL 的 AES-256-CBC 來進行加密。此外，所有加密過的值都會使用消息認證碼 (MAC) 來簽名，可以防止惡意使用者對值進行篡改：

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\User;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Crypt;

class DigitalOceanTokenController extends Controller
{
    /**
     *  為使用者儲存一個 DigitalOcean API 令牌。
     */
    public function store(Request $request): RedirectResponse
    {
        $request->user()->fill([
            'token' => Crypt::encryptString($request->token),
        ])->save();

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

38.3.1.2 解密一個值

你可以使用 `Crypt` 門面提供的 `decryptString` 來進行解密。如果該值不能被正確解密，例如消息認證碼

(MAC) 無效，會拋出異常 `Illuminate\Contracts\Encryption\DecryptException`：

```
use Illuminate\Contracts\Encryption\DecryptException;
use Illuminate\Support\Facades\Crypt;

try {
    $decrypted = Crypt::decryptString($encryptedValue);
} catch (DecryptException $e) {
    // ...
}
```

39 雜湊

39.1 介紹

Laravel `Hash` Facad 為儲存使用者密碼提供了安全的 Bcrypt 和 Argon2 雜湊。如果您使用的是一個 Laravel 應用程式啟動套件，那麼在默認情況下，Bcrypt 將用於註冊和身份驗證。

Bcrypt 是雜湊密碼的絕佳選擇，因為它的「加密係數」是可調節的，這意味著隨著硬體功率的增加，生成雜湊的時間可以增加。當雜湊密碼時，越慢越好。演算法花費的時間越長，惡意使用者生成「彩虹表」的時間就越長，該表包含所有可能的字串雜湊值，這些雜湊值可能會被用於針對應用程式的暴力攻擊中。

39.2 組態

你可以在 `config/hashing.php` 組態檔案中組態默認雜湊驅動程式。目前有幾個受支援的驅動程式：Bcrypt 和 Argon2（Argon2i 和 Argon2id 變體）。

39.3 基本用法

39.3.1 雜湊密碼

您可以通過在 `Hash` Facade 上呼叫 `make` 方法來雜湊密碼：

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;

class PasswordController extends Controller
{
    /**
     * 更新使用者的密碼。
     */
    public function update(Request $request): RedirectResponse
    {
        // 驗證新密碼的長度...

        $request->user()->fill([
            'password' => Hash::make($request->newPassword)
        ])->save();

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

39.3.1.1 調整 Bcrypt 加密係數

如果您正在使用 Bcrypt 演算法，則 `make` 方法允許您使用 `rounds` 選項來組態該演算法的加密係數。然而，對

大多數應用程式來說，預設值就足夠了：

```
$hashed = Hash::make('password', [
    'rounds' => 12,
]);
```

39.3.1.2 調整 Argon2 加密係數

如果您正在使用 Argon2 演算法，則 `make` 方法允許您使用 `memory`，`time` 和 `threads` 選項來組態該演算法的加密係數。然後，對大多數應用程式來說，預設值就足夠了：

```
$hashed = Hash::make('password', [
    'memory' => 1024,
    'time' => 2,
    'threads' => 2,
]);
```

> **注意** 有關這些選項的更多資訊，請參見 關於 Argon 雜湊的官方 PHP 文件 。

39.3.2 驗證密碼是否與雜湊值相匹配

由 `Hash` Facade 提供的 `check` 方法允許您驗證給定的明文字串是否與給定的雜湊值一致：

```
if (Hash::check('plain-text', $hashedPassword)) {
    // The passwords match...
}
```

39.3.3 確定密碼是否需要重新雜湊

由 `Hash` Facade 提供的 `needsRehash` 方法可以為你檢查當雜湊 / 雜湊的加密係數改變時，你的密碼是否被新的加密係數重新加密過。某些應用程式選擇在身份驗證過程中執行此檢查：

```
if (Hash::needsRehash($hashed)) {
    $hashed = Hash::make('plain-text');
}
```

40 重設密碼

40.1 介紹

大多數 Web 應用程式都提供了一種讓使用者重設密碼的方法。Laravel 已經提供了便捷的服務來傳送密碼重設連結和安全重設密碼，而不需要您為每個應用程式重新實現此功能。

> **注意** 想要快速入門嗎？在全新的 Laravel 應用程式中安裝 Laravel 入門套件。 Laravel 的起始包將為您的整個身份驗證系統包括重設忘記的密碼提供支援。

40.1.1 模型準備

在使用 Laravel 的密碼重設功能之前，您的應用程式的 `App\Models\User` 模型必須使用 `Illuminate\Notifications\Notifiable` trait。通常，在新建立的 Laravel 應用程式的 `App\Models\User` 模型中默認引入了該 trait 。

接下來，驗證您的 `App\Models\User` 模型是否實現了 `Illuminate\Contracts\Auth\CanResetPassword` 契約。框架中包含的 `App\Models\User` 模型已經實現了該介面，並使用 `Illuminate\Auth\Passwords\CanResetPassword` 特性來包括實現該介面所需的方法。

40.1.2 資料庫準備

必須建立一個表來儲存您的應用程式的密碼重設令牌。這個表的遷移被包含在默認的 Laravel 應用程式中，所以您只需要遷移您的資料庫來建立這個表：

```
php artisan migrate
```

40.1.3 組態受信任的主機

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

通常，您應該將 Web 伺服器（例如 Nginx 或 Apache）組態為僅向您的應用程式傳送與給定主機名匹配的請求。然而，如果你沒有能力直接自訂你的 web 伺服器並且需要指示 Laravel 只響應某些主機名，你可以通過為你的應用程式啟用 `App\Http\Middleware\TrustHosts` 中介軟體來實現。當您的應用程式提供密碼重設功能時，這一點尤其重要.

要瞭解有關此中介軟體的更多資訊，請參閱 `TrustHosts` 中介軟體文件。

40.2 路由

要正確實現支援允許使用者重設其密碼的功能，我們需要定義多個路由。首先，我們需要一對路由來處理允許使用者通過其電子郵件地址請求密碼重設連結。其次，一旦使用者訪問通過電子郵件傳送給他們的密碼重設連結並完成密碼重設表單，我們將需要一對路由來處理實際重設密碼。

40.2.1 請求密碼重設連結

40.2.1.1 密碼重設連結申請表

首先，我們將定義請求密碼重設連結所需的路由。首先，我們將定義一個路由，該路由返回一個帶有密碼重設連結請求表單的檢視：

```
Route::get('/forgot-password', function () {
    return view('auth.forgot-password');
})->middleware('guest')->name('password.request');
```

此路由返回的檢視應該有一個包含 `email` 欄位的表單，該欄位允許使用者請求給定電子郵件地址的密碼重設連結。

40.2.1.2 處理表單提交

接下來，我們將定義一個路由，該路由將從「忘記密碼」檢視處理表單提交請求。此路由將負責驗證電子郵件地址並將密碼重設請求傳送給相應使用者：

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Password;

Route::post('/forgot-password', function (Request $request) {
    $request->validate(['email' => 'required|email']);

    $status = Password::sendResetLink(
        $request->only('email')
    );

    return $status === Password::RESET_LINK_SENT
                ? back()->with(['status' => __($status)])
                : back()->withErrors(['email' => __($status)]);
})->middleware('guest')->name('password.email');
```

在繼續之前，讓我們更詳細地檢查一下這條路由。首先，驗證請求的 `email` 屬性。接下來，我們將使用 Laravel 內建的 `Password` 門面向使用者傳送一個密碼重設連結。密碼代理將負責按給定欄位（在本例中是電子郵件地址）檢索使用者，並通過 Laravel 的內建 消息通知系統 向使用者傳送密碼重設連結。

該 `sendResetLink` 方法返回一個狀態標識。可以使用 Laravel 的 本地化 助手來轉換此狀態，以便向使用者顯示有關請求狀態的使用者友好提示。密碼重設狀態的轉換由應用程式的 `lang/{lang}/passwords.php` 語言檔案決定。狀態 slug 的每個可能值的條目位於 `passwords` 語言檔案中。

> **注意** 默認情況下, Laravel 應用程式的框架不包括 `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.

你可能想知道，Laravel 在呼叫 Password 門面的 sendResetLink 方法時，Laravel 怎麼知道如何從應用程式資料庫中檢索使用者記錄。Laravel 密碼代理利用身份驗證系統的「使用者提供者」來檢索資料庫記錄。密碼代理使用的使用者提供程序是在 passwords 組態檔案的 config/auth.php 組態陣列中組態的。要瞭解有關編寫自訂使用者提供程序的更多資訊，請參閱 身份驗證文件。

> **Note**
> 技巧：當手動實現密碼重設時，你需要自己定義檢視和路由的內容。如果你想要包含所有必要的身份驗證和驗證邏輯的腳手架，請查看 Laravel 應用程式入門工具包。

40.2.2 重設密碼

40.2.2.1 重設密碼表單

接下來，我們將定義使用者點選重設密碼郵件中的連結，進行重設密碼所需要的一些路由。第一步，先定義一個獲取重設密碼表單的路由。這個路由需要一個 `token` 來驗證請求：

```
Route::get('/reset-password/{token}', function (string $token) {
    return view('auth.reset-password', ['token' => $token]);
})->middleware('guest')->name('password.reset');
```

通過路由返回的檢視應該顯示一個含有 `email` 欄位， `password` 欄位，`password_confirmation` 欄位和一個隱藏的值通過路由參數獲取的 `token` 欄位。

40.2.2.2 處理表單提交的資料

當然，我們需要定義一個路由來接受表單提交的資料。這個路由會檢查傳過來的參數並更新資料庫中使用者的密碼：

```
use App\Models\User;
use Illuminate\Auth\Events\PasswordReset;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Facades\Password;
use Illuminate\Support\Str;

Route::post('/reset-password', function (Request $request) {
    $request->validate([
        'token' => 'required',
        'email' => 'required|email',
        'password' => 'required|min:8|confirmed',
    ]);

    $status = Password::reset(
        $request->only('email', 'password', 'password_confirmation', 'token'),
        function (User $user, string $password) {
            $user->forceFill([
                'password' => Hash::make($password)
            ])->setRememberToken(Str::random(60));

            $user->save();

            event(new PasswordReset($user));
        }
    );

    return $status === Password::PASSWORD_RESET
                ? redirect()->route('login')->with('status', __($status))
                : back()->withErrors(['email' => [__($status)]]);
})->middleware('guest')->name('password.update');
```

在繼續之前，我們再詳細地檢查下這條路由。 首先，驗證請求的 `token`，`email` 和 `password` 屬性。 接下來，我們將使用 Laravel 的內建「密碼代理」 (通過 `Password` facade) 來驗證密碼重設請求憑據。

如果提供給密碼代理的令牌、電子郵件地址和密碼有效，則將呼叫傳遞給 `reset` 方法的閉包。 在這個接收使用者實例和純文字密碼的閉包中，我們可以更新資料庫中使用者的密碼。

該 `reset` 方法返回一個「狀態」標識。 此狀態可以使用 Laravel 的 本地化 助手來翻譯此狀態，以便向使用者顯示有關其請求狀態的使用者友好消息。密碼重設狀態的翻譯由應用程式的 `lang/{lang}/passwords.php` 語言檔案決定。狀態段的每個可能值的條目位於 `passwords` 語言檔案中。如果你的應用沒有 `lang` 資料夾，你可以使用 `lang:publish` artisan 命令來建立。

在繼續之前，你可能想知道 Laravel 如何在呼叫 `Password` facade 的 `reset` 方法時如何知道如何從應用程式的資料庫中檢索使用者記錄。Laravel 密碼代理利用你的身份驗證系統的「使用者提供者」來檢索資料庫記錄。密碼代理使用的使用者提供程序在組態檔案的 `config/auth.php` 組態檔案的 `passwords` 組態陣列中組態。要瞭解有關編寫自訂使用者提供程序的更多資訊，請參閱 身份驗證文件。

40.3 刪除過期令牌

已過期的密碼重設令牌仍將存在於你的資料庫中。然而，你可以使用 `auth:clear-resets` Artisan 命令輕鬆刪除這些記錄：

```
php artisan auth:clear-resets
```

如果你想使該過程自動化，請考慮將命令新增到應用程式的 調度程序：

```
$schedule->command('auth:clear-resets')->everyFifteenMinutes();
```

40.4 自訂

40.4.1.1 重設連結自訂

你可以使用 `ResetPassword` 通知類提供的 `createUrlUsing` 方法自訂密碼重設連結 URL。 此方法接受一個閉包，該閉包接收正在接收通知的使用者實例以及密碼重設連結令牌。 通常，你應該從 `App\Providers\AuthServiceProvider` 服務提供者的 `boot` 方法中呼叫此方法：

```
use App\Models\User;
use Illuminate\Auth\Notifications\ResetPassword;

/**
 * 註冊任何身份驗證/授權服務
 */
public function boot(): void
{
    ResetPassword::createUrlUsing(function (User $user, string $token) {
        return 'https://example.com/reset-password?token='.$token;
    });
}
```

40.4.1.2 重設郵件自訂

你可以輕鬆修改用於向使用者傳送密碼重設連結的通知類。 首先，覆蓋你的 `App\Models\User` 模型上的 `sendPasswordResetNotification` 方法。 在此方法中，你可以使用你自己建立的任何 通知類 傳送通知。 密碼重設 `$token` 是該方法收到的第一個參數。 你可以使用這個 `$token` 來建構你選擇的密碼重設 URL 並將你的通知傳送給使用者：

```
use App\Notifications\ResetPasswordNotification;

/**
 * 傳送密碼重設通知給使用者
 *
 * @param string $token
 */
public function sendPasswordResetNotification($token): void
{
    $url = 'https://example.com/reset-password?token='.$token;

    $this->notify(new ResetPasswordNotification($url));
}
```

41 資料庫快速入門

41.1 簡介

幾乎所有的應用程式都需要和資料庫進行互動。Laravel 為此提供了一套非常簡單易用的資料庫互動方式。開發者可以使用原生 SQL，查詢構造器，以及 Eloquent ORM 等方式與資料庫互動。目前，Laravel 為以下五種資料庫提供了官方支援：:

- MariaDB 10.3+ (版本策略)
- MySQL 5.7+ (版本策略)
- PostgreSQL 10.0+ (版本策略)
- SQLite 3.8.8+
- SQL Server 2017+ (版本策略)

41.1.1 組態

Laravel 資料庫服務的組態位於應用程式的 `config/database.php` 組態檔案中。在此檔案中，您可以定義所有資料庫連接，並指定默認情況下應使用的連接。此檔案中的大多數組態選項由應用程式環境變數的值驅動。本檔案提供了 Laravel 支援的大多數資料庫系統的示例。

在默認情況下，Laravel 的示例 環境組態 使用了 Laravel Sail，Laravel Sail 是一種用於在本地開發 Laravel 應用的 Docker 組態。但你依然可以根據本地資料庫的需要修改資料庫組態。

41.1.1.1 SQLite 組態

SQLite 資料庫本質上只是一個存在你檔案系統上的檔案。你可以通過 `touch` 命令來建立一個新的 SQLite 資料庫，如： `touch database/database.sqlite`. 建立資料庫之後，你就可以很簡單地使用資料庫的絕對路徑來組態 `DB_DATABASE` 環境變數，使其指向這個新建立的資料庫：

```
DB_CONNECTION=sqlite
DB_DATABASE=/absolute/path/to/database.sqlite
```

若要為 SQLite 連接啟用外部索引鍵約束，應將 `DB_FOREIGN_KEYS` 環境變數設定為 `true` ：

```
DB_FOREIGN_KEYS=true
```

41.1.1.2 Microsoft SQL Server 組態

在使用 SQL Server 資料庫前，你需要先確保你已安裝並啟用了 `sqlsrv` 和 `pdo_sqlsrv` PHP 擴展以及它們所需要的依賴項，例如 Microsoft SQL ODBC 驅動。

41.1.1.3 URL 形式組態

通常，資料庫連接使用多個組態項進行組態，例如 `host` 、`database` 、 `username` 、 `password` 等。這些組態項都擁有對應的環境變數。這意味著你需要在生產伺服器上管理多個不同的環境變數。

部分資料庫託管平台（如 AWS 和 Heroku）會提供了包含所有連接資訊的資料庫「URL」。它們通常看起來像這樣：

```
mysql://root:password@127.0.0.1/forge?charset=UTF-8
```

這些 URL 通常遵循標準模式約定：

```
driver://username:password@host:port/database?options
```

為了方便起見，Laravel 支援使用這些 URL 替代傳統的組態項來組態你的資料庫。如果組態項 `url`（或其對應的環境變數 `DATABASE_URL`）存在，那麼 Laravel 將會嘗試從 URL 中提取資料庫連接以及憑證資訊。

41.1.2 讀寫分離

有時候你可能會希望使用一個資料庫連接來執行 `SELECT` 語句，而 `INSERT`、`UPDATE` 和 `DELETE` 語句則由另一個資料庫連接來執行。在 Laravel 中，無論你是使用原生 SQL 查詢、查詢構造器 或是 `Eloquent ORM`，都能輕鬆實現讀寫分離。

為了弄明白如何組態讀寫分離，我們先來看個例子：

```
'mysql' => [
    'read' => [
        'host' => [
            '192.168.1.1',
            '196.168.1.2',
        ],
    ],
    'write' => [
        'host' => [
            '196.168.1.3',
        ],
    ],
    'sticky' => true,
    'driver' => 'mysql',
    'database' => 'database',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8mb4',
    'collation' => 'utf8mb4_unicode_ci',
    'prefix' => '',
],
```

請注意，我們在資料庫組態中加入了三個鍵，分別是： `read`, `write` 以及 `sticky` 。`read` 和 `write` 的值是一個只包含 `host` 鍵的陣列。這代表其他的資料庫選項將會從主 `mysql` 組態中獲取。

如果你想要覆寫主 `mysql` 組態，只需要將需要覆寫的值放到 `read` 和 `write` 陣列裡即可。所以，在這個例子中，`192.168.1.1` 將會被用作「讀」連接主機，而 `192.168.1.3` 將作為「寫」連接主機。這兩個連接將共享 `mysql` 陣列中的各項組態，如資料庫憑證（使用者名稱、密碼）、前綴、字元編碼等。如果 `host` 陣列中存在多個值，`Laravel` 將會為每個連接隨機選取所使用的資料庫主機。

41.1.2.1 `sticky` 選項

`sticky` 是一個 可選 值，它用於允許 Laravel 立即讀取在當前請求週期內寫入到資料庫的記錄。若 `sticky` 選項被啟用，且在當前請求週期中執行過「寫」操作，那麼在這之後的所有「讀」操作都將使用「寫」連接。這樣可以確保同一個請求週期中寫入的資料庫可以被立即讀取到，從而避免主從同步延遲導致的資料不一致。不過是否啟用它取決於項目的實際需求。

41.2 執行原生 SQL 查詢

一旦組態好資料庫連接，你就可以使用 DB Facade 來執行查詢。DB Facade 為每種類型的查詢都提供了相應的方法：`select`、`update`、`insert`、`delete` 以及 `statement`。

41.2.1.1 執行 SELECT 查詢

你可以使用 DB Facade 的 `select` 方法來執行一個基礎的 SELECT 查詢：

```
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller
{
    /**
     * 展示應用程式所有的使用者列表.
     */
    public function index(): View
    {
        $users = DB::select('select * from users where active = ?', [1]);

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

傳遞給 `select` 方法的第一個參數是一個原生 SQL 查詢語句，而第二個參數則是需要繫結到查詢中的參數值。通常，這些值用於約束 `where` 語句。使用參數繫結可以有效防止 SQL 隱碼攻擊。

`select` 方法將始終返回一個包含查詢結果的陣列。陣列中的每個結果都對應一個資料庫記錄的 `stdClass` 對象：

```
use Illuminate\Support\Facades\DB;

$users = DB::select('select * from users');

foreach ($users as $user) {
    echo $user->name;
}
```

41.2.1.2 選擇標量值

有時你的資料庫查詢可能得到一個單一的標量值。而不是需要從記錄對象中檢索查詢的標量結果，Laravel 允許你直接使用 `scalar` 方法檢索此值:

```
$burgers = DB::scalar(
    "select count(case when food = 'burger' then 1 end) as burgers from menu"
);
```

41.2.1.3 使用命名繫結

除了使用?表示參數繫結外，你還可以使用命名繫結的形式來執行一個查詢：

```
$results = DB::select('select * from users where id = :id', ['id' => 1]);
```

41.2.1.4 執行 Insert 語句

你可以使用 DB Facade 的 `insert` 方法來執行語句。跟 `select` 方法一樣，該方法的第一個和第二個參數分別是原生 SQL 語句和繫結的資料：

```
use Illuminate\Support\Facades\DB;

DB::insert('insert into users (id, name) values (?, ?)', [1, 'Marc']);
```

41.2.1.5 執行 Update 語句

`update` 方法用於更新資料庫中現有的記錄。該方法將會返回受到本次操作影響的記錄行數：

```
use Illuminate\Support\Facades\DB;

$affected = DB::update(
    'update users set votes = 100 where name = ?',
    ['Anita']
);
```

41.2.1.6 執行 Delete 語句

`delete` 函數被用於刪除資料庫中的記錄。它的返回值與 `update` 函數相同，返回本次操作受影響的總行數。

```
use Illuminate\Support\Facades\DB;

$deleted = DB::delete('delete from users');
```

41.2.1.7 執行指定的 SQL

部分 SQL 語句不返回任何值。在這種情況下，你可能需要使用 `DB::statement($sql)` 來執行你的 SQL 語句。

```
DB::statement('drop table users');
```

41.2.1.8 直接執行 SQL

有時候你可能想執行一段 SQL 語句，但不需要進行 SQL 預處理繫結。這種情況下你可以使用 `DB::unprepared($sql)` 來執行你的 SQL 語句。

```
DB::unprepared('update users set votes = 100 where name = "Dries"');
```

> **注意**
> 未經過預處理 SQL 的語句不繫結參數，它們可能容易受到 SQL 隱碼攻擊的攻擊。在沒有必要的理由的情況下，你不應直接在 SQL 中使用使用者傳入的資料。

41.2.1.9 在事務中的隱式提交

在事務中使用 `DB::statement($sql)` 與 `DB::unprepared($sql)` 時，你必須要謹慎處理，避免 SQL 語句產生隱式提交。這些語句會導致資料庫引擎間接地提交整個事務, 讓 Laravel 丟失資料庫當前的事務等級。下面是一個會產生隱式提交的示例 SQL：建立一個資料庫表。

```
DB::unprepared('create table a (col varchar(1) null)');
```

請參考 MySQL 官方手冊以瞭解更多隱式提交的資訊。

41.2.2 使用多資料庫連接

如果你在組態檔案 `config/database.php` 中定義了多個資料庫連接的話，你可以通過 DB Facade 的 `connection` 方法來使用它們。傳遞給 `connection` 方法的連接名稱應該是你在 `config/database.php` 裡或者通過 `config` 助手函數在執行階段組態的連接之一：

```
use Illuminate\Support\Facades\DB;

$users = DB::connection('sqlite')->select(/* ... */);
```

你也可以使用一個連接實例上的 `getPdo` 方法來獲取底層的 PDO 實例：

```
$pdo = DB::connection()->getPdo();
```

41.2.3 監聽查詢事件

如果你想要獲取程序執行的每一條 SQL 語句，可以使用 `Db` facade 的 `listen` 方法。該方法對查詢日誌和偵錯非常有用，你可以在 服務提供者 中使用 `boot` 方法註冊查詢監聽器。

```
<?php

namespace App\Providers;

use Illuminate\Database\Events\QueryExecuted;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * 註冊任意應用服務
     */
    public function register(): void
    {
        // ...
    }

    /**
     * 引導任意應用服務
     */
    public function boot(): void
    {
        DB::listen(function (QueryExecuted $query) {
            // $query->sql;
            // $query->bindings;
            // $query->time;
        });
    }
}
```

41.2.4 監控累積查詢時間

現代 web 應用程式的一個常見性能瓶頸是查詢資料庫所花費的時間。幸運的是，當 Laravel 在單個請求中花費了太多時間查詢資料庫時，它可以呼叫你定義的閉包或回呼。要使用它，你可以呼叫 `whenQueryingForLongerThan` 方法並提供查詢時間閥值(以毫秒為單位)和一個閉包作為參數。你可以在服務提供者 的 `boot` 方法中呼叫此方法：

```
<?php

namespace App\Providers;

use Illuminate\Database\Connection;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\ServiceProvider;
use Illuminate\Database\Events\QueryExecuted;

class AppServiceProvider extends ServiceProvider
{
    /**
     * 註冊任意應用服務
     */
    public function register(): void
    {
        // ...
    }

    /**
```

```
     * 引導任意應用服務
     */
    public function boot(): void
    {
        DB::whenQueryingForLongerThan(500, function (Connection $connection,
QueryExecuted $event) {
            // 通知開發團隊...
        });
    }
}
```

41.3 資料庫事務

想要在資料庫事務中運行一系列操作，你可以使用 DB 門面的 transaction 方法。如果在事務的閉包中出現了異常，事務將會自動回滾。如果閉包執行成功，事務將會自動提交。在使用 transaction 方法時不需要手動回滾或提交：

```
use Illuminate\Support\Facades\DB;

DB::transaction(function () {
    DB::update('update users set votes = 1');

    DB::delete('delete from posts');
});
```

41.3.1.1 處理死鎖

transaction 方法接受一個可選的第二個參數，該參數定義發生死鎖時事務應重試的次數。一旦這些嘗試用盡，就會拋出一個異常：

```
use Illuminate\Support\Facades\DB;

DB::transaction(function () {
    DB::update('update users set votes = 1');

    DB::delete('delete from posts');
}, 5);
```

41.3.1.2 手動執行事務

如果你想要手動處理事務並完全控制回滾和提交，可以使用 DB 門面提供的 beginTransaction 方法：

```
use Illuminate\Support\Facades\DB;

DB::beginTransaction();
```

你可以通過 rollBack 方法回滾事務：

```
DB::rollBack();
```

最後，你可以通過 commit 方法提交事務：

```
DB::commit();
```

> **技巧**
> DB 門面的事務方法還可以用於控制 查詢構造器 and Eloquent ORM.

41.4 連接到資料庫 CLI

如果你想連接到資料庫的 CLI，則可以使用 db Artisan 命令：

```
php artisan db
```

如果需要，你可以指定資料庫連接名稱以連接到不是默認連接的資料庫連接：

```
php artisan db mysql
```

41.5 檢查你的資料庫

使用 db:show 和 db:table Artisan 命令，你可以深入瞭解資料庫及其相關的表。要查看資料庫的概述，包括它的大小、類型、打開的連線以及表的摘要，你可以使用 db:show 命令：

```
php artisan db:show
```

你可以通過 --database 選項向命令提供資料庫連接名稱來指定應該檢查哪個資料庫連接:

```
php artisan db:show --database=pgsql
```

如果希望在命令的輸出中包含表行計數和資料庫檢視詳細資訊，你可以分別提供 --counts 和 --views 選項。在大型資料庫上，檢索行數和檢視詳細資訊可能很慢:

```
php artisan db:show --counts --views
```

41.5.1.1 表的摘要資訊

如果你想獲得資料庫中單張表的概覽，你可以執行 db:table Artisan 命令。這個命令提供了一個資料庫表的概覽，包括它的列、類型、屬性、鍵和索引:

```
php artisan db:table users
```

41.6 監視資料庫

使用 db:monitor Artisan 命令，如果你的資料庫正在管理超過指定數量的打開連接，可以通過 Laravel 調度觸發 Illuminate\Database\Events\DatabaseBusy 事件。

開始, 你應該將 db:monitor 命令安排為 每分鐘運行一次。 該命令接受要監視的資料庫連接組態的名稱，以及在分派事件之前應允許的最大打開連線：

```
php artisan db:monitor --databases=mysql,pgsql --max=100
```

僅調度此命令不足以觸發通知，提醒你打開的連線。當命令遇到打開連接計數超過閾值的資料庫時，將調度 DatabaseBusy 事件。你應該在應用程式的 EventServiceProvider 中偵聽此事件，以便向你或你的開發團隊傳送通知

```
use App\Notifications\DatabaseApproachingMaxConnections;
use Illuminate\Database\Events\DatabaseBusy;
use Illuminate\Support\Facades\Event;
use Illuminate\Support\Facades\Notification;

/**
 * 為應用程式註冊任何其他事件。
 */
public function boot(): void
{
    Event::listen(function (DatabaseBusy $event) {
        Notification::route('mail', 'dev@example.com')
                ->notify(new DatabaseApproachingMaxConnections(
                    $event->connectionName,
                    $event->connections
                ));
    });
}
```

42 查詢生成器

42.1 介紹

Laravel 的資料庫查詢生成器提供了一種便捷、流暢的介面來建立和運行資料庫查詢。它可用於執行應用程式中的大多數資料庫操作，並與 Laravel 支援的所有資料庫系統完美配合使用。

Laravel 查詢生成器使用 PDO 參數繫結來保護你的應用程式免受 SQL 隱碼攻擊。無需清理或淨化傳遞給查詢生成器的字串作為查詢繫結。

> **警告** PDO 不支援繫結列名。因此，你不應該允許使用者輸入來決定查詢引用的列名，包括 「order by」列名。

42.2 運行資料庫查詢

42.2.1.1 從表中檢索所有行

你可以使用 `DB` facade 提供的 `table` 方法開始查詢。table 方法為指定的表返回一個鏈式查詢構造器實例，允許在查詢上連結更多約束，最後使用 `get` 方法檢索查詢結果：

```
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller
{
    /**
     * 展示應用程式所有使用者的列表
     */
    public function index(): View
    {
        $users = DB::table('users')->get();

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

`get` 方法返回包含查詢結果的 `Illuminate\Support\Collection` 實例，每個結果都是 PHP `stdClass` 實例。可以將列作為對象的屬性來訪問每列的值：

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')->get();

foreach ($users as $user) {
    echo $user->name;
}
```

> **技巧:**
> Laravel 集合提供了各種及其強大的方法來對應和裁剪資料。有關 Laravel 集合的更多資訊，請查看 集合文件.

42.2.1.2 從表中檢索單行或單列

如果只需要從資料表中檢索單行，可以使用 DB facade 中的 first 方法。 此方法將返回單個 stdClass 對象

```
$user = DB::table('users')->where('name', 'John')->first();

return $user->email;
```

如果不想要整行，可以使用 value 方法從紀錄中提取單個值。此方法將直接返回列的值：

```
$email = DB::table('users')->where('name', 'John')->value('email');
```

如果要通過 id 欄位值獲取單行資料，可以使用 find 方法：

```
$user = DB::table('users')->find(3);
```

42.2.1.3 獲取某一列的值

如果要獲取包含單列值的 Illuminate\Support\Collection 實例，則可以使用 pluck 方法。在下面的例子中，我們將獲取角色表中標題的集合：

```
use Illuminate\Support\Facades\DB;

$titles = DB::table('users')->pluck('title');

foreach ($titles as $title) {
    echo $title;
}
```

你可以通過向 pluck 方法提供第二個參數來指定結果集中要作為鍵的列：

```
$titles = DB::table('users')->pluck('title', 'name');

foreach ($titles as $name => $title) {
    echo $title;
}
```

42.2.2 分塊結果

如果需要處理成千上萬的資料庫記錄，請考慮使用 DB 提供的 chunk 方法。這個方法一次檢索一小塊結果，並將每個塊反饋到閉包涵數中進行處理。例如，讓我們以一次 100 條記錄的塊為單位檢索整個 users 表：

```
use Illuminate\Support\Collection;
use Illuminate\Support\Facades\DB;

DB::table('users')->orderBy('id')->chunk(100, function (Collection $users) {
    foreach ($users as $user) {
        // ...
    }
});
```

你可以通過從閉包中返回 false 來停止處理其餘的塊:

```
DB::table('users')->orderBy('id')->chunk(100, function (Collection $users) {
    // 處理分塊...

    return false;
});
```

如果在對結果進行分塊時更新資料庫記錄，那分塊結果可能會以意想不到的方式更改。如果你打算在分塊時更新檢索到的記錄，最好使用 chunkById 方法。此方法將根據記錄的主鍵自動對結果進行分頁:

```
DB::table('users')->where('active', false)
    ->chunkById(100, function (Collection $users) {
        foreach ($users as $user) {
            DB::table('users')
                ->where('id', $user->id)
```

```
                    ->update(['active' => true]);
            }
        });
```

注意
當在更新或刪除塊回呼中的記錄時，對主鍵或外部索引鍵的任何更改都可能影響塊查詢。這可能會導致記錄未包含在分塊結果中。

42.2.3 Lazily 流式傳輸結果

lazy 方法的工作方式類似於 chunk 方法，因為它以塊的形式執行查詢。但是，lazy() 方法不是將每個塊傳遞給回呼，而是返回一個 LazyCollection，它可以讓你與結果進行互動單個流：

```
use Illuminate\Support\Facades\DB;

DB::table('users')->orderBy('id')->lazy()->each(function (object $user) {
    // ...
});
```

再一次，如果你打算在迭代它們時更新檢索到的記錄，最好使用 lazyById 或 lazyByIdDesc 方法。 這些方法將根據記錄的主鍵自動對結果進行分頁：

```
DB::table('users')->where('active', false)
    ->lazyById()->each(function (object $user) {
        DB::table('users')
            ->where('id', $user->id)
            ->update(['active' => true]);
    });
```

注意
在迭代記錄時更新或刪除記錄時，對主鍵或外部索引鍵的任何更改都可能影響塊查詢。這可能會導致記錄不包含在結果中。

42.2.4 聚合函數

查詢建構器還提供了多種檢索聚合值的方法，例如 count， max， min，avg 和 sum。你可以在建構查詢後呼叫這些方法中的任何一個：

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')->count();

$price = DB::table('orders')->max('price');
```

當然，你可以將這些方法與其他子句結合起來，以最佳化計算聚合值的方式：

```
$price = DB::table('orders')
                ->where('finalized', 1)
                ->avg('price');
```

42.2.4.1 判斷記錄是否存在

除了通過 count 方法可以確定查詢條件的結果是否存在之外，還可以使用 exists 和 doesntExist 方法：

```
if (DB::table('orders')->where('finalized', 1)->exists()) {
    // ...
}

if (DB::table('orders')->where('finalized', 1)->doesntExist()) {
    // ...
}
```

42.3 Select 語句

42.3.1.1 指定一個 Select 語句

可能你並不總是希望從資料庫表中獲取所有列。 使用 `select` 方法，可以自訂一個 「select」 查詢語句來查詢指定的欄位：

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')
             ->select('name', 'email as user_email')
             ->get();
```

`distinct` 方法會強制讓查詢返回的結果不重複：

```
$users = DB::table('users')->distinct()->get();
```

如果你已經有了一個查詢構造器實例，並且希望在現有的查詢語句中加入一個欄位，那麼你可以使用 `addSelect` 方法：

```
$query = DB::table('users')->select('name');

$users = $query->addSelect('age')->get();
```

42.4 原生表示式

當你需要在查詢中插入任意的字串時，你可以使用 `DB` 門面提供的 `raw` 方法以建立原生表示式。

```
$users = DB::table('users')
              ->select(DB::raw('count(*) as user_count, status'))
              ->where('status', '<>', 1)
              ->groupBy('status')
              ->get();
```

> **警告**
> 原生語句作為字串注入到查詢中，因此必須格外小心避免產生 SQL 隱碼攻擊漏洞。

42.4.1 原生方法。

可以使用以下方法代替 `DB::raw`，將原生表示式插入查詢的各個部分。**請記住，Laravel 無法保證所有使用原生表示式的查詢都不受到 SQL 隱碼攻擊漏洞的影響**。

42.4.1.1 `selectRaw`

`selectRaw` 方法可以用來代替 `addSelect(DB::raw(/* ... */))`。此方法接受一個可選的繫結陣列作為其第二個參數：

```
$orders = DB::table('orders')
                ->selectRaw('price * ? as price_with_tax', [1.0825])
                ->get();
```

42.4.1.2 `whereRaw / orWhereRaw`

`whereRaw` 和 `orWhereRaw` 方法可用於將原始 「where」 子句注入你的查詢。這些方法接受一個可選的繫結陣列作為它們的第二個參數：

```
$orders = DB::table('orders')
                ->whereRaw('price > IF(state = "TX", ?, 100)', [200])
```

```
                ->get();
```

42.4.1.3 havingRaw / orHavingRaw

havingRaw 和 orHavingRaw 方法可用於提供原始字串作為 「having」子句的值。這些方法接受一個可選的繫結陣列作為它們的第二個參數：

```
$orders = DB::table('orders')
                ->select('department', DB::raw('SUM(price) as total_sales'))
                ->groupBy('department')
                ->havingRaw('SUM(price) > ?', [2500])
                ->get();
```

42.4.1.4 orderByRaw

orderByRaw 方法可用於將原生字串設定為「order by」子句的值：

```
$orders = DB::table('orders')
                ->orderByRaw('updated_at - created_at DESC')
                ->get();
```

42.4.2 groupByRaw

groupByRaw 方法可以用於將原生字串設定為 group by 子句的值：

```
$orders = DB::table('orders')
                ->select('city', 'state')
                ->groupByRaw('city, state')
                ->get();
```

42.5 Joins

42.5.1.1 Inner Join 語句

查詢構造器也還可用於向查詢中新增連接子句。若要執行基本的「inner join」，你可以對查詢構造器實例使用 join 方法。傳遞給 join 方法的第一個參數是需要你連接到的表的名稱，而其餘參數指定連接的列約束。你甚至還可以在一個查詢中連接多個表：

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')
            ->join('contacts', 'users.id', '=', 'contacts.user_id')
            ->join('orders', 'users.id', '=', 'orders.user_id')
            ->select('users.*', 'contacts.phone', 'orders.price')
            ->get();
```

42.5.1.2 Left Join / Right Join 語句

如果你想使用「left join」或者「right join」代替「inner join」，可以使用 leftJoin 或者 rightJoin 方法。這兩個方法與 join 方法用法相同：

```
$users = DB::table('users')
            ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
            ->get();

$users = DB::table('users')
            ->rightJoin('posts', 'users.id', '=', 'posts.user_id')
            ->get();
```

42.5.1.3 Cross Join 語句

你可以使用 crossJoin 方法執行「交叉連接」。交叉連接在第一個表和被連接的表之間會生成笛卡爾積：

```
$sizes = DB::table('sizes')
             ->crossJoin('colors')
             ->get();
```

42.5.1.4 高級 Join 語句

你還可以指定更高級的聯接子句。首先，將閉包作為第二個參數傳遞給 join 方法。閉包將收到一個 Illuminate\Database\Query\JoinClause 實例，該實例允許你指定對 join 子句的約束：

```
DB::table('users')
         ->join('contacts', function (JoinClause $join) {
             $join->on('users.id', '=', 'contacts.user_id')->orOn(/* ... */);
         })
         ->get();
```

如果你想要在連接上使用「where」風格的語句，你可以在連接上使用 JoinClause 實例中的 where 和 orWhere 方法。這些方法會將列和值進行比較，而不是列和列進行比較：

```
DB::table('users')
         ->join('contacts', function (JoinClause $join) {
             $join->on('users.id', '=', 'contacts.user_id')
                  ->where('contacts.user_id', '>', 5);
         })
         ->get();
```

42.5.1.5 子連接查詢

你可以使用 joinSub，leftJoinSub 和 rightJoinSub 方法關聯一個查詢作為子查詢。他們每一種方法都會接收三個參數：子查詢、表別名和定義關聯欄位的閉包。如下面這個例子，獲取含有使用者最近一次發佈部落格時的 created_at 時間戳的使用者集合：

```
$latestPosts = DB::table('posts')
                    ->select('user_id', DB::raw('MAX(created_at) as
last_post_created_at'))
                    ->where('is_published', true)
                    ->groupBy('user_id');

$users = DB::table('users')
         ->joinSub($latestPosts, 'latest_posts', function (JoinClause $join) {
             $join->on('users.id', '=', 'latest_posts.user_id');
         })->get();
```

42.6 聯合

查詢構造器還提供了一種簡潔的方式將兩個或者多個查詢「聯合」在一起。例如，你可以先建立一個查詢，然後使用 union 方法來連接更多的查詢：

```
use Illuminate\Support\Facades\DB;

$first = DB::table('users')
             ->whereNull('first_name');

$users = DB::table('users')
             ->whereNull('last_name')
             ->union($first)
             ->get();
```

查詢構造器不僅提供了 union 方法，還提供了一個 unionAll 方法。當查詢結合 unionAll 方法使用時，將

不會刪除重複的結果。unionAll 方法的用法和 union 方法一樣。

42.7 基礎的 Where 語句

42.7.1 Where 語句

你可以在 where 語句中使用查詢構造器的 where 方法。呼叫 where 方法需要三個基本參數。第一個參數是欄位的名稱。第二個參數是一個運算子，它可以是資料庫中支援的任意運算子。第三個參數是與欄位比較的值。

例如。在 users 表中查詢 votes 欄位等於 100 並且 age 欄位大於 35 的資料：

```
$users = DB::table('users')
                    ->where('votes', '=', 100)
                    ->where('age', '>', 35)
                    ->get();
```

為了方便起見。如果你想要比較一個欄位的值是否 **等於** 給定的值。你可以將這個給定的值作為第二個參數傳遞給 where 方法。那麼，Laravel 會默認使用 = 運算子：

```
$users = DB::table('users')->where('votes', 100)->get();
```

如上所述，你可以使用資料庫支援的任意運算子：

```
$users = DB::table('users')
                    ->where('votes', '>=', 100)
                    ->get();

$users = DB::table('users')
                    ->where('votes', '<>', 100)
                    ->get();

$users = DB::table('users')
                    ->where('name', 'like', 'T%')
                    ->get();
```

你也可以將一個條件陣列傳遞給 where 方法。陣列的每個元素都應該是一個陣列，其中包是傳遞給 where 方法的三個參數：

```
$users = DB::table('users')->where([
    ['status', '=', '1'],
    ['subscribed', '<>', '1'],
])->get();
```

> **注意** PDO 不支援繫結欄位名。因此，你不應該允許讓使用者輸入欄位名進行查詢引用，包括結果集「order by」語句。

42.7.2 Or Where 語句

當鏈式呼叫多個 where 方法的時候，這些「where」語句將會被看成是 and 關係。另外，你也可以在查詢語句中使用 orWhere 方法來表示 or 關係。orWhere 方法接收的參數和 where 方法接收的參數一樣：

```
$users = DB::table('users')
                        ->where('votes', '>', 100)
                        ->orWhere('name', 'John')
                        ->get();
```

如果你需要在括號內對 「or」 條件進行分組，那麼可以傳遞一個閉包作為 orWhere 方法的第一個參數：

```
$users = DB::table('users')
```

```
            ->where('votes', '>', 100)
            ->orWhere(function(Builder $query) {
                $query->where('name', 'Abigail')
                      ->where('votes', '>', 50);
            })
            ->get();
```

上面的示例將生成以下 SQL：

```
select * from users where votes > 100 or (name = 'Abigail' and votes > 50)
```

注意 為避免全域範疇應用時出現意外，你應始終對 `orWhere` 呼叫進行分組。

42.7.3 Where Not 語句

`whereNot` 和 `orWhereNot` 方法可用於否定一組給定的查詢條件。例如, 下面的查詢排除了正在清倉甩賣或價格低於 10 的產品：

```
$products = DB::table('products')
                ->whereNot(function (Builder $query) {
                    $query->where('clearance', true)
                          ->orWhere('price', '<', 10);
                })
                ->get();
```

42.7.4 JSON Where 語句

Laravel 也支援 JSON 類型的欄位查詢，前提是資料庫也支援 JSON 類型。目前，有 MySQL 5.7+、PostgreSQL、SQL Server 2016 和 SQLite 3.39.0 支援 JSON 類型 (with the JSON1 extension)。可以使用 `->` 運算子來查詢 JSON 欄位：

```
$users = DB::table('users')
                ->where('preferences->dining->meal', 'salad')
                ->get();
```

你可以使用 `whereJsonContains` 方法來查詢 JSON 陣列。但是 SQLite 資料庫版本低於 3.38.0 時不支援該功能：

```
$users = DB::table('users')
                ->whereJsonContains('options->languages', 'en')
                ->get();
```

如果你的應用使用的是 MySQL 或者 PostgreSQL 資料庫，那麼你可以向 `whereJsonContains` 方法中傳遞一個陣列類型的值：

```
$users = DB::table('users')
                ->whereJsonContains('options->languages', ['en', 'de'])
                ->get();
```

你可以使用 `whereJsonLength` 方法來查詢 JSON 陣列的長度：

```
$users = DB::table('users')
                ->whereJsonLength('options->languages', 0)
                ->get();

$users = DB::table('users')
                ->whereJsonLength('options->languages', '>', 1)
                ->get();
```

42.7.5 其他 Where 語句

whereBetween / orWhereBetween

whereBetween 方法是用來驗證欄位的值是否在給定的兩個值之間：

```
$users = DB::table('users')
            ->whereBetween('votes', [1, 100])
            ->get();
```

whereNotBetween / orWhereNotBetween

whereNotBetween 方法用於驗證欄位的值是否不在給定的兩個值範圍之中：

```
$users = DB::table('users')
                        ->whereNotBetween('votes', [1, 100])
                        ->get();
```

whereBetweenColumns / whereNotBetweenColumns / orWhereBetweenColumns / orWhereNotBetweenColumns

whereBetweenColumns 方法用於驗證欄位是否在給定的兩個欄位的值的範圍中：

```
$patients = DB::table('patients')
                            ->whereBetweenColumns('weight', ['minimum_allowed_weight',
'maximum_allowed_weight'])
                            ->get();
```

whereNotBetweenColumns 方法用於驗證欄位是否不在給定的兩個欄位的值的範圍中：

```
$patients = DB::table('patients')
                            ->whereNotBetweenColumns('weight', ['minimum_allowed_weight',
'maximum_allowed_weight'])
                            ->get();
```

whereIn / whereNotIn / orWhereIn / orWhereNotIn

whereIn 方法用於驗證欄位是否在給定的值陣列中：

```
$users = DB::table('users')
                        ->whereIn('id', [1, 2, 3])
                        ->get();
```

whereIn 方法用於驗證欄位是否不在給定的值陣列中：

```
$users = DB::table('users')
                        ->whereNotIn('id', [1, 2, 3])
                        ->get();
```

你也可以為 whereIn 方法的第二個參數提供一個子查詢：

```
$activeUsers = DB::table('users')->select('id')->where('is_active', 1);

$users = DB::table('comments')
                        ->whereIn('user_id', $activeUsers)
                        ->get();
```

上面的例子將會轉換為下面的 SQL 查詢語句：

```
select * from comments where user_id in (
    select id
    from users
    where is_active = 1
)
```

> **注意**
> 如果你需要判斷一個整數的大陣列 whereIntegerInRaw 或 whereIntegerNotInRaw 方法可能會更適合，這種用法的記憶體佔用更小。

whereNull / whereNotNull / orWhereNull / orWhereNotNull

whereNull 方法用於判斷指定的欄位的值是否是 NULL：

```
$users = DB::table('users')
                    ->whereNull('updated_at')
                    ->get();
```

whereNotNull 方法是用來驗證給定欄位的值是否不為 NULL:

```
$users = DB::table('users')
                ->whereNotNull('updated_at')
                ->get();
```

whereDate / whereMonth / whereDay / whereYear / whereTime

whereDate 方法是用來比較欄位的值與給定的日期值是否相等:

```
$users = DB::table('users')
                ->whereDate('created_at', '2016-12-31')
                ->get();
```

whereMonth 方法是用來比較欄位的值與給定的月是否相等:

```
$users = DB::table('users')
                ->whereMonth('created_at', '12')
                ->get();
```

whereDay 方法是用來比較欄位的值與給定的日是否相等:

```
$users = DB::table('users')
                ->whereDay('created_at', '31')
                ->get();
```

whereYear 方法是用來比較欄位的值與給定的年是否相等:

```
$users = DB::table('users')
                ->whereYear('created_at', '2016')
                ->get();
```

whereTime 方法是用來比較欄位的值與給定的時間是否相等:

```
$users = DB::table('users')
                ->whereTime('created_at', '=', '11:20:45')
                ->get();
```

whereColumn / orWhereColumn

whereColumn 方法是用來比較兩個給定欄位的值是否相等:

```
$users = DB::table('users')
                ->whereColumn('first_name', 'last_name')
                ->get();
```

你也可以將比較運算子傳遞給 whereColumn 方法:

```
$users = DB::table('users')
                ->whereColumn('updated_at', '>', 'created_at')
                ->get();
```

你還可以向 whereColumn 方法中傳遞一個陣列。這些條件將使用 and 運算子聯接:

```
$users = DB::table('users')
                ->whereColumn([
                    ['first_name', '=', 'last_name'],
                    ['updated_at', '>', 'created_at'],
                ])->get();
```

42.7.6 邏輯分組

有時你可能需要將括號內的幾個「where」子句分組，以實現查詢所需的邏輯分組。實際上應該將 orWhere 方法的呼叫分組到括號中，以避免不可預料的查詢邏輯誤差。因此可以傳遞閉包給 where 方法：

```
$users = DB::table('users')
            ->where('name', '=', 'John')
            ->where(function (Builder $query) {
                $query->where('votes', '>', 100)
                      ->orWhere('title', '=', 'Admin');
```

```
            })
            ->get();
```

你可以看到， 通過一個閉包寫入 where 方法 建構一個查詢構造器來約束一個分組。這個閉包接收一個查詢實例，你可以使用這個實例來設定應該包含的約束。上面的例子將生成以下 SQL：

```
select * from users where name = 'John' and (votes > 100 or title = 'Admin')
```

注意
你應該用 orWhere 呼叫這個分組，以避免應用全域作用時出現意外。

42.7.7 高級 Where 語句

42.7.8 Where Exists 語句

whereExists 方法允許你使用 where exists SQL 語句。 whereExists 方法接收一個閉包參數，該閉包獲取一個查詢建構器實例，從而允許你定義放置在 exists 子句中查詢:

```
$users = DB::table('users')
            ->whereExists(function (Builder $query) {
                $query->select(DB::raw(1))
                      ->from('orders')
                      ->whereColumn('orders.user_id', 'users.id');
            })
            ->get();
```

或者，可以向 whereExists 方法提供一個查詢對象，替換上面的閉包：

```
$orders = DB::table('orders')
                ->select(DB::raw(1))
                ->whereColumn('orders.user_id', 'users.id');

$users = DB::table('users')
                    ->whereExists($orders)
                    ->get();
```

上面的兩個示例都會生成如下的 SQL 語句

```
select * from users
where exists (
    select 1
    from orders
    where orders.user_id = users.id
)
```

42.7.9 子查詢 Where 語句

有時候，你可能需要構造一個 where 子查詢，將子查詢的結果與給定的值進行比較。你可以通過向 where 方法傳遞閉包和值來實現此操作。例如，下面的查詢將檢索最後一次「會員」購買記錄是「Pro」類型的所有使用者；

```
use App\Models\User;
use Illuminate\Database\Query\Builder;

$users = User::where(function (Builder $query) {
    $query->select('type')
        ->from('membership')
        ->whereColumn('membership.user_id', 'users.id')
        ->orderByDesc('membership.start_date')
        ->limit(1);
}, 'Pro')->get();
```

或者，你可能需要建構一個 where 子句，將列與子查詢的結果進行比較。你可以通過將列、運算子和閉包傳遞給 where 方法來完成此操作。例如，以下查詢將檢索金額小於平均值的所有收入記錄；

```
use App\Models\Income;
use Illuminate\Database\Query\Builder;

$incomes = Income::where('amount', '<', function (Builder $query) {
    $query->selectRaw('avg(i.amount)')->from('incomes as i');
})->get();
```

42.7.10 全文 Where 子句

注意

MySQL 和 PostgreSQL 目前支援全文 where 子句。

可以使用 where FullText 和 orWhere FullText 方法將全文「where」 子句新增到具有 full text indexes 的列的查詢中。這些方法將由 Laravel 轉換為適用於底層資料庫系統的 SQL。例如，使用 MySQL 的應用會生成 MATCH AGAINST 子句

```
$users = DB::table('users')
            ->whereFullText('bio', 'web developer')
            ->get();
```

42.8 Ordering, Grouping, Limit & Offset

42.8.1 排序

42.8.1.1 orderBy 方法

orderBy 方法允許你按給定列對查詢結果進行排序。orderBy 方法接受的第一個參數應該是你希望排序的列，而第二個參數確定排序的方向，可以是 asc 或 desc：

```
$users = DB::table('users')
                ->orderBy('name', 'desc')
                ->get();
```

要按多列排序，你以根據需要多次呼叫 orderBy：

```
$users = DB::table('users')
                ->orderBy('name', 'desc')
                ->orderBy('email', 'asc')
                ->get();
```

42.8.1.2 latest 和 oldest 方法

latest 和 oldest 方法可以方便讓你把結果根據日期排序。查詢結果默認根據資料表的 created_at 欄位進行排序 。或者，你可以傳一個你想要排序的列名，通過:

```
$user = DB::table('users')
                ->latest()
                ->first();
```

42.8.1.3 隨機排序

inRandomOrder 方法被用來將查詢結果隨機排序。例如，你可以使用這個方法去獲得一個隨機使用者:

```
$randomUser = DB::table('users')
                ->inRandomOrder()
```

```
                    ->first();
```

42.8.1.4 移除已存在的排序

reorder 方法會移除之前已經被應用到查詢裡的排序:

```
$query = DB::table('users')->orderBy('name');

$unorderedUsers = $query->reorder()->get();
```

當你呼叫 reorder 方法去移除所有已經存在的排序的時候，你可以傳遞一個列名和排序方式去重新排序整個查詢:

```
$query = DB::table('users')->orderBy('name');

$usersOrderedByEmail = $query->reorder('email', 'desc')->get();
```

42.8.2 分組

42.8.2.1 groupBy 和 having 方法

如你所願，groupBy 和 having 方法可以將查詢結果分組。having 方法的使用方法類似於 where 方法:

```
$users = DB::table('users')
                    ->groupBy('account_id')
                    ->having('account_id', '>', 100)
                    ->get();
```

你可以使用 havingBetween 方法在一個給定的範圍內去過濾結果:

```
$report = DB::table('orders')
                    ->selectRaw('count(id) as number_of_orders, customer_id')
                    ->groupBy('customer_id')
                    ->havingBetween('number_of_orders', [5, 15])
                    ->get();
```

你可以傳多個參數給 groupBy 方法將多列分組:

```
$users = DB::table('users')
                    ->groupBy('first_name', 'status')
                    ->having('account_id', '>', 100)
                    ->get();
```

想要構造更高級的 having 語句, 看 havingRaw 方法。

42.8.3 限制和偏移量

42.8.3.1 skip 和 take 方法

你可以使用 skip 和 take 方法去限制查詢結果的返回數量或者在查詢結果中跳過給定數量:

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

或者，你可以使用 limit 和 offset 方法。這些方法在功能上等同於 take 和 skip 方法, 如下

```
$users = DB::table('users')
                    ->offset(10)
                    ->limit(5)
                    ->get();
```

42.9 條件語句

有時，可能希望根據另一個條件將某些查詢子句應用於查詢。例如，當傳入 HTTP 請求有一個給定的值的時候你才需要使用一個 `where` 語句。你可以使用 `when` 方法去實現:

```
$role = $request->string('role');

$users = DB::table('users')
                ->when($role, function (Builder $query, string $role) {
                    $query->where('role_id', $role);
                })
                ->get();
```

`when` 方法只有當第一個參數為 `true` 時才執行給定的閉包。如果第一個參數是 `false` ，閉包將不會被執行。因此，在上面的例子中，只有在傳入的請求包含 `role` 欄位且結果為 `true` 時，`when` 方法裡的閉包才會被呼叫。

你可以將另一個閉包作為第三個參數傳遞給 `when` 方法。這個閉包則旨在第一個參數結果為 `false` 時才會執行。為了說明如何使用該功能，我們將使用它來組態查詢的默認排序：

```
$sortByVotes = $request->boolean('sort_by_votes');

$users = DB::table('users')
                ->when($sortByVotes, function (Builder $query, bool $sortByVotes) {
                    $query->orderBy('votes');
                }, function (Builder $query) {
                    $query->orderBy('name');
                })
                ->get();
```

42.10 插入語句

查詢構造器也提供了一個 `insert` 方法來用於插入記錄到資料庫表中。`insert` 方法接受一個列名和值的陣列：

```
DB::table('users')->insert([
    'email' => 'kayla@example.com',
    'votes' => 0
]);
```

你可以通過傳遞一個二維陣列來實現一次插入多條記錄。每一個陣列都代表了一個應當插入到資料表中的記錄：

```
DB::table('users')->insert([
    ['email' => 'picard@example.com', 'votes' => 0],
    ['email' => 'janeway@example.com', 'votes' => 0],
]);
```

`insertOrIgnore` 方法將會在插入資料庫的時候忽略發生的錯誤。當使用該方法時，你應當注意，重複記錄插入的錯誤和其他類型的錯誤都將被忽略，這取決於資料庫引擎。例如， `insertOrIgnore` 將會 繞過 MySQL 的嚴格模式 ：

```
DB::table('users')->insertOrIgnore([
    ['id' => 1, 'email' => 'sisko@example.com'],
    ['id' => 2, 'email' => 'archer@example.com'],
]);
```

`insertUsing` 方法將在表中插入新記錄，同時用子查詢來確定應插入的資料：

```
DB::table('pruned_users')->insertUsing([
    'id', 'name', 'email', 'email_verified_at'
], DB::table('users')->select(
    'id', 'name', 'email', 'email_verified_at'
```

```
)->where('updated_at', '<=', now()->subMonth()));
```

42.10.1.1 自增 IDs

如果資料表有自增 ID ，使用 `insertGetId` 方法來插入記錄可以返回 ID 值：

```
$id = DB::table('users')->insertGetId(
    ['email' => 'john@example.com', 'votes' => 0]
);
```

> **注意**
> 當使用 PostgreSQL 時，`insertGetId` 方法將默認把 `id` 作為自動遞增欄位的名稱。如果你要從其他「欄位」來獲取 ID ，則需要將欄位名稱作為第二個參數傳遞給 `insertGetId` 方法。

42.10.2 更新插入

`upsert` 方法是是插入不存在的記錄和為已經存在記錄更新值。該方法的第一個參數包含要插入或更新的值，而第二個參數列出了在關聯表中唯一標識記錄的列。 該方法的第三個也是最後一個參數是一個列陣列，如果資料庫中已經存在匹配的記錄，則應該更新這些列：

```
DB::table('flights')->upsert(
    [
        ['departure' => 'Oakland', 'destination' => 'San Diego', 'price' => 99],
        ['departure' => 'Chicago', 'destination' => 'New York', 'price' => 150]
    ],
    ['departure', 'destination'],
    ['price']
);
```

在上面的例子中，Laravel 會嘗試插入兩條記錄。如果已經存在具有相同 `departure` 和 `destination` 列值的記錄，Laravel 將更新該記錄的 `price` 列。

> **注意**
> 除 SQL Server 之外的所有資料庫都要求 `upsert` 方法的第二個參數中的列具有「主」或「唯一」索引。 此外，MySQL 資料庫驅動程式忽略 `upsert` 方法的第二個參數，並始終使用表的「主」和「唯一」索引來檢測現有記錄。

42.11 更新語句

除了插入記錄到資料庫之外，查詢構造器也可以使用 `update` 方法來更新已經存在的記錄。`update` 方法像 `insert` 方法一樣，接受一個列名和值的陣列作為參數，它表示要更新的列和資料。`update` 方法返回受影響的行數。你可以使用 `where` 子句來限制 `update` 查詢：

```
$affected = DB::table('users')
              ->where('id', 1)
              ->update(['votes' => 1]);
```

42.11.1.1 更新或插入

有時你可能希望更新資料庫中的記錄，但如果指定記錄不存在的時候則建立它。在這種情況下，可以使用 `updateOrInsert` 方法。`updateOrInsert` 方法接受兩個參數：一個用於尋找記錄的條件陣列，以及一個包含要更改記錄的鍵值對陣列。

`updateOrInsert` 方法將嘗試使用第一個參數的列名和值來定位匹配的資料庫記錄。如果記錄存在，則使用第二個參數更新其值。如果找不到指定記錄，則會合併兩個參數的屬性來建立一條記錄並將其插入：

```
DB::table('users')
    ->updateOrInsert(
        ['email' => 'john@example.com', 'name' => 'John'],
        ['votes' => '2']
    );
```

42.11.2 更新 JSON 欄位

當更新一個 JSON 列的收，你可以使用 `->` 語法來更新 JSON 對象中恰當的鍵。此操作需要 MySQL 5.7+ 和 PostgreSQL 9.5+ 的資料庫：

```
$affected = DB::table('users')
                ->where('id', 1)
                ->update(['options->enabled' => true]);
```

42.11.3 自增與自減

查詢構造器還提供了方便的方法來增加或減少給定列的值。這兩種方法都至少接受一個參數：要修改的列。可以提供第二個參數來指定列應該增加或減少的數量：

```
DB::table('users')->increment('votes');

DB::table('users')->increment('votes', 5);

DB::table('users')->decrement('votes');

DB::table('users')->decrement('votes', 5);
```

你還可以在操作期間指定要更新的其他列：

```
DB::table('users')->increment('votes', 1, ['name' => 'John']);
```

此外，你可以使用 `incrementEach` 和 `decrementEach` 方法同時增加或減少多個列:

```
DB::table('users')->incrementEach([
    'votes' => 5,
    'balance' => 100,
]);
```

42.12 刪除語句

查詢建構器的 `delete` 方法可用於從表中刪除記錄。 `delete` 方法返回受影響的行數。你可以通過在呼叫 `delete` 方法之前新增 `where` 子句來限制 `delete` 語句：

```
$deleted = DB::table('users')->delete();

$deleted = DB::table('users')->where('votes', '>', 100)->delete();
```

如果你希望截斷整個表，這將從表中刪除所有記錄並將自動遞增 ID 重設為零，你可以使用 `truncate` 方法：

```
DB::table('users')->truncate();
```

42.12.1.1 截斷表 & PostgreSQL

截斷 PostgreSQL 資料庫時，將應用 CASCADE 行為。這意味著其他表中所有與外部索引鍵相關的記錄也將被刪除。

42.13 悲觀鎖

查詢建構器還包括一些函數，可幫助你在執行 `select` 語句時實現「悲觀鎖」。要使用「共享鎖」執行語句，你可以呼叫 `sharedLock` 方法。共享鎖可防止選定的行被修改，直到你的事務被提交：

```
DB::table('users')
         ->where('votes', '>', 100)
         ->sharedLock()
         ->get();
```

或者，你可以使用 `lockForUpdate` 方法。「update」鎖可防止所選記錄被修改或被另一個共享鎖選中：

```
DB::table('users')
         ->where('votes', '>', 100)
         ->lockForUpdate()
         ->get();
```

42.14 偵錯

你可以在建構查詢時使用 `dd` 和 `dump` 方法來轉儲當前查詢繫結和 SQL。`dd` 方法將顯示偵錯資訊，然後停止執行請求。`dump` 方法將顯示偵錯資訊，但允許請求繼續執行：

```
DB::table('users')->where('votes', '>', 100)->dd();

DB::table('users')->where('votes', '>', 100)->dump();
```

43 分頁

43.1 介紹

在其他框架中，分頁可能非常痛苦，我們希望 Laravel 的分頁方法像一股新鮮空氣。 Laravel 的分頁器整合了 query builder 和 Eloquent ORM，並提供了方便、易於使用的無需任何組態的資料庫記錄分頁。

默認情況下，由分頁器生成的 HTML 與 Tailwind CSS 框架 相容，然而，引導分頁支援也是可用的。

43.1.1.1 Tailwind JIT

如果你使用 Laravel 的默認 Tailwind 檢視和 Tailwind JIT 引擎，你應該確保你的應用程式的 `tailwind.config.js` 檔案的 `content` 關鍵引用 Laravel 的分頁檢視，這樣它們的 Tailwind 類就不會被清除：

```
content: [
    './resources/**/*.blade.php',
    './resources/**/*.js',
    './resources/**/*.vue',
    './vendor/laravel/framework/src/Illuminate/Pagination/resources/views/*.blade.php',
],
```

43.2 基礎用法

43.2.1 對查詢構造器結果進行分頁

有幾種方法可以對結果進行分頁，最簡單的方法是在 query builder 或 Eloquent query 上使用 `paginate` 方法，`paginate` 方法根據使用者查看的當前頁面自動設定查詢的「limit」和「offset」，默認情況下，通過 HTTP 請求中的 `page` 查詢字串參數的值檢測當前頁面，Laravel 會自動檢測這個值，它也會自動插入到分頁器生成的連結中。

在下面的例子中，傳遞給 `paginate` 方法的唯一參數是你想要在一頁中顯示的記錄數。在此例中，我們希望「每頁」顯示 `15` 條資料：

```
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller
{
    /**
     * 顯示應用中所有使用者列表
     */
    public function index(): View
    {
        return view('user.index', [
            'users' => DB::table('users')->paginate(15)
        ]);
    }
}
```

43.2.1.1 簡單分頁

該 `paginate` 方法會在查詢資料庫之前先計算與查詢匹配的記錄總數，從而讓分頁器知道總共需要有多少個頁面來顯示所有的記錄。不過，如果你不打算在介面上顯示總頁數的話，那麼計算記錄總數是沒有意義的。

因此，如果你只需要顯示一個簡單的「上一頁」和「下一頁」連結的話， `simplePaginate` 方法是一個更高效的選擇：

```
$users = DB::table('users')->simplePaginate(15);
```

43.2.2 Eloquent ORM 分頁

你也可以對 Eloquent 查詢結果進行分頁. 在下面的例子中，我們將 `App\Models\User` 模型按每頁 15 條記錄進行分頁。如你所見，其語法與查詢構造器分頁基本相同：

```
use App\Models\User;

$users = User::paginate(15);
```

當然，你也可以在呼叫 `paginate` 方法之前為查詢新增其他約束，例如 `where` 子句：

```
$users = User::where('votes', '>', 100)->paginate(15);
```

你也可以在 Eloquent ORM 分頁中使用 `simplePaginate`：

```
$users = User::where('votes', '>', 100)->simplePaginate(15);
```

同樣，您可以使用 `cursorPaginate` 方法對 Eloquent 模型進行遊標分頁：

```
$users = User::where('votes', '>', 100)->cursorPaginate(15);
```

43.2.2.1 每頁有多個 Paginator 實例

有時你可能需要在應用程式呈現的單個螢幕上呈現兩個單獨的分頁器。 但是，如果兩個分頁器實例都使用 `page` 查詢字串參數來儲存當前頁面，則兩個分頁器會發生衝突。 要解決此衝突，您可以通過提供給 `paginate`、`simplePaginate` 和 `cursorPaginate` 方法的第三個參數傳遞你希望用於儲存分頁器當前頁面的查詢字串參數的名稱：

```
use App\Models\User;

$users = User::where('votes', '>', 100)->paginate(
    $perPage = 15, $columns = ['*'], $pageName = 'users'
);
```

43.2.3 游標分頁

雖然 `paginate` 和 `simplePaginate` 使用 SQL「offset」 子句建立查詢，但游標分頁通過構造「where」子句來工作，這些子句比較查詢中包含的有序列的值，提供所有可用的最有效的資料庫性能 Laravel 的分頁方法。這種分頁方法特別適合大型資料集和「無限」滾動使用者介面。

與基於偏移量的分頁在分頁器生成的 URL 的查詢字串中包含頁碼不同，基於游標的分頁在查詢字串中放置一個「游標」字串。游標是一個編碼字串，包含下一個分頁查詢應該開始分頁的位置和它應該分頁的方向：

```
http://localhost/users?cursor=eyJpZCI6MTUsIl9wb2ludHNUb05leHRJdGVtcyI6dHJ1ZX0
```

你可以通過查詢生成器提供的 `cursorPaginate` 方法建立基於游標的分頁器實例。這個方法返回一個 `Illuminate\Pagination\CursorPaginator` 的實例：

```
$users = DB::table('users')->orderBy('id')->cursorPaginate(15);
```

檢索到游標分頁器實例後，你可以像使用 `paginate` 和 `simplePaginate` 方法時一樣顯示分頁結果。更多游標分頁器提供的實例方法請參考游標分頁器實例方法文件.

注意 你的查詢必須包含「order by」子句才能使用游標分頁。

43.2.3.1 游標與偏移分頁

為了說明偏移分頁和游標分頁之間的區別，讓我們檢查一些示例 SQL 查詢。 以下兩個查詢都將顯示按 `id` 排序的 `users` 表的「第二頁」結果：

```
# 偏移分頁...
select * from users order by id asc limit 15 offset 15;

# 游標分頁...
select * from users where id > 15 order by id asc limit 15;
```

與偏移分頁相比，游標分頁查詢具有以下優勢：

- 對於大型資料集，如果「order by」列被索引，游標分頁將提供更好的性能。 這是因為「offset」子句會掃描所有先前匹配的資料。
- 對於頻繁寫入的資料集，如果最近在使用者當前查看的頁面中新增或刪除了結果，偏移分頁可能會跳過記錄或顯示重複。

但是，游標分頁有以下限制：

- 與 `simplePaginate` 一樣，游標分頁只能用於顯示「下一個」和「上一個」連結，不支援生成帶頁碼的連結。
- 它要求排序基於至少一個唯一列或唯一列的組合。 不支援具有 `null` 值的列。

-「order by」子句中的查詢表示式僅在它們被別名並新增到「select」子句時才受支援。

43.2.4 手動建立分頁

有時你可能希望手動建立分頁，並傳遞一個包含資料的陣列給它。這可以通過手動建立 `Illuminate\Pagination\Paginator`, `Illuminate\Pagination\LengthAwarePaginator` 或者 `Illuminate\Pagination\CursorPaginator` 實例來實現，這取決於你的需要。

`Paginator` 不需要知道資料的總數。然而，你也無法通過 `Paginator` 獲取最後一頁的索引。而 `LengthAwarePaginator` 接受和 `Paginator` 幾乎相同的參數，不過，它會計算資料的總數。

或者說，`Paginator` 相當於查詢構造器或者 Eloquent ORM 分頁的 `simplePaginate` 方法，而 `LengthAwarePaginator` 相當於 `paginate` 方法。

> **注意**
> 手動建立分頁器實例時，你應該手動「切片」傳遞給分頁器的結果陣列。如果你不確定如何執行此操作，請查看 array_slice PHP 函數。

43.2.5 自訂分頁的 URL

默認情況下，分頁器生成的連結會匹配當前的請求 URL。不過，分頁器的 withPath 方法允許你自訂分頁器生成連結時使用的 URL。比如說，你想要分頁器生成類似 https://example.com/admin/users?page=N 的連結，你應該給 `withPath` 方法傳遞 `/admin/users` 參數：

```
use App\Models\User;

Route::get('/users', function () {
    $users = User::paginate(15);
```

```
    $users->withPath('/admin/users');

    // ...
});
```

43.2.5.1 附加參數到分頁連結

你可以使用 `appends` 方法向分頁連結中新增查詢參數。例如，要在每個分頁連結中新增 `sort=votes` ，你應該這樣呼叫 `appends`：

```
use App\Models\User;

Route::get('/users', function () {
    $users = User::paginate(15);

    $users->appends(['sort' => 'votes']);

    // ...
});
```

如果你想要把當前所有的請求查詢參數新增到分頁連結，你可以使用 `withQueryString` 方法：

```
$users = User::paginate(15)->withQueryString();
```

43.2.5.2 附加 hash 片段

如果你希望向分頁器的 URL 新增「雜湊片段」，你可以使用 `fragment` 方法。例如，你可以使用 `fragment` 方法，為 `#user` 新增分頁連結：

```
$users = User::paginate(15)->fragment('users');
```

43.3 顯示分頁結果

當呼叫 `paginate` 方法時， 你會得到一個 `Illuminate\Pagination\LengthAwarePaginator` 實例，而呼叫 `simplePaginate` 方法時，會得到一個 `Illuminate\Pagination\Paginator` 實例。 最後，呼叫 `cursorPaginate` 方法，會得到 `Illuminate\Pagination\CursorPaginator` 實例。

這些對象提供了數個方法來獲取結果集的資訊。除了這些輔助方法外，分頁器的實例是迭代器，可以像陣列一樣遍歷。所以，你可以使用 Blade 範本來顯示資料、渲染分頁連結等：

```
<div class="container">
    @foreach ($users as $user)
        {{ $user->name }}
    @endforeach
</div>

{{ $users->links() }}
```

links 方法會渲染結果集中剩餘頁面的連結。每個連結都包含了 page 查詢字串變數。請記住，links 方法生成的 HTML 相容 Tailwind CSS 框架 。

43.3.1 調整分頁連結窗口

在使用分頁器展示分頁連結時，將展示當前頁及當前頁面前後各三頁的連結。如果有需要，你可以通過 `onEachSide` 方法來控制每側顯示多少個連結：

```
{{ $users->onEachSide(5)->links() }}
```

43.3.2 將結果轉換為 JSON

Laravel 分頁器類實現了 `Illuminate\Contracts\Support\Jsonable` 介面契約，提供了 `toJson` 方法。這意味著你可以很方便地將分頁結果轉換為 JSON。你也可以通過直接在路由閉包或者 controller 方法中返回分頁實例來將其轉換為 JSON：

```
use App\Models\User;

Route::get('/users', function () {
    return User::paginate();
});
```

分頁器生成的 JSON 會包括諸如 `total`，`current_page`，`last_page` 等中繼資料資訊。實際結果對象將通過 JSON 陣列的 `data` 鍵提供。以下是通過自路由中分頁器實例的方式建立 JSON 的例子：

```
{
   "total": 50,
   "per_page": 15,
   "current_page": 1,
   "last_page": 4,
   "first_page_url": "http://laravel.app?page=1",
   "last_page_url": "http://laravel.app?page=4",
   "next_page_url": "http://laravel.app?page=2",
   "prev_page_url": null,
   "path": "http://laravel.app",
   "from": 1,
   "to": 15,
   "data":[
        {
            // 分頁資料...
        },
        {
            // 分頁資料...
        }
   ]
}
```

43.4 自訂分頁檢視

默認情況下，分頁器渲染的檢視與 Tailwind CSS 相容。不過，如果你並非使用 Tailwind，你也可以自由地定義用於渲染這些連結的檢視。在呼叫分頁器實例的 `links` 方法時，將檢視名稱作為第一個參數傳遞給該方法：

```
{{ $paginator->links('view.name') }}

<!-- 向檢視傳遞參數... -->
{{ $paginator->links('view.name', ['foo' => 'bar']) }}
```

不過，最簡單的自訂分頁檢視的方法依然是使用 `vendor:publish` 命令將它們匯出到 `resources/views/vendor` 目錄：

```
php artisan vendor:publish --tag=laravel-pagination
```

這個命令將會把分頁檢視匯出到 `resources/views/vendor/pagination` 目錄。該目錄下的 `tailwind.blade.php` 檔案就是默認的分頁檢視。你可以通過編輯這一檔案來自訂分頁檢視。

如果你想要定義不同的檔案作為默認的分頁檢視，你可以在 `App\Providers\AppServiceProvider` 服務提供者中的 `boot` 方法內呼叫 `defaultView` 和 `defaultSimpleView` 方法：

```
<?php

namespace App\Providers;

use Illuminate\Pagination\Paginator;
```

```
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * 引導應用程式服務
     */
    public function boot(): void
    {
        Paginator::defaultView('view-name');

        Paginator::defaultSimpleView('view-name');
    }
}
```

43.4.1 使用 Bootstrap

Laravel 同樣包含使用 Bootstrap CSS 建構的分頁檢視。要使用這些檢視來替代默認的 Tailwind 檢視，你可以在 `App\Providers\AppServiceProvider` 服務提供者中的 `boot` 方法內呼叫分頁器的 `useBootstrapFour` 或 `useBootstrapFive` 方法：

```
use Illuminate\Pagination\Paginator;

/**
 * 引導應用程式服務
 */
public function boot(): void
{
    Paginator::useBootstrapFive();
    Paginator::useBootstrapFour();
}
```

43.5 分頁器實例方法

每一個分頁器實例都提供了下列方法來獲取分頁資訊：

方法	描述
`$paginator->count()`	獲取分頁的總資料
`$paginator->currentPage()`	獲取當前頁碼
`$paginator->firstItem()`	獲取結果集中第一個資料的編號
`$paginator->getOptions()`	獲取分頁器選項
`$paginator->getUrlRange($start, $end)`	建立指定頁數範圍的 URL
`$paginator->hasPages()`	是否有足夠多的資料來建立多個頁面
`$paginator->hasMorePages()`	是否有更多的頁面可供展示
`$paginator->items()`	獲取當前頁的資料項
`$paginator->lastItem()`	獲取結果集中最後一個資料的編號
`$paginator->lastPage()`	獲取最後一頁的頁碼 （在 `simplePaginate` 中不可用）
`$paginator->nextPageUrl()`	獲取下一頁的 URL
`$paginator->onFirstPage()`	當前頁是否為第一頁
`$paginator->perPage()`	獲取每一頁顯示的數量總數
`$paginator->previousPageUrl()`	獲取上一頁的 URL
`$paginator->total()`	獲取結果集中的資料總數 （在 `simplePaginate` 中不可用）
`$paginator->url($page)`	獲取指定頁的 URL
`$paginator->getPageName()`	獲取用於儲存頁碼的查詢參數名

方法	描述
`$paginator->setPageName($name)`	設定用於儲存頁碼的查詢參數名

43.6 游標分頁器實例方法

每一個分頁器實例都提供了下列額外方法來獲取分頁資訊:

方法	描述
`$paginator->count()`	獲取當前頁的資料總數
`$paginator->cursor()`	獲取當前分頁實例
`$paginator->getOptions()`	獲取分頁偏好設定
`$paginator->hasPages()`	判斷是否有足夠資料用於分頁
`$paginator->hasMorePages()`	判斷資料儲存是否還有更多項目
`$paginator->getCursorName()`	獲取用於查詢實例的變數名稱
`$paginator->items()`	獲取當前頁面的資料項目
`$paginator->nextCursor()`	獲取下一頁資料實例
`$paginator->nextPageUrl()`	獲取下一頁 URL
`$paginator->onFirstPage()`	判斷頁面是否屬於第一頁
`$paginator->perPage()`	每頁顯示的資料數量
`$paginator->previousCursor()`	獲取上一頁資料實例
`$paginator->previousPageUrl()`	獲取上一頁 URL
`$paginator->setCursorName()`	設定用於查詢實例的變數名稱
`$paginator->url($cursor)`	獲取指定實例的 URL

44 遷移

44.1 介紹

遷移就像資料庫的版本控制，允許你的團隊定義和共享應用程式的資料庫架構定義。 如果你曾經不得不告訴團隊成員在從程式碼控制中拉取更新後手動新增欄位到他們的本地資料庫，那麼你就遇到了資料庫遷移解決的問題。

Laravel `Schema` facade 為所有 Laravel 支援的資料庫系統的建立和操作表提供了不依賴於資料庫的支援。通常情況下，遷移會使用 facade 來建立和修改資料表和欄位。

44.2 生成遷移

你可以使用 `make:migration` Artisan 命令 來生成資料庫遷移。新的遷移檔案將放在你的 `database/migrations` 目錄下。每個遷移檔案名稱都包含一個時間戳來使 Laravel 確定遷移的順序：

```
php artisan make:migration create_flights_table
```

Laravel 將使用遷移檔案的名稱來猜測表名以及遷移是否會建立一個新表。如果 Laravel 能夠從遷移檔案的名稱中確定表的名稱，它將在生成的遷移檔案中預填入指定的表，或者，你也可以直接在遷移檔案中手動指定表名。

如果要為生成的遷移指定自訂路徑，你可以在執行 `make:migration` 命令時使用 `--path` 選項。給定的路徑應該相對於應用程式的基本路徑。

> **技巧**
> 可以使用 stub publishing 自訂發佈。

44.2.1 整合遷移

在建構應用程式時，可能會隨著時間的推移積累越來越多的遷移。這可能會導致你的 `database/migrations` 目錄因為數百次遷移而變得臃腫。你如果願意的話，可以將遷移「壓縮」到單個 SQL 檔案中。如果你想這樣做，請先執行 `schema:dump` 命令：

```
php artisan schema:dump

# 轉儲當前資料庫架構並刪除所有現有遷移...
php artisan schema:dump --prune
```

執行此命令時，Laravel 將嚮應用程序的 `database/schema` 目錄寫入一個「schema」檔案。現在，當你嘗試遷移資料庫而沒有執行其他遷移時，Laravel 將首先執行模式檔案的 SQL 語句。在執行資料庫結構檔案的語句之後，Laravel 將執行不屬於資料庫結構的剩餘的所有遷移。

如果你的應用程式的測試使用的資料庫連接與你在本地開發過程中通常使用的不同，你應該確保你已經使用該資料庫連接轉儲了一個 schema 檔案，以便你的測試能夠建立你的資料庫。你可能希望在切換（dump）你在本地開發過程中通常使用的資料庫連接之後再做這件事。

```
php artisan schema:dump
php artisan schema:dump --database=testing --prune
```

你應該將資料庫模式檔案提交給原始碼管理，以便團隊中的其他新開發人員可以快速建立應用程式的初始資料庫結構。

注意
整合遷移僅適用於 MySQL、PostgreSQL 和 SQLite 資料庫，並使用資料庫命令列的客戶端。另外，資料庫結構不能還原到記憶體中的 SQLite 資料庫。

44.3 遷移結構

遷移類包含兩個方法：up 和 down 。up 方法用於向資料庫中新增新表、列或索引，而 down 方法用於撤銷 up 方法執行的操作。.

在這兩種方法中，可以使用 Laravel 模式建構器來富有表現力地建立和修改表。要瞭解 Schema 建構器上可用的所有方法，查看其文件。例如，以下遷移會建立一個 flights 表：

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * 執行遷移
     */
    public function up(): void
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }

    /**
     * 回滾遷移
     */
    public function down(): void
    {
        Schema::drop('flights');
    }
};
```

44.3.1.1 設定遷移連接

如果你的遷移將與應用程式默認資料庫連接以外的資料庫連接進行互動，你應該設定遷移的 $connection 屬性：

```
/**
 * The database connection that should be used by the migration.
 *
 * @var string
 */
protected $connection = 'pgsql';

/**
 * 執行遷移
 */
public function up(): void
{
    // ...
}
```

44.4 執行遷移

執行 Artisan 命令 `migrate`，來運行所有未執行過的遷移：

```
php artisan migrate
```

如果你想查看目前已經執行了哪些遷移，可以使用 `migrate:status` Artisan 命令：

```
php artisan migrate:status
```

如果你希望在不實際運行遷移的情況下看到將被執行的 SQL 語句，你可以在 `migrate` 命令中提供 `--pretend` 選項。

```
php artisan migrate --pretend
```

44.4.1.1 在隔離的環境中執行遷移

如果你在多個伺服器上部署你的應用程式，並在部署過程中運行遷移，你可能不希望兩個伺服器同時嘗試遷移資料庫。為了避免這種情況，你可以在呼叫 `migrate` 命令時使用 `isolated` 選項。

當提供 `isolated` 選項時, Laravel 將使用你的應用程式快取驅動獲得一個原子鎖，然後再嘗試運行你的遷移。所有其他試圖運行 `migrate` 命令的嘗試在鎖被持有時都不會執行; 然而, 命令仍然會以成功的退出狀態碼退出:

```
php artisan migrate --isolated
```

> **注意**
> 要使用這個功能，你的應用程式必須使用 `memcached` / `redis` / `dynamodb` / `database` / `file` 或 `array` 快取驅動作為你應用程式的默認快取驅動。此外，所有的伺服器必須與同一個中央快取伺服器進行通訊。

44.4.1.2 在生產環境中執行強制遷移

有些遷移操作是破壞性的，這意味著它們可能會導致資料丟失。為了防止你對生產資料庫運行這些命令，在執行這些命令之前，系統將提示你進行確認。如果要在運行強制命令的時候去掉提示，需要加上 `--force` 標誌：

```
php artisan migrate --force
```

44.4.2 回滾遷移

如果要回滾最後一次遷移操作，可以使用 Artisan 命令 `rollback`。該命令會回滾最後「一批」的遷移，這可能包含多個遷移檔案：

```
php artisan migrate:rollback
```

通過向 `rollback` 命令加上 `step` 參數，可以回滾指定數量的遷移。例如，以下命令將回滾最後五個遷移：

```
php artisan migrate:rollback --step=5
```

你可以通過向 `rollback` 命令提供 `batch` 選項來回滾特定的批次遷移，其中 `batch` 選項對應於應用程式中 `migrations` 資料庫表中的一個批次值。例如，下面的命令將回滾第三批中的所有遷移。

```
php artisan migrate:rollback --batch=3
```

命令 `migrate:reset` 會回滾應用已運行過的所有遷移：

```
php artisan migrate:reset
```

44.4.2.1 使用單個命令同時進行回滾和遷移操作

命令 `migrate:refresh` 首先會回滾已運行過的所有遷移，隨後會執行 `migrate`。這一命令可以高效地重建

你的整個資料庫：

```
php artisan migrate:refresh

# 重設資料庫，並運行所有的 seeds...
php artisan migrate:refresh --seed
```

通過在命令 `refresh` 中使用 `step` 參數，你可以回滾並重新執行指定數量的遷移操作。例如，下列命令會回滾並重新執行最後五個遷移操作：

```
php artisan migrate:refresh --step=5
```

44.4.2.2 刪除所有表然後執行遷移

命令 `migrate:fresh` 會刪去資料庫中的所有表，隨後執行命令 `migrate`：

```
php artisan migrate:fresh

php artisan migrate:fresh --seed
```

> **注意**
> 該命令 `migrate:fresh` 在刪去所有資料表的過程中，會無視它們的前綴。如果資料庫涉及到其它應用，使用該命令須十分小心。

44.5 資料表

44.5.1 建立資料表

接下來我們將使用 Schema 的 `create` 方法建立一個新的資料表。`create` 接受兩個參數：第一個參數是表名，而第二個參數是一個閉包，該閉包接受一個用來定義新資料表的 `Blueprint` 對象：

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::create('users', function (Blueprint $table) {
    $table->id();
    $table->string('name');
    $table->string('email');
    $table->timestamps();
});
```

建立表時，可以使用資料庫結構建構器的 列方法 來定義表的列。

44.5.1.1 檢查表 / 列是否存在

你可以使用 `hasTable` 和 `hasColumn` 方法檢查表或列是否存在： if (Schema::hasTable('users')) { // 「users」表存在… }

```
if (Schema::hasColumn('users', 'email')) {
    // 「users」表存在，並且有「email」列...
}
```

44.5.1.2 資料庫連接和表選項

如果要對不是應用程式默認的資料庫連接執行資料庫結構的操作，請使用 `connection` 方法：

```
Schema::connection('sqlite')->create('users', function (Blueprint $table) {
    $table->id();
});
```

此外，還可以使用其他一些屬性和方法來定義表建立的其他地方。使用 MySQL 時，可以使用 `engine` 屬性指定表的儲存引擎：

```
Schema::create('users', function (Blueprint $table) {
    $table->engine = 'InnoDB';

    // ...
});
```

`charset` 和 `collation` 屬性可用於在使用 MySQL 時為建立的表指定字元集和排序規則：

```
Schema::create('users', function (Blueprint $table) {
    $table->charset = 'utf8mb4';
    $table->collation = 'utf8mb4_unicode_ci';

    // ...
});
```

`temporary` 方法可用於將表標識為「臨時」狀態。臨時表僅對當前連接的資料庫 session 可見，當連接關閉時會自動刪除：

```
Schema::create('calculations', function (Blueprint $table) {
    $table->temporary();

    // ...
});
```

如果你想給資料庫表新增「註釋」，你可以在表實例上呼叫 `comment` 方法。目前只有 MySQL 和 Postgres 支援表註釋：

```
Schema::create('calculations', function (Blueprint $table) {
    $table->comment('Business calculations');

    // ...
});
```

44.5.2 更新資料表

`Schema` 門面的 `table` 方法可用於更新現有表。與 `create` 方法一樣，`table` 方法接受兩個參數：表的名稱和接收可用於向表新增列或索引的 `Blueprint` 實例的閉包：

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('users', function (Blueprint $table) {
    $table->integer('votes');
});
```

44.5.3 重新命名 / 刪除表

要重新命名已存在的資料表，使用 `rename` 方法：

```
use Illuminate\Support\Facades\Schema;

Schema::rename($from, $to);
```

要刪除已存在的表，你可以使用 `drop` 或 `dropIfExists` 方法：

```
Schema::drop('users');

Schema::dropIfExists('users');
```

44.5.3.1 使用外部索引鍵重新命名表

在重新命名表之前，應該確認表的所有外部索引鍵約束在遷移檔案中有一個顯式的名稱，而不是讓 Laravel 去指定。否則，外部索引鍵約束名稱將引用舊表名。

44.6 欄位

44.6.1 建立欄位

門面 Schema 的 `table` 方法可用於更新表。與 `create` 方法一樣， `table` 方法接受兩個參數：表名和一個閉包，該閉包接收一個 `illumb\Database\Schema\Blueprint` 實例，可以使用該實例向表中新增列：

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('users', function (Blueprint $table) {
    $table->integer('votes');
});
```

44.6.2 可用的欄位類型

Schema 建構器 `Illuminate\Database\Schema\Blueprint` 提供了多種方法，用來建立表中對應類型的列。下面列出了所有可用的方法：

不列印可用方法

44.6.3 欄位修飾符

除了上面列出的列類型外，在向資料庫表新增列時還有幾個可以使用的「修飾符」。例如，如果要把列設定為要使列為「可空」，你可以使用 `nullable` 方法：

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('users', function (Blueprint $table) {
    $table->string('email')->nullable();
});
```

下表時所有可用的列修飾符。此列表不包括索引修飾符:

修飾符	說明
`after('column')`	將該列放在其它欄位「之後」(MySQL)
`autoIncrement()`	設定 INTEGER 類型的列為自動遞增 (主鍵)
`charset('utf8mb4')`	為該列指定字元集 (MySQL)
`collation('utf8mb4_unicode_ci')`	為該列指定排序規則 (MySQL/PostgreSQL/SQL Server)
`comment('my comment')`	為該列新增註釋 (MySQL/PostgreSQL)
`default($value)`	為該列指定一個「預設值」
`first()`	將該列放在該表「首位」 (MySQL)
`from($integer)`	設定自動遞增欄位的起始值 (MySQL / PostgreSQL)
`invisible()`	使列對「SELECT *」查詢不可見（MySQL）。
`nullable($value = true)`	允許 NULL 值插入到該列
`storedAs($expression)`	建立一個儲存生成的列 (MySQL)
`unsigned()`	設定 INTEGER 類型的欄位為 UNSIGNED (MySQL)
`useCurrent()`	設定 TIMESTAMP 類型的列使用 CURRENT_TIMESTAMP 作為預

修飾符	說明
	設值
`useCurrentOnUpdate()`	將 TIMESTAMP 類型的列設定為在更新時使用 CURRENT_TIMESTAMP 作為新值
`virtualAs($expression)`	建立一個虛擬生成的列 (MySQL)
`generatedAs($expression)`	使用指定的序列選項建立標識列 (PostgreSQL)
`always()`	定義序列值優先於標識列的輸入 (PostgreSQL)
`isGeometry()`	將空間列類型設定為 `geometry` - 默認類型為 `geography` (PostgreSQL)。

44.6.3.1 預設值表示式

`default` 修飾符接收一個變數或者一個 `\Illuminate\Database\Query\Expression` 實例。使用 `Expression` 實例可以避免使用包含在引號中的值，並且允許你使用特定資料庫函數。這在當你需要給 `JSON` 欄位指定預設值的時候特別有用：

```
<?php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Query\Expression;
use Illuminate\Database\Migrations\Migration;

return new class extends Migration
{
    /**
     * 運行遷移
     */
    public function up(): void
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->json('movies')->default(new Expression('(JSON_ARRAY())'));
            $table->timestamps();
        });
    }
};
```

> **注意**
> 支援哪些預設值的表示方式取決於你的資料庫驅動、資料庫版本、還有欄位類型。請參考合適的文件使用。還有一點要注意的是，使用資料庫特定函數，可能會將你綁牢到特定的資料庫驅動上。

44.6.3.2 欄位順序

使用 MySQL 資料庫時，可以使用 `after` 方法在模式中的現有列後新增列：

```
$table->after('password', function (Blueprint $table) {
    $table->string('address_line1');
    $table->string('address_line2');
    $table->string('city');
});
```

44.6.4 修改欄位

`change` 方法可以將現有的欄位類型修改為新的類型或修改屬性。比如，你可能想增加 `string` 欄位的長度，可以使用 `change` 方法把 `name` 欄位的長度從 25 增加到 50。所以，我們可以簡單的更新欄位屬性然後呼叫 `change` 方法：

```
Schema::table('users', function (Blueprint $table) {
```

```
    $table->string('name', 50)->change();
});
```

當修改一個列時，你必須明確包括所有你想在列定義上保留的修改器 —— 任何缺失的屬性都將被丟棄。例如，為了保留 `unsigned`、`default` 和 `comment` 屬性，你必須在修改列時明確每個屬性的修改。

```
Schema::table('users', function (Blueprint $table) {
    $table->integer('votes')->unsigned()->default(1)->comment('my comment')->change();
});
```

44.6.4.1 在 SQLite 上修改列

如果應用程式使用的是 SQLite 資料庫，請確保你已經通過 Composer 包管理器安裝了 `doctrine/dbal` 包。Doctrine DBAL 庫用於確定欄位的當前狀態，並建立對該欄位進行指定調整所需的 SQL 查詢：

```
composer require doctrine/dbal
```

如果你打算修改 `timestamp` 方法來建立列，你還需要將以下組態新增到應用程式的 `config/database.php` 組態檔案中：

```
use Illuminate\Database\DBAL\TimestampType;

'dbal' => [
    'types' => [
        'timestamp' => TimestampType::class,
    ],
],
```

> **注意**
> 當使用 `doctrine/dbal` 包時，你可以修改以下列類型：`bigInteger`、`binary`、`boolean`、`char`、`date`、`dateTime`、`dateTimeTz`、`decimal`、`double`、`integer`、`json`、`longText`、`mediumText`、`smallInteger`、`string`、`text`、`time`、`tinyText`、`unsignedBigInteger`、`unsignedInteger`、`unsignedSmallInteger`、`ulid`、和 `uuid`。

44.6.4.2 重新命名欄位

要重新命名一個列，你可以使用模式建構器提供的 `renameColumn` 方法：

```
Schema::table('users', function (Blueprint $table) {
    $table->renameColumn('from', 'to');
});
```

44.6.4.3 在較低版本資料庫上重新命名列

如果你運行的資料庫低於以下版本，你應該確保在重新命名列之前通過 Composer 軟體包管理器安裝了 `doctrine/dbal` 庫。

- MySQL 版本低於 `8.0.3`
- MariaDB 版本低於 `10.5.2`
- SQLite 版本低於 `3.25.0`

44.6.5 刪除欄位

要刪除一個列，你可以使用 `dropColumn` 方法。

```
Schema::table('users', function (Blueprint $table) {
    $table->dropColumn('votes');
});
```

你可以傳遞一個欄位陣列給 `dropColumn` 方法來刪除多個欄位：

```
Schema::table('users', function (Blueprint $table) {
    $table->dropColumn(['votes', 'avatar', 'location']);
});
```

44.6.5.1 在較低版本的資料庫中刪除列的內容

如果你運行的 SQLite 版本在 `3.35.0` 之前，你必須通過 Composer 軟體包管理器安裝 `doctrine/dbal` 包，然後才能使用 `dropColumn` 方法。不支援在使用該包時在一次遷移中刪除或修改多個列。

44.6.5.2 可用的命令別名

Laravel 提供了幾種常用的刪除相關列的便捷方法。如下表所示：

命令	說明
`$table->dropMorphs('morphable');`	刪除 `morphable_id` 和 `morphable_type` 欄位
`$table->dropRememberToken();`	刪除 `remember_token` 欄位
`$table->dropSoftDeletes();`	刪除 `deleted_at` 欄位
`$table->dropSoftDeletesTz();`	`dropSoftDeletes()` 方法的別名
`$table->dropTimestamps();`	刪除 `created_at` 和 `updated_at` 欄位
`$table->dropTimestampsTz();`	`dropTimestamps()` 方法別名

44.7 索引

44.7.1 建立索引

結構生成器支援多種類型的索引。下面的例子中新建了一個值唯一的 `email` 欄位。我們可以將 `unique` 方法鏈式地新增到欄位定義上來建立索引：

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('users', function (Blueprint $table) {
    $table->string('email')->unique();
});
```

或者，你也可以在定義完欄位之後建立索引。為此，你應該呼叫結構生成器上的 `unique` 方法，此方法應該傳入唯一索引的列名稱：

```
$table->unique('email');
```

你甚至可以將陣列傳遞給索引方法來建立一個復合（或合成）索引：

```
$table->index(['account_id', 'created_at']);
```

建立索引時，Laravel 會自動生成一個合理的索引名稱，但你也可以傳遞第二個參數來自訂索引名稱：

```
$table->unique('email', 'unique_email');
```

44.7.1.1 可用的索引類型

Laravel 的結構生成器提供了 Laravel 支援的所有類型的索引方法。每個索引方法都接受一個可選的第二個參數來指定索引的名稱。如果省略，名稱將根據表和列的名稱生成。下面是所有可用的索引方法：

命令	說明
`$table->primary('id');`	新增主鍵
`$table->primary(['id', 'parent_id']);`	新增復合主鍵
`$table->unique('email');`	新增唯一索引
`$table->index('state');`	新增普通索引

命令	說明
`$table->fullText('body');`	新增全文索引 (MySQL/PostgreSQL)
`$table->fullText('body')->language('english');`	新增指定語言 (PostgreSQL) 的全文索引
`$table->spatialIndex('location');`	新增空間索引（不支援 SQLite）

44.7.1.2 索引長度 & MySQL / MariaDB

默認情況下，Laravel 使用 `utf8mb4` 編碼。如果你是在版本低於 5.7.7 的 MySQL 或者版本低於 10.2.2 的 MariaDB 上建立索引，那你就需要手動組態資料庫遷移的默認字串長度。也就是說，你可以通過在 `App\Providers\AppServiceProvider` 類的 `boot` 方法中呼叫 `Schema::defaultStringLength` 方法來組態默認字串長度：

```
use Illuminate\Support\Facades\Schema;

/**
 * 引導任何應用程式「全域組態」
 */
public function boot(): void
{
    Schema::defaultStringLength(191);
}
```

當然，你也可以選擇開啟資料庫的 `innodb_large_prefix` 選項。至於如何正確開啟，請自行查閱資料庫文件。

44.7.2 重新命名索引

若要重新命名索引，你需要呼叫 `renameIndex` 方法。此方法接受當前索引名稱作為其第一個參數，並將所需名稱作為其第二個參數：

```
$table->renameIndex('from', 'to')
```

> **注意**
> 如果你的應用程式使用的是 SQLite 資料庫，你必須通過 Composer 軟體包管理器安裝 `doctrine/dbal` 包，然後才能使用 `renameIndex` 方法。

44.7.3 刪除索引

若要刪除索引，則必須指定索引的名稱。Laravel 默認會自動將資料表名稱、索引的欄位名及索引類型簡單地連接在一起作為名稱。舉例如下：

命令	說明
`$table->dropPrimary('users_id_primary');`	從 users 表中刪除主鍵
`$table->dropUnique('users_email_unique');`	從 users 表中除 unique 索引
`$table->dropIndex('geo_state_index');`	從 geo 表中刪除基本索引
`$table->dropFullText('posts_body_fulltext');`	從 post 表中刪除一個全文索引
`$table->dropSpatialIndex('geo_location_spatialindex');`	從 geo 表中刪除空間索引（不支援 SQLite）

如果將欄位陣列傳給 `dropIndex` 方法，會刪除根據表名、欄位和鍵類型生成的索引名稱。

```
Schema::table('geo', function (Blueprint $table) {
    $table->dropIndex(['state']); // 刪除 'geo_state_index' 索引
});
```

44.7.4 外部索引鍵約束

Laravel 還支援建立用於在資料庫層中的強制引用完整性的外部索引鍵約束。例如，讓我們在 `posts` 表上定義一個引用 `users` 表的 `id` 欄位的 `user_id` 欄位：

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('posts', function (Blueprint $table) {
    $table->unsignedBigInteger('user_id');

    $table->foreign('user_id')->references('id')->on('users');
});
```

由於這種外部索引鍵約束的定義方式過於繁複，Laravel 額外提供了更簡潔的方法，基於約定來提供更好的開發人員體驗。當使用 `foreignId` 方法來建立列時，上面的示例還可以這麼寫：

```
Schema::table('posts', function (Blueprint $table) {
    $table->foreignId('user_id')->constrained();
});
```

`foreignId` 方法是 `unsignedBigInteger` 的別名，而 `constrained` 方法將使用約定來確定所引用的表名和列名。如果表名與約定不匹配，可以通過將表名作為參數傳遞給 `constrained` 方法來指定表名：

```
Schema::table('posts', function (Blueprint $table) {
    $table->foreignId('user_id')->constrained('users');
});
```

你可以為約束的「on delete」和「on update」屬性指定所需的操作：

```
$table->foreignId('user_id')
      ->constrained()
      ->onUpdate('cascade')
      ->onDelete('cascade');
```

還為這些操作提供了另一種表達性語法：

方法	說明
`$table->cascadeOnUpdate();`	更新應該級聯
`$table->restrictOnUpdate();`	應該限制更新
`$table->cascadeOnDelete();`	刪除應該級聯
`$table->restrictOnDelete();`	應該限制刪除
`$table->nullOnDelete();`	刪除應將外部索引鍵值設定為空

當使用任意 欄位修飾符 的時候，必須在呼叫 `constrained` 之前呼叫：

```
$table->foreignId('user_id')
      ->nullable()
      ->constrained();
```

44.7.4.1 刪除外部索引鍵

要刪除一個外部索引鍵，你需要使用 `dropForeign` 方法，將要刪除的外部索引鍵約束作為參數傳遞。外部索引鍵約束採用的命名方式與索引相同。即，將資料表名稱和約束的欄位連接起來，再加上 `_foreign` 後綴：

```
$table->dropForeign('posts_user_id_foreign');
```

或者，可以給 `dropForeign` 方法傳遞一個陣列，該陣列包含要刪除的外部索引鍵的列名。陣列將根據 Laravel 的 結構生成器使用的約束名稱約定自動轉換：

```
$table->dropForeign(['user_id']);
```

44.7.4.2 更改外部索引鍵約束

你可以在遷移檔案中使用以下方法來開啟或關閉外部索引鍵約束：

```
Schema::enableForeignKeyConstraints();

Schema::disableForeignKeyConstraints();

Schema::withoutForeignKeyConstraints(function () {
    // 閉包中停用的約束…
});
```

> 注意：SQLite 默認停用外部索引鍵約束。使用 SQLite 時，請確保在資料庫組態中啟用外部索引鍵支援然後再嘗試在遷移中建立它們。另外，SQLite 只在建立表時支援外部索引鍵，並且將在修改表時不會支援。

44.8 事件

為方便起見，每個遷移操作都會派發一個 事件。以下所有事件都擴展了基礎 `Illuminate\Database\Events\MigrationEvent` 類：

類	描述
`Illuminate\Database\Events\MigrationsStarted`	即將執行一批遷移
`Illuminate\Database\Events\MigrationsEnded`	一批遷移已完成執行
`Illuminate\Database\Events\MigrationStarted`	即將執行單個遷移
`Illuminate\Database\Events\MigrationEnded`	單個遷移已完成執行
`Illuminate\Database\Events\SchemaDumped`	資料庫結構轉儲已完成
`Illuminate\Database\Events\SchemaLoaded`	已載入現有資料庫結構轉儲

45 資料填充

45.1 簡介

Laravel 內建了一個可為你的資料庫填充測試資料的資料填充類。所有的資料填充類都應該放在 `database/seeds` 目錄下。Laravel 默認定義了一個 `DatabaseSeeder` 類。通過這個類，你可以用 `call` 方法來運行其他的 `seed` 類，從而控制資料填充的順序。

> **注意**
> 在資料庫填充期間，批次賦值保護被自動停用。

45.2 編寫 Seeders

運行 Artisan 命令 `make:seeder` 可以生成 Seeder，生成的 seeders 都放在 `database/seeders` 目錄下：

```
php artisan make:seeder UserSeeder
```

一個資料填充類默認只包含一個方法：`run` ，當執行 `db:seed` Artisan 命令 時，被呼叫。在 `run` 方法中, 可以按需將資料插入資料庫中。 也可以使用查詢構造器來手動插入資料，或者可以使用 Eloquent 資料工廠。

例如，在默認 `DatabaseSeeder` 類的 `run` 方法中新增一個資料庫插入語句：

```
<?php

namespace Database\Seeders;

use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Str;

class DatabaseSeeder extends Seeder
{
    /**
     * 運行資料填充
     */
    public function run(): void
    {
        DB::table('users')->insert([
            'name' => Str::random(10),
            'email' => Str::random(10).'@gmail.com',
            'password' => Hash::make('password'),
        ]);
    }
}
```

> **注意**
> 可以在 `run` 方法的參數中鍵入你需要的任何依賴性,它們將自動通過 Laravel 服務容器注入。

45.2.1 使用模型工廠

當然，手動指定每個模型填充的屬性是很麻煩的。因此可以使用 Eloquent 資料工廠來更方便地生成大量的資料庫記錄。首先，查看 Eloquent 資料工廠，瞭解如何定義工廠。

例如，建立 50 個使用者，每個使用者有一個相關的帖子：

```
use App\Models\User;

/**
 * 運行資料庫遷移
 */
public function run(): void
{
    User::factory()
            ->count(50)
            ->hasPosts(1)
            ->create();
}
```

45.2.2 呼叫其他 Seeders

在 `DatabaseSeeder` 類中，可以使用 `call` 方法來執行其他的填充類。使用 `call` 方法可以將資料庫遷移分成多個檔案，這樣就不會出現單個資料填充類過大。`call` 方法接受一個由資料填充類組成的陣列：

```
/**
 * 運行資料庫遷移
 */
public function run(): void
{
    $this->call([
        UserSeeder::class,
        PostSeeder::class,
        CommentSeeder::class,
    ]);
}
```

45.2.3 停用模型事件

在執行階段，你可能想阻止模型呼叫事件，可以使用 `WithoutModelEvents` 特性。使用 `WithoutModelEvents` trait 可確保不呼叫模型事件，即使通過 `call` 方法執行了額外的 seed 類：

```
<?php

namespace Database\Seeders;

use Illuminate\Database\Seeder;
use Illuminate\Database\Console\Seeds\WithoutModelEvents;

class DatabaseSeeder extends Seeder
{
    use WithoutModelEvents;

    /**
     * 運行資料庫遷移
     */
    public function run(): void
    {
        $this->call([
            UserSeeder::class,
        ]);
    }
}
```

45.3 運行 Seeders

執行 `db:seed` Artisan 命令來為資料庫填充資料。默認情況下，`db:seed` 命令會運行 `Database\Seeders\DatabaseSeeder` 類來呼叫其他資料填充類。當然，也可以使用 `--class` 選項來指定一個特定的填充類：

```
php artisan db:seed

php artisan db:seed --class=UserSeeder
```

你還可以使用 `migrate:fresh` 命令結合 `--seed` 選項，這將刪除資料庫中所有表並重新運行所有遷移。此命令對於完全重建資料庫非常有用。 `--seeder` 選項可以用來指定要運行的填充檔案：

```
php artisan migrate:fresh --seed

php artisan migrate:fresh --seed --seeder=UserSeeder
```

45.3.1.1 在生產環境中強制運行填充

一些填充操作可能會導致原有資料的更新或丟失。為了保護生產環境資料庫的資料，在 **生產環境** 中運行填充命令前會進行確認。可以新增 `--force` 選項來強制運行填充命令：

```
php artisan db:seed --force
```

46 Redis

46.1 簡介

Redis 是一個開放原始碼的, 高級鍵值對儲存資料庫。 保護的資料庫類型有

- 字串
- hash
- 列表
- 集合
- 有序集合

在將 Redis 與 Laravel 一起使用前，我們鼓勵你通過 PECL 安裝並使用 PhpRedis ，儘管擴展安裝起來更複雜，但對於大量使用 Redis 的應用程式可能會帶來更好的性能。如果你使用 Laravel Sail, 這個擴展已經事先在你的 Docker 容器中安裝完成。

如果你不能安裝 PHPRedis 擴展，你或許可以使用 composer 安裝 predis/predis 包。Predis 是一個完全用 PHP 編寫的 Redis 客戶端，不需要任何額外的擴展：

```
composer require predis/predis
```

46.2 組態

在你的應用中組態 Redis 資訊，你要在 `config/database.php` 檔案中進行組態。在該檔案中，你將看到一個 `Redis` 陣列包含了你的 Redis 組態資訊。

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

    'default' => [
        'host' => env('REDIS_HOST', '127.0.0.1'),
        'password' => env('REDIS_PASSWORD'),
        'port' => env('REDIS_PORT', 6379),
        'database' => env('REDIS_DB', 0),
    ],

    'cache' => [
        'host' => env('REDIS_HOST', '127.0.0.1'),
        'password' => env('REDIS_PASSWORD'),
        'port' => env('REDIS_PORT', 6379),
        'database' => env('REDIS_CACHE_DB', 1),
    ],

],
```

在你的組態檔案裡定義的每個 Redis 伺服器，除了用 URL 來表示的 Redis 連接，都必需要指定名稱 、 host （主機）和 port （連接埠）欄位：

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

    'default' => [
        'url' => 'tcp://127.0.0.1:6379?database=0',
    ],

    'cache' => [
```

```
        'url' => 'tls://user:password@127.0.0.1:6380?database=1',
    ],

],
```

46.2.1.1 組態連接方案

默認情況下，Redis 客戶端使用 `tcp` 方案連接 Redis 伺服器。另外，你也可以在你的 Redis 服務組態陣列中指定一個 `scheme` 組態項，來使用 TLS/SSL 加密：

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

    'default' => [
        'scheme' => 'tls',
        'host' => env('REDIS_HOST', '127.0.0.1'),
        'password' => env('REDIS_PASSWORD'),
        'port' => env('REDIS_PORT', 6379),
        'database' => env('REDIS_DB', 0),
    ],

],
```

46.2.2 叢集

如果你的應用使用 Redis 叢集，你應該在 Redis 組態檔案中用 `clusters` 鍵來定義叢集。這個組態鍵默認沒有，所以你需要在 `config/database.php` 組態檔案中手動建立：

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

    'clusters' => [
        'default' => [
            [
                'host' => env('REDIS_HOST', 'localhost'),
                'password' => env('REDIS_PASSWORD'),
                'port' => env('REDIS_PORT', 6379),
                'database' => 0,
            ],
        ],
    ],

],
```

默認情況下，叢集可以在節點上實現客戶端分片，允許你實現節點池以及建立大量可用記憶體。這裡要注意，客戶端共享不會處理失敗的情況；因此，這個功能主要適用於從另一個主資料庫獲取的快取資料。 如果要使用 Redis 原生叢集，需要把 `config/database.php` 組態檔案下的 `options.cluster` 組態項的值設定為 `redis`：

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

    'options' => [
        'cluster' => env('REDIS_CLUSTER', 'redis'),
    ],

    'clusters' => [
        // ...
    ],
```

```
],
```

46.2.3 Predis

要使用 Predis 擴展去連接 Redis， 請確保環境變數 `REDIS_CLIENT` 的值為 `predis` ：

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'predis'),

    // ...
],
```

除默認的 `host` ，`port` ，`database` 和 `password` 這些服務組態選項外， Predis 還支援為每個 Redis 伺服器定義其它的 連接參數。如果要使用這些額外的組態項，可以在 config/database.php 組態檔案中將任意選項新增到 Redis 伺服器組態內：

```
'default' => [
    'host' => env('REDIS_HOST', 'localhost'),
    'password' => env('REDIS_PASSWORD'),
    'port' => env('REDIS_PORT', 6379),
    'database' => 0,
    'read_write_timeout' => 60,
],
```

46.2.3.1 Redis Facade 別名

Laravel 的 `config/app.php` 組態檔案包含了 `aliases` 陣列，該陣列可用於定義通過框架註冊的所有類別名。方便起見，Laravel 提供了一份包含了所有 facade 的別名入口；不過，`Redis` 別名不能在這裡使用，因為這與 phpredis 擴展提供的 `Redis` 類名衝突。如果正在使用 `Predis` 客戶端並確實想要用這個別名，你可以在 `config/app.php` 組態檔案中取消對此別名的註釋。

```
'aliases' => Facade::defaultAliases()->merge([
    'Redis' => Illuminate\Support\Facades\Redis::class,
])->toArray(),
```

46.2.4 phpredis

Laravel 默認使用 phpredis 擴展與 Redis 通訊。Laravel 用於與 Redis 通訊的客戶端由 `redis.client` 組態項決定，這個組態通常為環境變數 `REDIS_CLIENT` 的值：

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

    // 重設 Redis 組態項...
],
```

除默認的 `scheme` , `host` , `port`, `database` 和 `password` 的伺服器組態選項外，`phpredis` 還支援以下額外的連接參數：`name` , `persistent`, `persistent_id`, `prefix`, `read_timeout`, `retry_interval`, `timeout` 和 `context`。 你可以在 `config/database.php` 組態檔案中將任意選項新增到 Redis 伺服器組態內：

```
'default' => [
    'host' => env('REDIS_HOST', 'localhost'),
    'password' => env('REDIS_PASSWORD'),
    'port' => env('REDIS_PORT', 6379),
    'database' => 0,
    'read_timeout' => 60,
    'context' => [
        // 'auth' => ['username', 'secret'],
        // 'stream' => ['verify_peer' => false],
    ],
```

```
],
```

46.2.4.1 phpredis 序列化和壓縮

phpredis 擴展可以組態使用各種序列化和壓縮演算法。可以通過設定 Redis 組態中的 `options` 陣列進行組態：

```
'redis' => [

    'client' => env('REDIS_CLIENT', 'phpredis'),

    'options' => [
        'serializer' => Redis::SERIALIZER_MSGPACK,
        'compression' => Redis::COMPRESSION_LZ4,
    ],

    // 重設 Redis 組態項...
],
```

當前支援的序列化演算法包括： `Redis::SERIALIZER_NONE` （默認）, `Redis::SERIALIZER_PHP` , `Redis::SERIALIZER_JSON` , `Redis::SERIALIZER_IGBINARY` 和 `Redis::SERIALIZER_MSGPACK` 。

支援的壓縮演算法包括： `Redis::COMPRESSION_NONE` （默認）, `Redis::COMPRESSION_LZF` , `Redis::COMPRESSION_ZSTD` 和 `Redis::COMPRESSION_LZ4` 。

46.3 與 Redis 互動

你可以通過呼叫 `Redis` facade 上的各種方法來與 Redis 進行互動。 `Redis` facade 支援動態方法，所以你可以在 facade 上呼叫各種 Redis 命令 ,這些命令將直接傳遞給 Redis 。 在本例中，我們將呼叫 `Redis` facade 的 `get` 方法，來呼叫 Redis `GET` 方法：

```
<?php

namespace App\Http\Controllers;

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

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

如上所述，你可以在 `Redis` facade 上呼叫任意 Redis 命令。 Laravel 使用魔術方法將命令傳遞給 Redis 伺服器。如果一個 Redis 命令需要參數，則應將這些參數傳遞給 `Redis` facade 的相應方法：

```
use Illuminate\Support\Facades\Redis;

Redis::set('name', 'Taylor');

$values = Redis::lrange('names', 5, 10);
```

或者，你也可以使用 `Redis` facade 上的 `command` 方法將命令傳遞給伺服器，它接受命令的名稱作為其第一個參數，並將值的陣列作為其第二個參數：

```
$values = Redis::command('lrange', ['name', 5, 10]);
```

46.3.1.1 使用多個 Redis 連接

你應用裡的 `config/database.php` 組態檔案允許你去定義多個 Redis 連接或者伺服器。你可以使用 `Redis` facade 上的 `connection` 方法獲得指定的 Redis 連接：

```
$redis = Redis::connection('connection-name');
```

要獲取獲取一個默認的 Redis 連接，你可以呼叫 `connection` 方法時，不帶任何參數：

```
$redis = Redis::connection();
```

46.3.2 事務

`Redis` facade 上的 `transaction` 方法對 Redis 原生的 `MULTI` 和 `EXEC` 命令進行了封裝。 `transaction` 方法接受一個閉包作為其唯一參數。這個閉包將接收一個 Redis 連接實例，並可能向這個實例發出想要的任何命令。閉包中發出的所有 Redis 命令都將在單個原子性事務中執行：

```
use Redis;
use Illuminate\Support\Facades;

Facades\Redis::transaction(function (Redis $redis) {
    $redis->incr('user_visits', 1);
    $redis->incr('total_visits', 1);
});
```

> **注意**
> 定義一個 Redis 事務時，你不能從 Redis 連接中獲取任何值。請記住，事務是作為單個原子性操作執行的，在整個閉包執行完其命令之前，不會執行該操作。

46.3.2.1 Lua 指令碼

`eval` 方法提供了另外一種原子性執行多條 Redis 命令的方式。但是，`eval` 方法的好處是能夠在操作期間與 Redis 鍵值互動並檢查它們。 Redis 指令碼是用 Lua 程式語言 編寫的。

`eval` 方法一開始可能有點令人勸退，所以我們將用一個基本示例來明確它的使用方法。 `eval` 方法需要幾個參數。第一，在方法中傳遞一個 Lua 指令碼（作為一個字串）。第二，在方法中傳遞指令碼互動中用到的鍵的數量（作為一個整數）。第三，在方法中傳遞所有鍵名。最後，你可以傳遞一些指令碼中用到的其他參數。

在本例中，我們要對第一個計數器進行遞增，檢查它的新值，如果該計數器的值大於 5，那麼遞增第二個計數器。最終，我們將返回第一個計數器的值：

```
$value = Redis::eval(<<<'LUA'
    local counter = redis.call("incr", KEYS[1])

    if counter > 5 then
        redis.call("incr", KEYS[2])
    end

    return counter
LUA, 2, 'first-counter', 'second-counter');
```

> **注意**
> 請參考 Redis 文件 更多關於 Redis 指令碼的資訊。

46.3.3 管道命令

當你需要執行很多個 Redis 命令時，你可以使用 `pipeline` 方法一次性提交所有命令，而不需要每條命令都與 Redis 伺服器建立一次網路連線。 `pipeline` 方法只接受一個參數：接收一個 Redis 實例的閉包。你可以將所有

命令發給這個 Redis 實例，它們將同時傳送到 Redis 伺服器，以減少到伺服器的網路訪問。這些命令仍然會按照發出的順序執行：

```
use Redis;
use Illuminate\Support\Facades;

Facades\Redis::pipeline(function (Redis $pipe) {
    for ($i = 0; $i < 1000; $i++) {
        $pipe->set("key:$i", $i);
    }
});
```

46.4 發佈 / 訂閱

Laravel 為 Redis 的 `publish` 和 `subscribe` 命令提供了方便的介面。你可以用這些 Redis 命令監聽指定「頻道」上的消息。你也可以從一個應用程式發消息給另一個應用程式，哪怕它是用其它程式語言開發的，讓應用程式和處理程序之間能夠輕鬆進行通訊。

首先，用 `subscribe` 方法設定一個頻道監聽器。我們將這個方法呼叫放到一個 Artisan 命令 中，因為呼叫 `subscribe` 方法會啟動一個常駐處理程序：

```
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;
use Illuminate\Support\Facades\Redis;

class RedisSubscribe extends Command
{
    /**
     * 控制台命令的名稱和簽名
     *
     * @var string
     */
    protected $signature = 'redis:subscribe';

    /**
     * 控制台命令的描述
     *
     * @var string
     */
    protected $description = 'Subscribe to a Redis channel';

    /**
     * 執行控制台命令
     */
    public function handle(): void
    {
        Redis::subscribe(['test-channel'], function (string $message) {
            echo $message;
        });
    }
}
```

現在我們可以使用 `publish` 方法將消息發佈到頻道：

```
use Illuminate\Support\Facades\Redis;

Route::get('/publish', function () {
    // ...

    Redis::publish('test-channel', json_encode([
        'name' => 'Adam Wathan'
```

```
    ]));
});
```

46.4.1.1 萬用字元訂閱

使用 `psubscribe` 方法，你可以訂閱一個萬用字元頻道，用來獲取所有頻道中的所有消息，頻道名稱將作為第二個參數傳遞給提供的回呼閉包：

```
Redis::psubscribe(['*'], function (string $message, string $channel) {
    echo $message;
});

Redis::psubscribe(['users.*'], function (string $message, string $channel) {
    echo $message;
});
```

47 Eloquent 快速入門

47.1 簡介

Laravel 包含的 Eloquent 模組，是一個對象關係對應(ORM)，能使你更愉快地互動資料庫。當你使用 Eloquent 時，資料庫中每張表都有一個相對應的”模型”用於操作這張表。除了能從資料表中檢索資料記錄之外，Eloquent 模型同時也允許你新增，更新和刪除這對應表中的資料

> **注意**
> 開始使用之前, 請確認在你的項目裡的 `config/database.php` 組態檔案中已經組態好一個可用的資料庫連接. 關於組態資料庫的更多資訊, 請查閱資料庫組態文件.

47.1.1.1 Laravel 訓練營

如果你是 Laravel 的新手，可以隨時前往 Laravel 訓練營。Laravel 訓練營將指導你使用 Eloquent 建立你的第一個 Laravel 應用。這是一個很好的方式來瞭解 Laravel 和 Eloquent 所提供的一切。

47.2 生成模型類

首先，讓我們建立一個 Eloquent 模型。模型通常位於 `app\Models` 目錄中，並繼承 `Illuminate\Database\Eloquent\Model` 類。 你可以使用 `make:model` Artisan 命令 來生成新模型類：

```
php artisan make:model Flight
```

如果你想要在生成模型類的同時生成 資料庫遷移， 可以使用 `--migration` 或 `-m` 選項：

```
php artisan make:model Flight --migration
```

在生成模型的同時，你可能還想要各種其他類型的類，例如模型工廠、資料填充和 controller 。這些選項可以組合在一起從而一次建立多個類：

```
# 生成模型和 Flight 工廠類...
php artisan make:model Flight --factory
php artisan make:model Flight -f

# 生成模型和 Flight 資料填充類...
php artisan make:model Flight --seed
php artisan make:model Flight -s

# 生成模型和 Flight  controller 類...
php artisan make:model Flight --controller
php artisan make:model Flight -c

# 生成模型，Flight  controller 類，資源類和表單驗證類...
php artisan make:model Flight --controller --resource --requests
php artisan make:model Flight -crR

# 生成模型和 Flight 授權策略類...
php artisan make:model Flight --policy

# 生成模型和資料庫遷移，Filght 工廠類，資料庫填充類和 Flight  controller ...
php artisan make:model Flight -mfsc

# 快捷生成模型，資料庫遷移，Flight 工廠類，資料庫填充類，授權策略類，Flight  controller 和表單驗證類...
php artisan make:model Flight --all
```

```
# 生成中間表模型...
php artisan make:model Member --pivot
```

47.2.1.1 檢查模型

有時，僅僅通過略讀程式碼來確定一個模型的所有可用屬性和關係是很困難的。作為替代，試試 `model:show` Artisan 命令，它提供了一個對於模型的所有屬性和關係的方便概述。

```
php artisan model:show Flight
```

47.3 Eloquent 模型約定

由 `make:model` 命令生成的模型會被放置在 `app/Models` 目錄下。讓我們檢查一個基本的模型類並討論 Eloquent 的一些關鍵約定：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    // ...
}
```

47.3.1 資料表名稱

看了上面的例子，你可能已經注意到我們沒有告訴 Eloquent 哪個資料庫表對應我們的 `Flight` 模型。按照約定，除非明確指定另一個名稱，類名稱的下劃線格式的複數形態將被用作表名。因此，在這個例子中，Eloquent 將假定 `Flight` 模型將記錄儲存在 `flights` 表中，而 `AirTrafficController` 模型將記錄儲存在 `air_traffic_controllers` 表中。

如果你的模型對應的資料表不符合這個約定，你可以通過在模型上定義一個 `table` 屬性來手動指定模型的表名：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * 與模型關聯的資料表。
     *
     * @var string
     */
    protected $table = 'my_flights';
}
```

47.3.2 主鍵

Eloquent 還會假設每個模型對應的資料表都有一個名為 `id` 的列作為主鍵。如有必要，你可以在模型上定義一個受保護的 `$primaryKey` 屬性，來指定一個不同的列名稱用作模型的主鍵：

```
<?php
```

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * 與資料表關聯的主鍵。
     *
     * @var string
     */
    protected $primaryKey = 'flight_id';
}
```

此外，Eloquent 默認有一個 integer 值的主鍵，Eloquent 會自動轉換這個主鍵為一個 integer 類型，如果你的主鍵不是自增或者不是數字類型，你可以在你的模型上定義一個 public 屬性的 `$incrementing` ，並將其設定為 `false`：

```
<?php

class Flight extends Model
{
    /**
     * 指明模型的 ID 是否自動遞增。
     *
     * @var bool
     */
    public $incrementing = false;
}
```

如果你的模型主鍵不是 `integer`，應該定義一個 `protected $keyType` 屬性在模型上，其值應為 `string`：

```
<?php

class Flight extends Model
{
    /**
     * 自動遞增 ID 的資料類型。
     *
     * @var string
     */
    protected $keyType = 'string';
}
```

47.3.2.1 復合主鍵

Eloquent 要求每個模型至少有一個可以作為其主鍵的唯一標識 ID。它不支援「復合」主鍵。但是，除了表的唯一標識主鍵之外，還可以向資料庫表新增額外的多列唯一索引。

47.3.3 UUID 與 ULID 鍵

你可以選擇使用 UUID，而不是使用自動遞增的整數作為 Elquent 模型的主鍵。UUID 是 36 個字元長的通用唯一字母數字識別碼。

如果你希望模型使用 UUID 鍵而不是自動遞增的整數鍵，可以在模型上使用 `Illuminate\Database\Eloquent\Concerns\HasUuids` trait，在此情況下應該確保模型具有 UUID 相等的主鍵列:

```
use Illuminate\Database\Eloquent\Concerns\HasUuids;
use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
```

```
    use HasUuids;

    // ...
}

$article = Article::create(['title' => 'Traveling to Europe']);

$article->id; // "8f8e8478-9035-4d23-b9a7-62f4d2612ce5"
```

默認情況下，`HasUuids` trait 將會為模型生成 「ordered」 UUIDs 。 這些 UUIDs 對於索引資料庫儲存更有效，因為它們可以按字典順序進行排序。

通過在模型中定義一個 `newUniqueId` 方法，你可以推翻給定模型的 UUID 生成方法。此外，你可以通過模型中的 `uniqueIds` 方法，來指定哪個欄位是需要接收 UUIDs:

```
use Ramsey\Uuid\Uuid;

/**
 * 為模型生成一個新的 UUID。
 */
public function newUniqueId(): string
{
    return (string) Uuid::uuid4();
}

/**
 * 獲取應該接收唯一識別碼的列。
 *
 * @return array<int, string>
 */
public function uniqueIds(): array
{
    return ['id', 'discount_code'];
}
```

如果你願意，你可以選擇利用 「ULIDs」 來替代 UUIDs。 ULIDs 類似於 UUIDs； 然而，它們的長度僅為 26 字元。類似於訂單 UUIDs， ULIDs 是字典順序排序，以實現高效的資料索引。為了利用 ULIDs， 你需要在你的模型中引用 `Illuminate\Database\Eloquent\Concerns\HasUlids` trait。 同樣還需要確保模型中有一個 ULID 匹配的主鍵欄位:

```
use Illuminate\Database\Eloquent\Concerns\HasUlids;
use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
    use HasUlids;

    // ...
}

$article = Article::create(['title' => 'Traveling to Asia']);

$article->id; // "01gd4d3tgrrfqeda94gdbtdk5c"
```

47.3.4 時間戳

默認情況下，Eloquent 需要 `created_at` 和 `updated_at` 欄位存在你的模型資料表中。當模型被建立或更新時，Eloquent 將自動地設定這些欄位的值。如果你不想讓這些欄位被 Eloquent 自動管理，你需要在你的模型中定義一個 `$timestamps` 屬性並賦值為 `false`:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
```

```
class Flight extends Model
{
    /**
     * 指示模型是否主動維護時間戳。
     *
     * @var bool
     */
    public $timestamps = false;
}
```

如果你需要自訂模型時間戳的格式，請在模型上設定 `$dateFormat` 屬性。以此來定義時間戳在資料庫中的儲存方式以及模型序列化為陣列或 JSON 時的格式：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * 模型日期欄位的儲存格式。
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

如果需要自訂用於儲存時間戳的欄位的名稱，可以在模型上定義 `CREATED_AT` 和 `UPDATED_AT` 常數：

```
<?php

class Flight extends Model
{
    const CREATED_AT = 'creation_date';
    const UPDATED_AT = 'updated_date';
}
```

如果你想在不修改模型的 `updated_at` 時間戳的情況下執行模型操作，你可以在給 `withoutTimestamps` 方法的閉包中對模型進行操作：

```
Model::withoutTimestamps(fn () => $post->increment(['reads']));
```

47.3.5 資料庫連接

默認情況下，所有 Eloquent 模型使用的是應用程式組態的默認資料庫連接。如果想指定在與特定模型互動時應該使用的不同連接，可以在模型上定義 `$connection` 屬性：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * 設定當前模型使用的資料庫連接名。
     *
     * @var string
     */
    protected $connection = 'sqlite';
}
```

47.3.6 默認屬性值

默認情況下，被實例化的模型不會包含任何屬性值。如果你想為模型的某些屬性定義預設值，可以在模型上定義一個 $attributes 屬性。放在 $attributes 陣列中的屬性值應該是原始的，“可儲存的”格式，就像它們剛剛從資料庫中讀取一樣：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * 模型的屬性預設值。
     *
     * @var array
     */
    protected $attributes = [
        'options' => '[]',
        'delayed' => false,
    ];
}
```

47.3.7 組態嚴格 Eloquent

Laravel 提供了幾種方法允許你在各種情況下組態 Eloquent 的行為和其「嚴格性」。

首先，preventLazyLoading 方法接受一個可選的布林值參數，它代表是否需要停用延遲載入。例如，你可能希望僅在非生產環境下停用延遲載入，以便即使在生產環境中的程式碼意外出現延遲載入關係，你的生產環境也可以繼續正常運行。一般來說，該方法應該在應用程式的 AppServiceProvider 的 boot 方法中呼叫：

```
use Illuminate\Database\Eloquent\Model;

/**
 * 啟動任意應用程式服務。
 */
public function boot(): void
{
    Model::preventLazyLoading(! $this->app->isProduction());
}
```

此外，你可以通過呼叫 preventSilentlyDiscardingAttributes 方法來讓 Laravel 在使用嘗試填充一個不能填充的屬性的時候拋出一個異常。這有助於防止在本地開發過程中嘗試設定尚未到模型的 fillable 陣列中的屬性時出現意外情況：

```
Model::preventSilentlyDiscardingAttributes(! $this->app->isProduction());
```

最後，在你嘗試訪問模型上的一個無法從資料庫中檢索到或是該屬性不存在的時候，你可能想要讓 Eloquent 拋出一個異常。例如，當你忘記將屬性新增到 Eloquent 查詢的 select 子句時候，便可能發生這樣的情況。

```
Model::preventAccessingMissingAttributes(! $this->app->isProduction());
```

47.3.7.1 啟用 Eloquent 的嚴格模式

為了方便，你可以通過呼叫 shouldBeStrict 方法來啟用上述的三種方法：

```
Model::shouldBeStrict(! $this->app->isProduction());
```

47.4 檢索模型

一旦你建立了一個模型和 其關聯的資料庫表，就可以開始從資料庫中檢索資料了。可以將每個 Eloquent 模型視為一個強大的查詢建構器 ，讓你能流暢地查詢與該模型關聯的資料庫表。模型中的 `all` 方法將從模型的關聯資料庫表中檢索所有記錄：

```
use App\Models\Flight;

foreach (Flight::all() as $flight) {
    echo $flight->name;
}
```

47.4.1.1 建構查詢

Eloquent 的 `all` 方法會返回模型中所有的結果。由於每個 Eloquent 模型都可以被視為查詢建構器，可以新增額外的查詢條件，然後使用 `get` 方法獲取查詢結果：

```
$flights = Flight::where('active', 1)
                ->orderBy('name')
                ->take(10)
                ->get();
```

> **技巧**
> 由於 Eloquent 模型是查詢建構器，因此你應該查看 Laravel 的查詢建構器提供的所有方法。在編寫 Eloquent 查詢時，這些是通用的。

47.4.1.2 刷新模型

如果已經有一個從資料庫中檢索到的 Eloquent 模型的實例，你可以使用 `fresh` 和 `refresh` 方法「刷新」模型。 `fresh` 方法將從資料庫中重新檢索模型。現有模型實例不會受到影響：

```
$flight = Flight::where('number', 'FR 900')->first();

$freshFlight = $flight->fresh();
```

`refresh` 方法會使用資料庫中的新資料重新賦值現有的模型。此外，已經載入的關係也會被重新載入：

```
$flight = Flight::where('number', 'FR 900')->first();

$flight->number = 'FR 456';

$flight->refresh();

$flight->number; // "FR 900"
```

47.4.2 集合

正如我們所見，像 `all` 和 `get` 這樣的 Eloquent 方法從資料庫中檢索出多條記錄。但是，這些方法不會返回一個普通的 PHP 陣列。相反，會返回一個 `Illuminate\Database\Eloquent\Collection` 的實例。

Eloquent `Collection` 類擴展了 Laravel 的 `Illuminate\Support\Collection` 基類，它提供了大量的輔助方法來與資料集合互動。例如，`reject` 方法可用於根據呼叫閉包的結果從集合中刪除模型：

```
$flights = Flight::where('destination', 'Paris')->get();

$flights = $flights->reject(function (Flight $flight) {
    return $flight->cancelled;
});
```

除了 Laravel 的基礎集合類提供的方法之外，Eloquent 集合類還提供了一些額外的方法，專門用於與 Eloquent 的

模型。

由於 Laravel 的所有集合都實現了 PHP 的可迭代介面，因此你可以像陣列一樣循環遍歷集合：

```
foreach ($flights as $flight) {
    echo $flight->name;
}
```

47.4.3 結果分塊

如果你嘗試通過 `all` 或 `get` 方法載入數萬條 Eloquent 記錄，你的應用程式可能會耗盡記憶體。為了避免出現這種情況， `chunk` 方法可以用來更有效地處理這些大量資料。

`chunk` 方法將傳遞 Eloquent 模型的子集，將它們交給閉包進行處理。由於一次只檢索當前的 Eloquent 模型塊的資料，所以當處理大量模型資料時， `chunk` 方法將顯著減少記憶體使用：

```
use App\Models\Flight;
use Illuminate\Database\Eloquent\Collection;

Flight::chunk(200, function (Collection $flights) {
    foreach ($flights as $flight) {
        // ...
    }
});
```

傳遞給 `chunk` 方法的第一個參數是每個分塊檢索的資料數量。第二個參數傳遞的閉包將方法將應用到每個分塊，以資料庫中查詢到的分塊結果來作為參數。

如果要根據一個欄位來過濾 `chunk` 方法拿到的資料，同時，這個欄位的資料在遍歷的時候還需要更新的話，那麼可以使用「chunkById」方法。在這種場景下如果使用 `chunk` 方法的話，得到的結果可能和預想中的不一樣。在 `chunkById` 方法的內部，默認會查詢 id 欄位大於前一個分塊中最後一個模型的 id。

```
Flight::where('departed', true)
    ->chunkById(200, function (Collection $flights) {
        $flights->each->update(['departed' => false]);
    }, $column = 'id');
```

47.4.4 使用惰性集合進行分塊

`lazy` 方法的工作方式類似於 [chunk 方法](#)，因為它在後台以塊的形式執行查詢。 然而，`lazy` 方法不是將每個塊直接傳遞到回呼中，而是返回 Eloquent 模型的扁平化 [LazyCollection](#)，它可以讓你將結果作為單個流進行互動：

```
use App\Models\Flight;

foreach (Flight::lazy() as $flight) {
    // ...
}
```

如果要根據一個欄位來過濾 `lazy` 方法拿到的資料，同時，這個欄位的資料在遍歷的時候還需要更新的話，那麼可以使用 `lazyById` 方法。在 `lazyById` 方法的內部，默認會查詢 id 欄位大於前一個 `chunk` 中最後一個模型的 id 。

```
Flight::where('departed', true)
    ->lazyById(200, $column = 'id')
    ->each->update(['departed' => false]);
```

你可以使用 `lazyByIdDesc` 方法根據 `id` 的降序過濾結果。

47.4.5 游標

與 `lazy` 方法類似，`cursor` 方法可用於在查詢數萬條 Eloquent 模型記錄時減少記憶體的使用。

`cursor` 方法只會執行一次資料庫查詢；但是，各個 Eloquent 模型在實際迭代之前不會被資料填充。因此，在遍歷游標時，在任何給定時間，只有一個 Eloquent 模型保留在記憶體中。

> **注意**
> 由於 `cursor` 方法一次只能在記憶體中保存一個 Eloquent 模型，因此它不能預載入關係。如果需要預載入關係，請考慮使用 `lazy` 方法。

在內部，`cursor` 方法使用 PHP generators 來實現此功能：

```
use App\Models\Flight;

foreach (Flight::where('destination', 'Zurich')->cursor() as $flight) {
    // ...
}
```

`cursor` 返回一個 `Illuminate\Support\LazyCollection` 實例。惰性集合 可以使用 Laravel 集合中的可用方法，同時一次僅將單個模型載入到記憶體中：

```
use App\Models\User;

$users = User::cursor()->filter(function (User $user) {
    return $user->id > 500;
});

foreach ($users as $user) {
    echo $user->id;
}
```

儘管 `cursor` 方法使用的記憶體比常規查詢要少得多（一次只在記憶體中保存一個 Eloquent 模型），但它最終仍會耗盡記憶體。這是由於 PHP 的 PDO 驅動程式內部將所有原始查詢結果快取在其緩衝區中。 如果要處理大量 Eloquent 記錄，請考慮使用 `lazy` 方法。

47.4.6 高級子查詢

47.4.6.1 selects 子查詢

Eloquent 還提供高級子查詢支援，你可以在單條語句中從相關表中提取資訊。 例如，假設我們有一個航班目的地表 `destinations` 和一個到達這些目的地的航班表 `flights`。 `flights` 表包含一個 `arrived_at` 欄位，指示航班何時到達目的地。

使用查詢生成器可用的子查詢功能 `select` 和 `addSelect` 方法，我們可以用單條語句查詢全部目的地 `destinations` 和 抵達各目的地最後一班航班的名稱：

```
use App\Models\Destination;
use App\Models\Flight;

return Destination::addSelect(['last_flight' => Flight::select('name')
    ->whereColumn('destination_id', 'destinations.id')
    ->orderByDesc('arrived_at')
    ->limit(1)
])->get();
```

47.4.6.2 子查詢排序

此外，查詢建構器的 `orderBy` 也同樣支援子查詢。繼續使用我們的航班為例，根據最後一次航班到達該目的

地的時間對所有目的地進行排序。這同樣可以在執行單個資料庫查詢時完成：

```
return Destination::orderByDesc(
    Flight::select('arrived_at')
        ->whereColumn('destination_id', 'destinations.id')
        ->orderByDesc('arrived_at')
        ->limit(1)
)->get();
```

47.5 檢索單個模型 / 聚合

除了檢索與給定查詢匹配的所有記錄之外，還可以使用 `find`、`first` 或 `firstWhere` 方法檢索單個記錄。這些方法不是返回模型集合，而是返回單個模型實例：

```
use App\Models\Flight;

// 通過主鍵檢索模型...
$flight = Flight::find(1);

// 檢索與查詢約束匹配的第一個模型...
$flight = Flight::where('active', 1)->first();

// 替代檢索與查詢約束匹配的第一個模型...
$flight = Flight::firstWhere('active', 1);
```

有時你可能希望檢索查詢的第一個結果或在未找到結果時執行一些其他操作。`firstOr` 方法將返回匹配查詢的第一個結果，或者，如果沒有找到結果，則執行給定的閉包。閉包返回的值將被視為 `firstOr` 方法的結果：

```
$flight = Flight::findOr(1, function () {
    // ...
});

$flight = Flight::where('legs', '>', 3)->firstOr(function () {
    // ...
});
```

47.5.1.1 未找到時拋出異常

如果找不到模型，你可能希望拋出異常。這在路由或 controller 中特別有用。 `findOrFail` 和 `firstOrFail` 方法將檢索查詢的第一個結果；但是，如果沒有找到結果，則會拋出 `Illuminate\Database\Eloquent\ModelNotFoundException`：

```
$flight = Flight::findOrFail(1);

$flight = Flight::where('legs', '>', 3)->firstOrFail();
```

如果沒有捕獲到 `ModelNotFoundException`，則會自動將 404 HTTP 響應傳送回客戶端：

```
use App\Models\Flight;

Route::get('/api/flights/{id}', function (string $id) {
    return Flight::findOrFail($id);
});
```

47.5.2 檢索或建立模型

`firstOrCreate` 方法將嘗試使用給定的列 / 值對來尋找資料庫記錄。如果在資料庫中找不到該模型，則將插入一條記錄，其中包含將第一個陣列參數與可選的第二個陣列參數合併後產生的屬性：

`firstOrNew` 方法，類似 `firstOrCreate`，會嘗試在資料庫中找到與給定屬性匹配的記錄。如果沒有找到，則會返回一個新的模型實例。請注意，由 `firstOrNew` 返回的模型尚未持久化到資料庫中。需要手動呼

叫 save 方法來保存它：

```
use App\Models\Flight;

// 按名稱檢索航班，如果不存在則建立它...
$flight = Flight::firstOrCreate([
    'name' => 'London to Paris'
]);

// 按名稱檢索航班或使用名稱、延遲和到達時間屬性建立它...
$flight = Flight::firstOrCreate(
    ['name' => 'London to Paris'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);

// 按名稱檢索航班或實例化一個新的航班實例...
$flight = Flight::firstOrNew([
    'name' => 'London to Paris'
]);

// 按名稱檢索航班或使用名稱、延遲和到達時間屬性實例化...
$flight = Flight::firstOrNew(
    ['name' => 'Tokyo to Sydney'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);
```

47.5.3 檢索聚合

當使用 Eloquent 模型互動的時候，你可以使用 count、sum、max，以及一些 laravel 查詢生成器提供的其他聚合方法。如你所需要的，這些方法會返回一個數字值而不是 Eloquent 模型實例：

```
$count = Flight::where('active', 1)->count();

$max = Flight::where('active', 1)->max('price');
```

47.6 新增 & 更新模型

47.6.1 新增

顯然，使用 Eloquent 的時候，我們不僅需要從資料庫中檢索模型，同時也需要新增新的資料記錄。值得高興的是，對於這種需求 Eloquent 可以從容應對。為了向資料庫新增新的資料記錄，你需要實例化一個新的模型實例並且為它的屬性賦值，然後呼叫這個實例的 save 方法：

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Flight;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class FlightController extends Controller
{
    /**
     * 向資料庫中儲存條新的航班資訊.
     */
    public function store(Request $request): RedirectResponse
    {
        // 驗證 request...
```

```
        $flight = new Flight;

        $flight->name = $request->name;

        $flight->save();

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

在這個例子中，我們使用來自 HTTP request 請求中的 `name` 參數值來對 `App\Models\Flight` 模型實例的 `name` 屬性賦值，當我們呼叫 `save` 方法時，資料庫便會增加一條資料記錄，模型的 `created_at` 和 `updated_at` 欄位將會在呼叫 `save` 方法時自動設定為相應的時間, 所以不需要手動去設定這兩個屬性。

或者，可以使用 `create` 方法使用單個 PHP 語句「保存」一個新模型。插入的模型實例將通過 `create` 方法返回：

```
use App\Models\Flight;

$flight = Flight::create([
    'name' => 'London to Paris',
]);
```

但是，在使用 `create` 方法之前，你需要在模型類上指定 `fillable` 或 `guarded` 屬性。這些屬性是必需的，因為默認情況下，所有 Eloquent 模型都受到保護，免受批次賦值漏洞的影響。 要瞭解有關批次賦值的更多資訊，請參閱批次賦值文件。

47.6.2 更新

`save` 方法也可以用來更新資料庫中已經存在的模型。要更新模型，應該檢索它並設定你想更新的任何屬性。然後呼叫模型的 `save` 方法。 同樣，`updated_at` 時間戳將自動更新，因此無需手動設定其值：

```
use App\Models\Flight;

$flight = Flight::find(1);

$flight->name = 'Paris to London';

$flight->save();
```

47.6.2.1 批次更新

還可以批次更新與給定條件匹配的所有模型。在此示例中，所有 `active` 且 `destination` 為 `San Diego` 的航班都將被標記為延遲：

```
Flight::where('active', 1)
      ->where('destination', 'San Diego')
      ->update(['delayed' => 1]);
```

`update` 方法需要一個表示應該更新的列的列和值對陣列。`update` 方法返回受影響的行數。

> **注意**
> 通過 Eloquent 批次更新時，不會觸發模型的 `saving`、`saved`、`updating` 和 `updated` 模型事件。 這是因為在批次更新時從未真正檢索到模型。

47.6.2.2 檢查屬性變更

Eloquent 提供了 `isDirty`、`isClean` 和 `wasChanged` 方法來檢查模型的內部狀態，並確定它的屬性與最初檢索模型時的變化情況。

isDirty 方法確定模型的任何屬性在檢索模型後是否已更改。你可以傳遞特定的屬性名稱來確定它是否「變髒」。isClean 方法將確定自檢索模型以來屬性是否保持不變。 它也接受可選的屬性參數：

```
use App\Models\User;

$user = User::create([
    'first_name' => 'Taylor',
    'last_name' => 'Otwell',
    'title' => 'Developer',
]);

$user->title = 'Painter';

$user->isDirty(); // true
$user->isDirty('title'); // true
$user->isDirty('first_name'); // false
$user->isDirty(['first_name', 'title']); // true

$user->isClean(); // false
$user->isClean('title'); // false
$user->isClean('first_name'); // true
$user->isClean(['first_name', 'title']); // false

$user->save();

$user->isDirty(); // false
$user->isClean(); // true
```

wasChanged 方法確定在當前請求週期內最後一次保存模型時是否更改了任何屬性。你還可以傳遞屬性名稱以查看特定屬性是否已更改：

```
$user = User::create([
    'first_name' => 'Taylor',
    'last_name' => 'Otwell',
    'title' => 'Developer',
]);

$user->title = 'Painter';

$user->save();

$user->wasChanged(); // true
$user->wasChanged('title'); // true
$user->wasChanged(['title', 'slug']); // true
$user->wasChanged('first_name'); // false
$user->wasChanged(['first_name', 'title']); // true
```

getOriginal 方法返回一個包含模型原始屬性的陣列，忽略載入模型之後進行的任何更改。你也可以傳遞特定的屬性名稱來獲取特定屬性的原始值：

```
$user = User::find(1);

$user->name; // John
$user->email; // john@example.com

$user->name = "Jack";
$user->name; // Jack

$user->getOriginal('name'); // John
$user->getOriginal(); // 原始屬性陣列
```

47.6.3 批次賦值

你可以使用 create 方法使用單個 PHP 語句「保存」一個新模型。插入的模型實例將通過該方法返回：

```
use App\Models\Flight;

$flight = Flight::create([
```

```
    'name' => 'London to Paris',
]);
```

但是，在使用 `create` 方法之前，需要在模型類上指定 `fillable` 或 `guarded` 屬性。 這些屬性是必需的，因為默認情況下，所有 Eloquent 模型都受到保護，免受批次分配漏洞的影響。

當使用者傳遞一個意外的 HTTP 請求欄位並且該欄位更改了你的資料庫中的一個欄位，而你沒有預料到時，就會出現批次分配漏洞。 例如，惡意使用者可能通過 HTTP 請求傳送 `is_admin` 參數，然後將其傳遞給模型的 `create` 方法，從而允許使用者將自己升級為管理員。

因此，你應該定義要使哪些模型屬性可批次分配。可以使用模型上的 `$fillable` 屬性來執行此操作。 例如，讓 `Flight` 模型的 `name` 屬性可以批次賦值：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * 可批次賦值的屬性。
     *
     * @var array
     */
    protected $fillable = ['name'];
}
```

一旦你指定了哪些屬性是可批次分配的，可以使用 `create` 方法在資料庫中插入一條新記錄。`create` 方法返回新建立的模型實例

```
$flight = Flight::create(['name' => 'London to Paris']);
```

如果你已經有一個模型實例，你可以使用 `fill` 方法來填充它的屬性陣列：

```
$flight->fill(['name' => 'Amsterdam to Frankfurt']);
```

47.6.3.1 批次賦值 & JSON 列

分配 JSON 列時，必須在模型的 `$fillable` 陣列中指定每個列的批次分配鍵。為了安全起見，Laravel 不支援在使用 `guarded` 屬性時更新巢狀的 JSON 屬性：

```
/**
 * 可以批次賦值的屬性。
 *
 * @var array
 */
protected $fillable = [
    'options->enabled',
];
```

47.6.3.2 允許批次分配

如果你想讓所有屬性都可以批次賦值，你可以將 `$guarded` 定義成一個空陣列。如果你選擇解除你的模型的保護，你應該時刻特別注意傳遞給 Eloquent 的 `fill`、`create` 和 `update` 方法的陣列：

```
/**
 * 不可以批次賦值的屬性。
 *
 * @var array
 */
protected $guarded = [];
```

47.6.3.3 批次作業異常拋出

默認情況下，在執行批次分配操作時，未包含在`$fillable`陣列中的屬性將被靜默丟棄。 在生產環境中，這是預期行為; 然而，在局部開發過程中，它可能導致為什麼模型更改沒有生效的困惑。

如果你願意，你可以指示 Laravel 在試圖通過呼叫 `preventSilentlyDiscardingAttributes` 方法填充一個不可填充的屬性時拋出一個異常。 通常，這個方法在應用程式服務提供者的 `boot` 方法中呼叫:

```
use Illuminate\Database\Eloquent\Model;

/**
 * 載入任意應用服務。
 */
public function boot(): void
{
    Model::preventSilentlyDiscardingAttributes($this->app->isLocal());
}
```

47.6.4 新增或更新

有時，如果不存在匹配的模型，你可能需要更新現有模型或建立新模型。與 `firstOrCreate` 方法一樣，`updateOrCreate` 方法會持久化模型，因此無需手動呼叫 `save` 方法。

在下面的示例中，如果存在 `departure` 位置為 `Oakland` 且 `destination` 位置為 `San Diego` 的航班，則其 `price` 和 `discounted` 列將被更新。 如果不存在這樣的航班，將建立一個新航班，該航班具有將第一個參數陣列與第二個參數陣列合併後的屬性：

```
$flight = Flight::updateOrCreate(
    ['departure' => 'Oakland', 'destination' => 'San Diego'],
    ['price' => 99, 'discounted' => 1]
);
```

如果你想在單個查詢中執行多個「新增或更新」，那麼應該使用 `upsert` 方法。該方法的第一個參數包含要插入或更新的值，而第二個參數列出了在關聯表中唯一標識記錄的列。該方法的第三個也是最後一個參數是一個列陣列，如果資料庫中已經存在匹配的記錄，則應該更新這些列。如果在模型上啟用了時間戳，`upsert` 方法將自動設定 `created_at` 和 `updated_at` 時間戳：

```
Flight::upsert([
    ['departure' => 'Oakland', 'destination' => 'San Diego', 'price' => 99],
    ['departure' => 'Chicago', 'destination' => 'New York', 'price' => 150]
], ['departure', 'destination'], ['price']);
```

> **注意**
> 除 SQL Server 外，其他所有資料庫都要求 `upsert` 方法的第二個參數中的列具有主鍵索引或唯一索引。 此外，MySQL 資料庫驅動程式忽略了 `upsert` 方法的第二個參數，總是使用表的主鍵索引和唯一索引來檢測現有的記錄

47.7 刪除模型

想刪除模型，你可以呼叫模型實例的 `delete` 方法:

```
use App\Models\Flight;

$flight = Flight::find(1);

$flight->delete();
```

你可以呼叫 `truncate` 方法來刪除所有模型關聯的資料庫記錄。 `truncate` 操作還將重設模型關聯表上的所有自動遞增 ID：

```
Flight::truncate();
```

47.7.1.1 通過其主鍵刪除現有模型

在上面的示例中，我們在呼叫 delete 方法之前從資料庫中檢索模型。但是，如果你知道模型的主鍵，則可以通過呼叫 destroy 方法刪除模型而無需顯式檢索它。除了接受單個主鍵之外，destroy 方法還將接受多個主鍵、主鍵陣列或主鍵 集合：

```
Flight::destroy(1);

Flight::destroy(1, 2, 3);

Flight::destroy([1, 2, 3]);

Flight::destroy(collect([1, 2, 3]));
```

> **注意**
> destroy 方法單獨載入每個模型並呼叫 delete 方法，以便為每個模型正確調度 deleting 和 deleted 事件。

47.7.1.2 使用查詢刪除模型

當然，你可以建構一個 Eloquent 查詢來刪除所有符合你查詢條件的模型。在此示例中，我們將刪除所有標記為非活動的航班。與批次更新一樣，批次刪除不會為已刪除的模型調度模型事件：

```
$deleted = Flight::where('active', 0)->delete();
```

> **注意**
> 通過 Eloquent 執行批次刪除語句時，不會為已刪除的模型調度 deleting 和 deleted 模型事件。這是因為在執行 delete 語句時從未真正檢索到模型。

47.7.2 軟刪除

除了實際從資料庫中刪除記錄之外，Eloquent 還可以「軟刪除」。軟刪除不會真的從資料庫中刪除記錄。相反，它在模型上設定了一個 deleted_at 屬性，記錄模型被「刪除」的日期和時間。要為模型啟用軟刪除，請將 Illuminate\Database\Eloquent\SoftDeletes trait 新增到模型中：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Flight extends Model
{
    use SoftDeletes;
}
```

> **注意**
> SoftDeletes trait 會自動將 deleted_at 屬性轉換為 DateTime / Carbon 實例

當然，你需要把 deleted_at 欄位新增到資料表中。Laravel 的資料遷移有建立這個欄位的方法：

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('flights', function (Blueprint $table) {
    $table->softDeletes();
});
```

```
Schema::table('flights', function (Blueprint $table) {
    $table->dropSoftDeletes();
});
```

那現在，當你在模型實例上使用 `delete` 方法，當前日期時間會寫入 `deleted_at` 欄位。同時，查詢出來的結果也會自動排除已被軟刪除的記錄。

判斷模型實例是否已被軟刪除，可以使用 `trashed` 方法：

```
if ($flight->trashed()) {
    // ...
}
```

47.7.2.1 恢復軟刪除的模型

有時你可能希望「撤銷」軟刪除的模型。要恢復軟刪除的模型，可以在模型實例上呼叫 `restore` 方法。`restore` 方法會將模型的 `deleted_at` 列設定為 `null`：

```
$flight->restore();
```

你也可以在查詢中使用 `restore` 方法，從而快速恢復多個模型。和其他「批次」操作一樣，這個操作不會觸發模型的任何事件：

```
Flight::withTrashed()
        ->where('airline_id', 1)
        ->restore();
```

`restore` 方法可以在關聯查詢中使用：

```
$flight->history()->restore();
```

47.7.2.2 永久刪除模型

有時你可能需要從資料庫中真正刪除模型。要從資料庫中永久刪除軟刪除的模型，請使用 `forceDelete` 方法：

```
$flight->forceDelete();
```

`forceDelete` 同樣可以用在關聯查詢上:

```
$flight->history()->forceDelete();
```

47.7.3 查詢軟刪除模型

47.7.3.1 包括已軟刪除的模型

如上所述，軟刪除模型將自動從查詢結果中排除。但是，你也可以通過在查詢上呼叫 `withTrashed` 方法來強制將軟刪除模型包含在查詢結果中：

```
use App\Models\Flight;

$flights = Flight::withTrashed()
                 ->where('account_id', 1)
                 ->get();
```

`withTrashed` 方法可以在 關聯查詢 中使用

```
$flight->history()->withTrashed()->get();
```

47.7.3.2 僅檢索軟刪除的模型

`onlyTrashed` 方法將檢索 **只被** 軟刪除模型：

```
$flights = Flight::onlyTrashed()
                 ->where('airline_id', 1)
                 ->get();
```

47.8 修剪模型

有時你可能希望定期刪除不再需要的模型。為此，你可以將 `Illuminate\Database\Eloquent\Prunable` 或 `Illuminate\Database\Eloquent\MassPrunable` trait 新增到要定期修剪的模型中。將其中一個 trait 新增到模型後，實現 `prunable` 方法，該方法返回一個 Eloquent 查詢建構器，用於檢索不再需要的模型資料：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Prunable;

class Flight extends Model
{
    use Prunable;

    /**
     * 獲取可修剪模型查詢構造器。
     */
    public function prunable(): Builder
    {
        return static::where('created_at', '<=', now()->subMonth());
    }
}
```

當將模型標記為 `Prunable` 時，你還可以在模型上定義 `pruning` 方法。該方法將在模型被刪除之前被呼叫。在從資料庫中永久刪除模型之前，此方法可用於刪除與模型關聯的任何其他資源，例如儲存的檔案：

```
/**
 * 準備模型進行修剪。
 */
protected function pruning(): void
{
    // ...
}
```

組態可修剪模型後，你還應該在應用程式的 `App\Console\Kernel` 類中調度 `model:prune` Artisan 命令。你可以自由選擇運行此命令的時間間隔：

```
/**
 * 定義應用程式的命令計畫。
 */
protected function schedule(Schedule $schedule): void
{
    $schedule->command('model:prune')->daily();
}
```

在後台，`model:prune` 命令會自動檢測應用程式的 `app/Models` 目錄中的「Prunable」模型。 如果模型位於不同的位置，可以使用 `--model` 選項來指定模型類名稱：

```
$schedule->command('model:prune', [
    '--model' => [Address::class, Flight::class],
])->daily();
```

如果你想在修剪所有其他檢測到的模型時排除某些模型被修剪，你可以使用 `--except` 選項：

```
$schedule->command('model:prune', [
    '--except' => [Address::class, Flight::class],
])->daily();
```

你可以通過執行帶有 `--pretend` 選項的 `model:prune` 命令來預測你的 `prunable` 查詢。預測時，`model:prune` 命令將報告該命令實際運行將修剪多少記錄：

```
php artisan model:prune --pretend
```

> **注意**
> 如果軟刪除模型與可修剪查詢匹配，則它們將被永久刪除（`forceDelete`）。

47.8.1.1 批次修剪模型

當模型被標記為 `Illuminate\Database\Eloquent\MassPrunable` 特徵時，模型會使用批次刪除查詢從資料庫中刪除。因此，不會呼叫 `pruning` 方法，也不會觸發 `deleting` 和 `deleted` 模型事件。這是因為模型在刪除之前從未真正檢索過，因此更高效：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\MassPrunable;

class Flight extends Model
{
    use MassPrunable;

    /**
     * 獲取可修剪模型查詢。
     */
    public function prunable(): Builder
    {
        return static::where('created_at', '<=', now()->subMonth());
    }
}
```

47.9 複製模型

可以使用 `replicate` 方法建立現有模型實例的未保存副本。在擁有共享許多相同屬性的模型實例時，此方法特別有用：

```
use App\Models\Address;

$shipping = Address::create([
    'type' => 'shipping',
    'line_1' => '123 Example Street',
    'city' => 'Victorville',
    'state' => 'CA',
    'postcode' => '90001',
]);

$billing = $shipping->replicate()->fill([
    'type' => 'billing'
]);

$billing->save();
```

要排除一個或多個屬性被覆制到新模型，可以將陣列傳遞給 `replicate` 方法：

```
$flight = Flight::create([
    'destination' => 'LAX',
    'origin' => 'LHR',
    'last_flown' => '2020-03-04 11:00:00',
    'last_pilot_id' => 747,
```

```
]);

$flight = $flight->replicate([
    'last_flown',
    'last_pilot_id'
]);
```

47.10 查詢範疇

47.10.1 全域範疇

全域範疇可以為模型的所有查詢新增約束。 Laravel 的軟刪除 功能就是利用全域範圍僅從資料庫中檢索「未刪除」模型。編寫全域範圍查詢可以為模型的每個查詢都新增約束條件。

47.10.1.1 編寫全域範疇

編寫全域範圍很簡單。首先，定義一個實現 `Illuminate\Database\Eloquent\Scope` 介面的類。 Laravel 沒有放置範疇類的常規位置，因此你可以自由地將此類放置在你希望的任何目錄中。

`Scope` 介面要求實現 `apply` 方法。 `apply` 方法可以根據需要向查詢中新增 `where` 約束或其他類型的子句：

```
<?php

namespace App\Models\Scopes;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Scope;

class AncientScope implements Scope
{
    /**
     * 將範疇應用於給定的 Eloquent 查詢建構器
     */
    public function apply(Builder $builder, Model $model): void
    {
        $builder->where('created_at', '<', now()->subYears(2000));
    }
}
```

> **注意**
> 如果需要在 `select` 語句裡新增欄位，應使用 `addSelect` 方法，而不是 `select` 方法。這將有效防止無意中替換現有 `select` 語句的情況。

47.10.1.2 應用全域範疇

要將全域範疇分配給模型，需要重寫模型的 `booted` 方法並使用 `addGlobalScope` 方法，`addGlobalScope` 方法接受範疇的一個實例作為它的唯一參數：

```
<?php

namespace App\Models;

use App\Models\Scopes\AncientScope;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
```

```
     * 模型的「引導」方法。
     */
    protected static function booted(): void
    {
        static::addGlobalScope(new AncientScope);
    }
}
```

將上例中的範疇新增到 App\Models\User 模型後，用 User::all() 方法將執行以下 SQL 查詢：

```
select * from `users` where `created_at` < 0021-02-18 00:00:00
```

47.10.1.3 匿名全域範疇

Eloquent 同樣允許使用閉包定義全域範疇，這樣就不需要為一個簡單的範疇而編寫一個單獨的類。使用閉包定義全域範疇時，你應該指定一個範疇名稱作為 addGlobalScope 方法的第一個參數：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 模型的「引導」方法。
     */
    protected static function booted(): void
    {
        static::addGlobalScope('ancient', function (Builder $builder) {
            $builder->where('created_at', '<', now()->subYears(2000));
        });
    }
}
```

47.10.1.4 取消全域範疇

如果需要對當前查詢取消全域範疇，需要使用 withoutGlobalScope 方法。該方法僅接受全域範疇類名作為它唯一的參數：

```
User::withoutGlobalScope(AncientScope::class)->get();
```

或者，如果你使用閉包定義了全域範疇，則應傳遞分配給全域範疇的字串名稱：

```
User::withoutGlobalScope('ancient')->get();
```

如果需要取消部分或者全部的全域範疇的話，需要使用 withoutGlobalScopes 方法：

```
// 取消全部全域範疇...
User::withoutGlobalScopes()->get();

// 取消部分範疇...
User::withoutGlobalScopes([
    FirstScope::class, SecondScope::class
])->get();
```

47.10.2 局部範疇

局部範疇允許定義通用的約束集合以便在應用程式中重複使用。例如，你可能經常需要獲取所有「流行」的使用者。要定義這樣一個範圍，只需要在對應的 Eloquent 模型方法前新增 scope 前綴。

範疇總是返回一個查詢構造器實例或者 void：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 只查詢受歡迎的使用者的範疇。
     */
    public function scopePopular(Builder $query): void
    {
        $query->where('votes', '>', 100);
    }

    /**
     * 只查詢 active 使用者的範疇。
     */
    public function scopeActive(Builder $query): void
    {
        $query->where('active', 1);
    }
}
```

47.10.2.1 使用局部範疇

一旦定義了範疇，就可以在查詢該模型時呼叫範疇方法。不過，在呼叫這些方法時不必包含 scope 前綴。甚至可以鏈式呼叫多個範疇，例如：

```
use App\Models\User;

$users = User::popular()->active()->orderBy('created_at')->get();
```

通過 or 查詢運算子組合多個 Eloquent 模型範疇可能需要使用閉包來實現正確的邏輯分組：

```
$users = User::popular()->orWhere(function (Builder $query) {
    $query->active();
})->get();
```

然而這可能有點麻煩，所以 Laravel 提供了一個更高階的 orWhere 方法，允許你流暢地將範疇連結在一起，而無需使用閉包：

```
$users = App\Models\User::popular()->orWhere->active()->get();
```

47.10.2.2 動態範疇

有時可能地希望定義一個可以接受參數的範疇。把額外參數傳遞給範疇就可以達到此目的。範疇參數要放在 $query 參數之後：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 將查詢範疇限制為僅包含給定類型的使用者。
     */
    public function scopeOfType(Builder $query, string $type): void
    {
        $query->where('type', $type);
    }
```

```
}
```

一旦將預期的參數新增到範疇方法的簽名中，你就可以在呼叫範疇時傳遞參數：

```
$users = User::ofType('admin')->get();
```

47.11 模型比較

有時可能需要判斷兩個模型是否「相同」。`is` 和 `isNot` 方法可以用來快速校驗兩個模型是否擁有相同的主鍵、表和資料庫連接：

```
if ($post->is($anotherPost)) {
    // ...
}

if ($post->isNot($anotherPost)) {
    // ...
}
```

當使用 `belongsTo`、`hasOne`、`morphTo` 和 `morphOne` relationships 時，`is` 和 `isNot` 方法也可用。當你想比較相關模型而不發出查詢來檢索該模型時，此方法特別有用：

```
if ($post->author()->is($user)) {
    // ...
}
```

47.12 Events

> **注意**
> 想要將 Eloquent 事件直接廣播到客戶端應用程式？查看 Laravel 的模型事件廣播。

Eloquent 模型觸發幾個事件，允許你掛接到模型生命週期的如下節點：`retrieved`、`creating`、`created`、`updating`、`updated`、`saving`、`saved`、`deleting`、`deleted`、`restoring`、`restored`、`replicating`。事件允許你每當特定模型保存或更新資料庫時執行程式碼。每個事件通過其構造器接受模型實例。

當從資料庫中檢索到現有模型時，將調度 `retrieved` 事件。 當一個新模型第一次被保存時，`creating` 和 `created` 事件將被觸發。 `updating` / `updated` 事件將在修改現有模型並呼叫 `save` 方法時觸發。`saving` / `saved` 事件將在建立或更新模型時觸發 - 即使模型的屬性沒有更改。以 `-ing` 結尾的事件名稱在模型的任何更改被持久化之前被調度，而以 `-ed` 結尾的事件在對模型的更改被持久化之後被調度。

要開始監聽模型事件，請在 Eloquent 模型上定義一個 $dispatchesEvents 屬性。此屬性將 Eloquent 模型生命週期的各個點對應到你定義的事件類。每個模型事件類都應該通過其建構函式接收受影響的模型的實例：

```
<?php

namespace App\Models;

use App\Events\UserDeleted;
use App\Events\UserSaved;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * 模型的事件對應。
     *
     * @var array
```

```
     */
    protected $dispatchesEvents = [
        'saved' => UserSaved::class,
        'deleted' => UserDeleted::class,
    ];
}
```

在定義和對應了 Eloquent 事件之後，可以使用 event listeners 來處理事件。

> **注意**
> 在使用 Eloquent 進行批次更新或刪除查詢時，受影響的模型不會觸發 `saved`、`updated`、`deleting` 和 `deleted` 等事件。這是因為在執行批次更新或刪除操作時，實際上沒有檢索到這些模型，所以也就不會觸發這些事件。

47.12.1 使用閉包

你可以註冊一些閉包涵數來處理模型事件，而不使用自訂事件類。通常，你應該在模型的 `booted` 方法中註冊這些閉包

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 模型的「booted」方法。
     */
    protected static function booted(): void
    {
        static::created(function (User $user) {
            // ...
        });
    }
}
```

如果需要，你可以在註冊模型事件時使用佇列匿名事件偵聽器 。這將指示 Laravel 使用應用程式的 queue 在後台執行模型事件監聽器：

```
use function Illuminate\Events\queueable;

static::created(queueable(function (User $user) {
    // ...
}));
```

47.12.2 觀察者

47.12.2.1 定義觀察者

如果在一個模型上監聽了多個事件，可以使用觀察者來將這些監聽器組織到一個單獨的類中。觀察者類的方法名對應到你希望監聽的 Eloquent 事件。這些方法都以模型作為其唯一參數。`make:observer` Artisan 命令可以快速建立新的觀察者類：

```
php artisan make:observer UserObserver --model=User
```

此命令將在 `App/Observers` 資料夾放置新的觀察者類。如果這個目錄不存在，Artisan 將替你建立。使用如下方式開啟觀察者：

```
<?php
```

```
namespace App\Observers;

use App\Models\User;

class UserObserver
{
    /**
     * 處理使用者「建立」事件。
     */
    public function created(User $user): void
    {
        // ...
    }

    /**
     * 處理使用者「更新」事件。
     */
    public function updated(User $user): void
    {
        // ...
    }

    /**
     * 處理使用者「刪除」事件。
     */
    public function deleted(User $user): void
    {
        // ...
    }

    /**
     * 處理使用者「還原」事件。
     */
    public function restored(User $user): void
    {
        // ...
    }

    /**
     * 處理使用者「強制刪除」事件。
     */
    public function forceDeleted(User $user): void
    {
        // ...
    }
}
```

要註冊觀察者，需要在要觀察的模型上呼叫 `Observer` 方法。你可以在應用程式的 `boot` 方法中註冊觀察者

`App\Providers\EventServiceProvider` 服務提供者:

```
use App\Models\User;
use App\Observers\UserObserver;

/**
 * 為你的應用程式註冊任何事件。
 */
public function boot(): void
{
    User::observe(UserObserver::class);
}
```

或者，可以在應用程式的 `$observers` 屬性中列出你的觀察者

`App\Providers\EventServiceProvider` class:

```
use App\Models\User;
use App\Observers\UserObserver;

/**
 * 應用程式的模型觀察者。
 *
 * @var array
 */
protected $observers = [
    User::class => [UserObserver::class],
];
```

技巧
觀察者可以監聽其他事件，例如「saving」和「retrieved」。這些事件在 events 文件中進行了描述。

47.12.2.2 觀察者與資料庫事務

在資料庫事務中建立模型時，你可能希望指示觀察者僅在提交資料庫事務後執行其事件處理程序。可以通過在觀察者上定義一個 `$afterCommit` 屬性來完成此操作。如果資料庫事務不在進行中，事件處理程序將立即執行：

```
<?php

namespace App\Observers;

use App\Models\User;

class UserObserver
{
    /**
     * 在提交所有事務後處理事件
     *
     * @var bool
     */
    public $afterCommit = true;

    /**
     *  處理使用者「建立」事件。
     */
    public function created(User $user): void
    {
        // ...
    }
}
```

47.12.3 靜默事件

也許有時候你會需要暫時將所有由模型觸發的事件「靜默」處理。使用 `withoutEvents` 達到目的。`withoutEvents` 方法接受一個閉包作為唯一參數。任何在閉包中執行的程式碼都不會被分配模型事件，並且閉包涵數返回的任何值都將被 `withoutEvents` 方法所返回：

```
use App\Models\User;

$user = User::withoutEvents(function () {
    User::findOrFail(1)->delete();

    return User::find(2);
});
```

47.12.3.1 靜默的保存單個模型

有時候，你也許會想要「保存」一個已有的模型，且不觸發任何事件。那麼你可用 `saveQuietly` 方法達到目的：

```
$user = User::findOrFail(1);

$user->name = 'Victoria Faith';

$user->saveQuietly();
```

你也可以「更新」「刪除」「軟刪除」「還原」「複製」給定模型且不觸發任何事件：

```
$user->deleteQuietly();
$user->forceDeleteQuietly();
$user->restoreQuietly();
```

48 關聯

48.1 簡介

資料庫表通常相互關聯。例如，一篇部落格文章可能有許多評論，或者一個訂單對應一個下單使用者。Eloquent 讓這些關聯的管理和使用變得簡單，並支援多種常用的關聯類型：

48.2 定義關聯

Eloquent 關聯在 Eloquent 模型類中以方法的形式呈現。如同 Eloquent 模型本身，關聯也可以作為強大的查詢語句構造器，使用，提供了強大的鏈式呼叫和查詢功能。例如，我們可以在 posts 關聯的鏈式呼叫中附加一個約束條件：

```
$user->posts()->where('active', 1)->get();
```

不過在深入使用關聯之前，讓我們先學習如何定義每種關聯類型。

48.2.1 一對一

一對一是最基本的資料庫關係。 例如，一個 User 模型可能與一個 Phone 模型相關聯。為了定義這個關聯關係，我們要在 User 模型中寫一個 phone 方法。 在 phone 方法中呼叫 hasOne 方法並返回其結果。 hasOne 方法被定義在 Illuminate\Database\Eloquent\Model 這個模型基類中：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasOne;

class User extends Model
{
    /**
     * 獲取與使用者相關的電話記錄
     */
    public function phone(): HasOne
    {
        return $this->hasOne(Phone::class);
    }
}
```

hasOne 方法的第一個參數是關聯模型的類名。一旦定義了模型關聯，我們就可以使用 Eloquent 的動態屬性獲得相關的記錄。動態屬性允許你訪問該關聯方法，就像訪問模型中定義的屬性一樣：

```
$phone = User::find(1)->phone;
```

Eloquent 基於父模型 User 的名稱來確定關聯模型 Phone 的外部索引鍵名稱。在本例中，會自動假定 Phone 模型有一個 user_id 的外部索引鍵。如果你想重寫這個約定，可以傳遞第二個參數給 hasOne 方法：

```
return $this->hasOne(Phone::class, 'foreign_key');
```

另外，Eloquent 假設外部索引鍵的值是與父模型的主鍵（Primary Key）相同的。換句話說，Eloquent 將會通過 Phone 記錄的 user_id 列中尋找與使用者表的 id 列相匹配的值。如果你希望使用自訂的主鍵值，而不是使用 id 或者模型中的 $primaryKey 屬性，你可以給 hasOne 方法傳遞第三個參數：

```
return $this->hasOne(Phone::class, 'foreign_key', 'local_key');
```

48.2.1.1 定義反向關聯

我們已經能從 User 模型訪問到 Phone 模型了。接下來，讓我們再在 Phone 模型上定義一個關聯，它能讓我們訪問到擁有該電話的使用者。我們可以使用 belongsTo 方法來定義反向關聯， belongsTo 方法與 hasOne 方法相對應：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Phone extends Model
{
    /**
     * 獲取擁有此電話的使用者
     */
    public function user(): BelongsTo
    {
        return $this->belongsTo(User::class);
    }
}
```

在呼叫 user 方法時，Eloquent 會嘗試尋找一個 User 模型，該 User 模型上的 id 欄位會與 Phone 模型上的 user_id 欄位相匹配。

Eloquent 通過關聯方法（user）的名稱並使用 _id 作為後綴名來確定外部索引鍵名稱。因此，在本例中，Eloquent 會假設 Phone 模型有一個 user_id 欄位。但是，如果 Phone 模型的外部索引鍵不是 user_id，這時你可以給 belongsTo 方法的第二個參數傳遞一個自訂鍵名：

```
/**
 * 獲取擁有此電話的使用者
 */
public function user(): BelongsTo
{
    return $this->belongsTo(User::class, 'foreign_key');
}
```

如果父模型的主鍵未使用 id 作為欄位名，或者你想要使用其他的欄位來匹配相關聯的模型，那麼你可以向 belongsTo 方法傳遞第三個參數，這個參數是在父模型中自己定義的欄位名稱：

```
/**
 * 獲取當前手機號的使用者
 */
public function user(): BelongsTo
{
    return $this->belongsTo(User::class, 'foreign_key', 'owner_key');
}
```

48.2.2 一對多

當要定義一個模型是其他 （一個或者多個）模型的父模型這種關係時，可以使用一對多關聯。例如，一篇部落格可以有很多條評論。和其他模型關聯一樣，一對多關聯也是在 Eloquent 模型檔案中用一個方法來定義的：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasMany;

class Post extends Model
{
```

```
    /**
     * 獲取這篇部落格的所有評論
     */
    public function comments(): HasMany
    {
        return $this->hasMany(Comment::class);
    }
}
```

注意，Eloquent 將會自動為 Comment 模型選擇一個合適的外部索引鍵。通常，這個外部索引鍵是通過使用父模型的「蛇形命名」方式，然後再加上 _id. 的方式來命名的。因此，在上面這個例子中，Eloquent 將會默認 Comment 模型的外部索引鍵是 post_id 欄位。

如果關聯方法被定義，那麼我們就可以通過 comments 屬性來訪問相關的評論 集合。注意，由於 Eloquent 提供了「動態屬性」，所以我們就可以像訪問模型屬性一樣來訪問關聯方法：

```
use App\Models\Post;

$comments = Post::find(1)->comments;

foreach ($comments as $comment) {
    // ...
}
```

由於所有的關係都可以看成是查詢構造器，所以你也可以通過鏈式呼叫的方式，在 comments 方法中繼續新增條件約束：

```
$comment = Post::find(1)->comments()
                        ->where('title', 'foo')
                        ->first();
```

像 hasOne 方法一樣，你也可以通過將附加參數傳遞給 hasMany 方法來覆蓋外部索引鍵和本地鍵：

```
return $this->hasMany(Comment::class, 'foreign_key');

return $this->hasMany(Comment::class, 'foreign_key', 'local_key');
```

48.2.3 一對多 (反向) / 屬於

目前我們可以訪問一篇文章的所有評論，下面我們可以定義一個關聯關係，從而讓我們可以通過一條評論來獲取到它所屬的文章。這個關聯關係是 hasMany 的反向，可以在子模型中通過 belongsTo 方法來定義這種關聯關係：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Comment extends Model
{
    /**
     * 獲取這條評論所屬的文章。
     */
    public function post(): BelongsTo
    {
        return $this->belongsTo(Post::class);
    }
}
```

如果定義了這種關聯關係，那麼我們就可以通過 Comment 模型中的 post 「動態屬性」來獲取到這條評論所屬的文章：

```
use App\Models\Comment;
```

```
$comment = Comment::find(1);

return $comment->post->title;
```

在上面這個例子中，Eloquent 將會嘗試尋找 `Post` 模型中的 `id` 欄位與 `Comment` 模型中的 `post_id` 欄位相匹配。

Eloquent 通過檢查關聯方法的名稱，從而在關聯方法名稱後面加上 `_`，然後再加上父模型（Post）的主鍵名稱，以此來作為默認的外部索引鍵名。因此，在上面這個例子中，Eloquent 將會默認 `Post` 模型在 `comments` 表中的外部索引鍵是 `post_id`。

但是，如果你的外部索引鍵不遵循這種約定的話，那麼你可以傳遞一個自訂的外部索引鍵名來作為 `belongsTo` 方法的第二個參數：

```
/**
 * 獲取這條評論所屬的文章。
 */
public function post(): BelongsTo
{
    return $this->belongsTo(Post::class, 'foreign_key');
}
```

如果你的父表不使用 `id` 作為主鍵，或者你希望使用不同的列來關聯模型，你可以將第三個參數傳遞給 `belongsTo` 方法，指定父表的自訂鍵：

```
/**
 * 獲取這條評論所屬的文章。
 */
public function post(): BelongsTo
{
    return $this->belongsTo(Post::class, 'foreign_key', 'owner_key');
}
```

48.2.3.1 默認模型

當 `belongsTo`，`hasOne`，`hasOneThrough` 和 `morphOne` 這些關聯方法返回 `null` 的時候，你可以定義一個默認的模型返回。該模式通常被稱為 空對象模式，它可以幫你省略程式碼中的一些條件判斷。在下面這個例子中，如果 `Post` 模型中沒有使用者，那麼 `user` 關聯關係將會返回一個空的 `App\Models\User` 模型：

```
/**
 * 獲取文章的作者。
 */
public function user(): BelongsTo
{
    return $this->belongsTo(User::class)->withDefault();
}
```

可以向 `withDefault` 方法傳遞陣列或者閉包來填充默認模型的屬性。

```
/**
 * 獲取文章的作者。
 */
public function user(): BelongsTo
{
    return $this->belongsTo(User::class)->withDefault([
        'name' => 'Guest Author',
    ]);
}

/**
 * 獲取文章的作者。
 */
public function user(): BelongsTo
{
    return $this->belongsTo(User::class)->withDefault(function (User $user, Post $post)
{
```

```
        $user->name = 'Guest Author';
    });
}
```

48.2.3.2 查詢所屬關係

在查詢「所屬」的子模型時，可以建構 `where` 語句來檢索相應的 Eloquent 模型：

```
use App\Models\Post;

$posts = Post::where('user_id', $user->id)->get();
```

但是，你會發現使用 `whereBelongsTo` 方法更方便，它會自動確定給定模型的正確關係和外部索引鍵：

```
$posts = Post::whereBelongsTo($user)->get();
```

你還可以向 `whereBelongsTo` 方法提供一個 集合 實例。 這樣 Laravel 將檢索屬於集合中任何父模型的子模型：

```
$users = User::where('vip', true)->get();

$posts = Post::whereBelongsTo($users)->get();
```

默認情況下，Laravel 將根據模型的類名來確定給定模型的關聯關係； 你也可以通過將關係名稱作為 `whereBelongsTo` 方法的第二個參數來手動指定關係名稱：

```
$posts = Post::whereBelongsTo($user, 'author')->get();
```

48.2.4 一對多檢索

有時一個模型可能有許多相關模型，如果你想很輕鬆的檢索「最新」或「最舊」的相關模型。例如，一個 `User` 模型可能與許多 `Order` 模型相關，但你想定義一種方便的方式來與使用者最近下的訂單進行互動。 可以使用 `hasOne` 關係類型結合 `ofMany` 方法來完成此操作：

```
/**
 * 獲取使用者最新的訂單。
 */
public function latestOrder(): HasOne
{
    return $this->hasOne(Order::class)->latestOfMany();
}
```

同樣，你可以定義一個方法來檢索 「oldest」或第一個相關模型：

```
/**
 * 獲取使用者最早的訂單。
 */
public function oldestOrder(): HasOne
{
    return $this->hasOne(Order::class)->oldestOfMany();
}
```

默認情況下，`latestOfMany` 和 `oldestOfMany` 方法將根據模型的主鍵檢索最新或最舊的相關模型，該主鍵必須是可排序的。 但是，有時你可能希望使用不同的排序條件從更大的關係中檢索單個模型。

例如，使用 `ofMany` 方法，可以檢索使用者最昂貴的訂單。 `ofMany` 方法接受可排序列作為其第一個參數，以及在查詢相關模型時應用哪個聚合函數（`min` 或 `max`）：

```
/**
 * 獲取使用者最昂貴的訂單。
 */
public function largestOrder(): HasOne
{
    return $this->hasOne(Order::class)->ofMany('price', 'max');
}
```

注意 因為 PostgreSQL 不支援對 UUID 列執行 MAX 函數，所以目前無法將一對多關係與 PostgreSQL UUID 列結合使用。

48.2.4.1 進階一對多檢索

可以建構更高級的「一對多檢索」關係。例如，一個 Product 模型可能有許多關聯的 Price 模型，即使在新定價發佈後，這些模型也會保留在系統中。此外，產品的新定價資料能夠通過 published_at 列提前發佈，以便在未來某日生效。

因此，我們需要檢索最新的發佈定價。 此外，如果兩個價格的發佈日期相同，我們優先選擇 ID 更大的價格。為此，我們必須將一個陣列傳遞給 ofMany 方法，其中包含確定最新價格的可排序列。此外，將提供一個閉包作為 ofMany 方法的第二個參數。此閉包將負責向關係查詢新增額外的發佈日期約束：

```
/**
 * 獲取產品的當前定價。
 */
public function currentPricing(): HasOne
{
    return $this->hasOne(Price::class)->ofMany([
        'published_at' => 'max',
        'id' => 'max',
    ], function (Builder $query) {
        $query->where('published_at', '<', now());
    });
}
```

48.2.5 遠端一對一

「遠端一對一」關聯定義了與另一個模型的一對一的關聯。然而，這種關聯是聲明的模型通過第三個模型來與另一個模型的一個實例相匹配。

例如，在一個汽車維修的應用程式中，每一個 Mechanic 模型都與一個 Car 模型相關聯，同時每一個 Car 模型也和一個 Owner 模型相關聯。雖然維修師（mechanic）和車主（owner）在資料庫中並沒有直接的關聯，但是維修師可以通過 Car 模型來找到車主。讓我們來看看定義這種關聯所需要的資料表：

```
mechanics
    id - integer
    name - string

cars
    id - integer
    model - string
    mechanic_id - integer

owners
    id - integer
    name - string
    car_id - integer
```

既然我們已經瞭解了遠端一對一的表結構，那麼我們就可以在 Mechanic 模型中定義這種關聯：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasOneThrough;

class Mechanic extends Model
{
    /**
     * 獲取汽車的主人。
```

```
     */
    public function carOwner(): HasOneThrough
    {
        return $this->hasOneThrough(Owner::class, Car::class);
    }
}
```

傳遞給 `hasOneThrough` 方法的第一個參數是我們希望訪問的最終模型的名稱，而第二個參數是中間模型的名稱。

或者，如果相關的關聯已經在關聯中涉及的所有模型上被定義，你可以通過呼叫 `through` 方法和提供這些關聯的名稱來流式定義一個「遠端一對一」關聯。例如，`Mechanic` 模型有一個 `cars` 關聯，`Car` 模型有一個 `owner` 關聯，你可以這樣定義一個連接維修師和車主的「遠端一對一」關聯：

```
// 基於字串的語法...
return $this->through('cars')->has('owner');

// 動態語法...
return $this->throughCars()->hasOwner();
```

48.2.5.1 鍵名約定

當使用遠端一對一進行關聯查詢時，Eloquent 將會使用約定的外部索引鍵名。如果你想要自訂相關聯的鍵名的話，可以傳遞兩個參數來作為「hasOneThrough」 方法的第三個和第四個參數。第三個參數是中間表的外部索引鍵名。第四個參數是最終想要訪問的模型的外部索引鍵名。第五個參數是當前模型的本地鍵名，第六個參數是中間模型的本地鍵名：

```
class Mechanic extends Model
{
    /**
     * Get the car's owner.
     */
    public function carOwner(): HasOneThrough
    {
        return $this->hasOneThrough(
            Owner::class,
            Car::class,
            'mechanic_id', // 機械師表的外部索引鍵...
            'car_id', // 車主表的外部索引鍵...
            'id', // 機械師表的本地鍵...
            'id' // 汽車表的本地鍵...
        );
    }
}
```

如果所涉及的模型已經定義了相關關係，可以呼叫 `through` 方法並提供關係名來定義「遠端一對一」關聯。該方法的優點是重複使用已有關係上定義的主鍵約定：

```
// 基本語法...
return $this->through('cars')->has('owner');

// 動態語法...
return $this->throughCars()->hasOwner();
```

48.2.6 遠端一對多

「遠端一對多」關聯是可以通過中間關係來實現遠端一對多的。例如，我們正在建構一個像 Laravel Vapor 這樣的部署平台。一個 `Project` 模型可以通過一個中間的 `Environment` 模型來訪問許多個 `Deployment` 模型。就像上面的這個例子，可以在給定的 environment 中很方便的獲取所有的 deployments。下面是定義這種關聯關係所需要的資料表：

```
projects
```

```
    id - integer
    name - string

environments
    id - integer
    project_id - integer
    name - string

deployments
    id - integer
    environment_id - integer
    commit_hash - string
```

既然我們已經檢查了關係的表結構，現在讓我們在 `Project` 模型上定義該關係：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasManyThrough;

class Project extends Model
{
    /**
     * 獲取該項目的所有部署。
     */
    public function deployments(): HasManyThrough
    {
        return $this->hasManyThrough(Deployment::class, Environment::class);
    }
}
```

`hasManyThrough` 方法中傳遞的第一個參數是我們希望訪問的最終模型名稱，而第二個參數是中間模型的名稱。

或者，所有模型上都定義好了關係，你可以通過呼叫 `through` 方法並提供這些關係的名稱來定義「has-many-through」關係。例如，如果 `Project` 模型具有 `environments` 關係，而 `Environment` 模型具有 `deployments` 關係，則可以定義連接 project 和 deployments 的「has-many-through」關係，如下所示：

```
// 基於字串的語法。。。
return $this->through('environments')->has('deployments');

// 動態語法。。。
return $this->throughEnvironments()->hasDeployments();
```

雖然 `Deployment` 模型的表格不包含 `project_id` 列，但 `hasManyThrough` 關係通過 `$project->deployments` 提供了訪問項目的部署方式。為了檢索這些模型，Eloquent 在中間的 `Environment` 模型表中檢查 `project_id` 列。在找到相關的 environment ID 後，它們被用來查詢 `Deployment` 模型。

48.2.6.1 鍵名約定

在執行關係查詢時，通常會使用典型的 Eloquent 外部索引鍵約定。如果你想要自訂關係鍵名，可以將它們作為 `hasManyThrough` 方法的第三個和第四個參數傳遞。第三個參數是中間模型上的外部索引鍵名稱。第四個參數是最終模型上的外部索引鍵名稱。第五個參數是本地鍵，而第六個參數是中間模型的本地鍵：

```
class Project extends Model
{
    public function deployments(): HasManyThrough
    {
        return $this->hasManyThrough(
            Deployment::class,
            Environment::class,
            'project_id', // 在 environments 表上的外部索引鍵...
```

```
            'environment_id', // 在 deployments 表上的外部索引鍵...
            'id', // 在 projects 表上的本地鍵...
            'id' // 在 environments 表格上的本地鍵...
        );
    }
}
```

或者，如前所述，如果涉及關係的相關關係已經在所有模型上定義，你可以通過呼叫 `through` 方法並提供這些關係的名稱來定義「has-many-through」關係。這種方法的優點是可以重複使用已經定義在現有關係上的鍵約定：

```
// 基於字串的語法。。。
return $this->through('environments')->has('deployments');

// 動態語法。。。
return $this->throughEnvironments()->hasDeployments();
```

48.3 多對多關聯

多對多關聯比 `hasOne` 和 `hasMany` 關聯略微複雜。舉個例子，一個使用者可以擁有多個角色，同時這些角色也可以分配給其他使用者。例如，一個使用者可是「作者」和「編輯」；當然，這些角色也可以分配給其他使用者。所以，一個使用者可以擁有多個角色，一個角色可以分配給多個使用者。

48.3.1.1 表結構

要定義這種關聯，需要三個資料庫表: `users`, `roles` 和 `role_user`。`role_user` 表的命名是由關聯的兩個模型按照字母順序來的，並且包含了 `user_id` 和 `role_id` 欄位。該表用作連結 `users` 和 `roles` 的中間表

特別提醒，由於角色可以屬於多個使用者，因此我們不能簡單地在 `roles` 表上放置 `user_id` 列。如果這樣，這意味著角色只能屬於一個使用者。為了支援將角色分配給多個使用者，需要使用 `role_user` 表。我們可以這樣定義表結構：

```
users
    id - integer
    name - string

roles
    id - integer
    name - string

role_user
    user_id - integer
    role_id - integer
```

48.3.1.2 模型結構

多對多關聯是通過呼叫 `belongsToMany` 方法結果的方法來定義的。 `belongsToMany` 方法由 `Illuminate\Database\Eloquent\Model` 基類提供，所有應用程式的 Eloquent 模型都使用該基類。 例如，讓我們在 `User` 模型上定義一個 `roles` 方法。 傳遞給此方法的第一個參數是相關模型類的名稱：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsToMany;

class User extends Model
{
    /**
```

```
     * 使用者所擁有的角色
     */
    public function roles(): BelongsToMany
    {
        return $this->belongsToMany(Role::class);
    }
}
```

定義關係後，可以使用 `roles` 動態關係屬性訪問使用者的角色：

```
use App\Models\User;

$user = User::find(1);

foreach ($user->roles as $role) {
    // ...
}
```

由於所有的關係也可以作為查詢建構器，你可以通過呼叫 `roles()` 方法查詢來為關係新增約束：

```
$roles = User::find(1)->roles()->orderBy('name')->get();
```

為了確定關係的中間表的表名，Eloquent 會按字母順序連接兩個相關的模型名。 你也可以隨意覆蓋此約定。 通過將第二個參數傳遞給 `belongsToMany` 方法來做到這一點：

```
return $this->belongsToMany(Role::class, 'role_user');
```

除了自訂連接表的表名，你還可以通過傳遞額外的參數到 `belongsToMany` 方法來定義該表中欄位的鍵名。 第三個參數是定義此關聯的模型在連接表裡的外部索引鍵名，第四個參數是另一個模型在連接表裡的外部索引鍵名:

```
return $this->belongsToMany(Role::class, 'role_user', 'user_id', 'role_id');
```

48.3.1.3 定義反向關聯

要定義多對多的反向關聯，只需要在關聯模型中呼叫 `belongsToMany` 方法。我們在 `Role` 模型中定義 `users` 方法:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsToMany;

class Role extends Model
{
    /**
     * 擁有此角色的使用者
     */
    public function users(): BelongsToMany
    {
        return $this->belongsToMany(User::class);
    }
}
```

如你所見，除了引用 `App\Models\User` 模型之外，該關係的定義與其對應的 `User` 模型完全相同。 由於我們復用了 `belongsToMany` 方法，所以在定義多對多關係的「反向」關係時，所有常用的表和鍵自訂選項都可用。

48.3.2 獲取中間表欄位

如上所述，處理多對多關係需要一個中間表。 Eloquent 提供了一些非常有用的方式來與它進行互動。 假設我們的 `User` 對象關聯了多個 `Role` 對象。在獲得這些關聯對象後，可以使用模型的 `pivot` 屬性訪問中間表的屬

性：

```
use App\Models\User;

$user = User::find(1);

foreach ($user->roles as $role) {
    echo $role->pivot->created_at;
}
```

需要注意的是，我們獲取的每個 Role 模型對象，都會被自動賦予 pivot 屬性，它代表中間表的一個模型對象，並且可以像其他的 Eloquent 模型一樣使用。

默認情況下，pivot 對象只包含兩個關聯模型的主鍵，如果你的中間表裡還有其他額外欄位，你必須在定義關聯時明確指出：

```
return $this->belongsToMany(Role::class)->withPivot('active', 'created_by');
```

如果你想讓中間表自動維護 created_at 和 updated_at 時間戳，那麼在定義關聯時附加上 withTimestamps 方法即可：

```
return $this->belongsToMany(Role::class)->withTimestamps();
```

> **注意** 使用 Eloquent 自動維護時間戳的中間表需要同時具有 created_at 和 updated_at 時間戳欄位。

48.3.2.1 自訂 pivot 屬性名稱

如前所述，可以通過 pivot 屬性在模型上訪問中間表中的屬性。 但是，你可以隨意自訂此屬性的名稱，以更好地反映其在應用程式中的用途。

例如，如果你的應用程式包含可能訂閱播客的使用者，則使用者和播客之間可能存在多對多關係。 如果是這種情況，你可能希望將中間表屬性重新命名為 subscription 而不是 pivot。 這可以在定義關係時使用 as 方法來完成：

```
return $this->belongsToMany(Podcast::class)
                ->as('subscription')
                ->withTimestamps();
```

一旦定義完成，你可以使用自訂名稱訪問中間表資料：

```
$users = User::with('podcasts')->get();

foreach ($users->flatMap->podcasts as $podcast) {
    echo $podcast->subscription->created_at;
}
```

48.3.3 通過中間表過濾查詢

你還可以在定義關係時使用 wherePivot、wherePivotIn、wherePivotNotIn、wherePivotBetween、wherePivotNotBetween、wherePivotNull 和 wherePivotNotNull 方法過濾 belongsToMany 關係查詢返回的結果：

```
return $this->belongsToMany(Role::class)
                ->wherePivot('approved', 1);

return $this->belongsToMany(Role::class)
                ->wherePivotIn('priority', [1, 2]);

return $this->belongsToMany(Role::class)
                ->wherePivotNotIn('priority', [1, 2]);

return $this->belongsToMany(Podcast::class)
```

```
                  ->as('subscriptions')
                  ->wherePivotBetween('created_at', ['2020-01-01 00:00:00', '2020-12-31
00:00:00']);

return $this->belongsToMany(Podcast::class)
                  ->as('subscriptions')
                  ->wherePivotNotBetween('created_at', ['2020-01-01 00:00:00', '2020-12-
31 00:00:00']);

return $this->belongsToMany(Podcast::class)
                  ->as('subscriptions')
                  ->wherePivotNull('expired_at');

return $this->belongsToMany(Podcast::class)
                  ->as('subscriptions')
                  ->wherePivotNotNull('expired_at');
```

48.3.4 通過中間表欄位排序

你可以使用 `orderByPivot` 方法對 `belongsToMany` 關係查詢返回的結果進行排序。在下面的例子中，我們將檢索使用者的最新徽章：

```
return $this->belongsToMany(Badge::class)
                  ->where('rank', 'gold')
                  ->orderByPivot('created_at', 'desc');
```

48.3.5 自訂中間表模型

如果你想定義一個自訂模型來表示多對多關係的中間表，你可以在定義關係時呼叫 `using` 方法。

自訂多對多中間表模型都必須繼承 `Illuminate\Database\Eloquent\Relations\Pivot` 類，自訂多對多（多型）中間表模型必須繼承 `Illuminate\Database\Eloquent\Relations\MorphPivot` 類。例如，我們在寫 `Role` 模型的關聯時，使用自訂中間表模型 `RoleUser`：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsToMany;

class Role extends Model
{
    /**
     * 屬於該角色的使用者。
     */
    public function users(): BelongsToMany
    {
        return $this->belongsToMany(User::class)->using(RoleUser::class);
    }
}
```

當定義 `RoleUser` 模型時，我們要繼承 `Illuminate\Database\Eloquent\Relations\Pivot` 類：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Relations\Pivot;

class RoleUser extends Pivot
{
    // ...
}
```

注意 Pivot 模型不可以使用 `SoftDeletes` trait。 如果需要軟刪除資料關聯記錄，請考慮將資料關聯模型轉換為實際的 Eloquent 模型。

48.3.5.1 自訂中間模型和自增 ID

如果你用一個自訂的中繼模型定義了多對多的關係，而且這個中繼模型擁有一個自增的主鍵，你應當確保這個自訂中繼模型類中定義了一個 `incrementing` 屬性且其值為 `true`。

```
/**
 * 標識 ID 是否自增
 *
 * @var bool
 */
public $incrementing = true;
```

48.4 多型關係

多型關聯允許子模型使用單個關聯屬於多種類型的模型。例如，假設你正在建構一個應用程式，允許使用者共享部落格文章和視訊。在這樣的應用程式中，`Comment` 模型可能同時屬於 `Post` 和 `Video` 模型。

48.4.1 一對一 (多型)

48.4.1.1 表結構

一對一多型關聯類似於典型的一對一關係，但是子模型可以使用單個關聯屬於多個類型的模型。例如，一個部落格 `Post` 和一個 `User` 可以共享到一個 `Image` 模型的多型關聯。使用一對一多型關聯允許你擁有一個唯一圖像的單個表，這些圖像可以與帖子和使用者關聯。首先，讓我們查看表結構：

```
posts
    id - integer
    name - string

users
    id - integer
    name - string

images
    id - integer
    url - string
    imageable_id - integer
    imageable_type - string
```

請注意 `images` 表上的 `imageable_id` 和 `imageable_type` 兩列。`imageable_id` 列將包含帖子或使用者的 ID 值，而 `imageable_type` 列將包含父模型的類名。`imageable_type` 列用於 Eloquent 在訪問 `imageable` 關聯時確定要返回哪種類型的父模型。在本例中，該列將包含 `App\Models\Post` 或 `App\Models\User`。

48.4.1.2 模型結構

接下來，讓我們來看一下建構這個關係所需的模型定義：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphTo;
```

```
class Image extends Model
{
     /**
      * 獲取父級 imageable 模型（使用者或帖子）。
      */
     public function imageable(): MorphTo
     {
          return $this->morphTo();
     }
}

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphOne;

class Post extends Model
{
     /**
      * 獲取文章圖片
      */
     public function image(): MorphOne
     {
          return $this->morphOne(Image::class, 'imageable');
     }
}

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphOne;

class User extends Model
{
     /**
      * 獲取使用者的圖片。
      */
     public function image(): MorphOne
     {
          return $this->morphOne(Image::class, 'imageable');
     }
}
```

48.4.1.3 檢索關聯關係

一旦定義了表和模型，就可以通過模型訪問此關聯。比如，要獲取文章圖片，可以使用 image 動態屬性：

```
use App\Models\Post;

$post = Post::find(1);

$image = $post->image;
```

還可以通過訪問執行 morphTo 呼叫的方法名來從多型模型中獲知父模型。在這個例子中，就是 Image 模型的 imageable 方法。所以，我們可以像動態屬性那樣訪問這個方法：

```
use App\Models\Image;

$image = Image::find(1);

$imageable = $image->imageable;
```

Image 模型上的 imageable 關係將返回 Post 實例或 User 實例，具體取決於模型擁有圖像的類型。

48.4.1.4 鍵名約定

如有需要，你可以指定多型子模型中使用的 id 和 type 列的名稱。 如果這樣做，請確保始終將關聯名稱作為第一個參數傳遞給 morphTo 方法。 通常，此值應與方法名稱匹配，因此你可以使用 PHP 的 __FUNCTION__

常數：

```
/**
 * 獲取 image 實例所屬的模型
 */
public function imageable(): MorphTo
{
    return $this->morphTo(__FUNCTION__, 'imageable_type', 'imageable_id');
}
```

48.4.2 一對多（多型）

48.4.2.1 表結構

一對多多型關聯與簡單的一對多關聯類似，不過，目標模型可以在一個關聯中從屬於多個模型。假設應用中的使用者可以同時「評論」文章和視訊。使用多型關聯，可以用單個 comments 表同時滿足這些情況。我們還是先來看看用來建構這種關聯的表結構：

```
posts
    id - integer
    title - string
    body - text

videos
    id - integer
    title - string
    url - string

comments
    id - integer
    body - text
    commentable_id - integer
    commentable_type - string
```

48.4.2.2 模型結構

接下來，看看建構這種關聯的模型定義：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphTo;

class Comment extends Model
{
    /**
     * 獲取擁有此評論的模型（Post 或 Video）。
     */
    public function commentable(): MorphTo
    {
        return $this->morphTo();
    }
}

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphMany;

class Post extends Model
{
    /**
     * 獲取此文章的所有評論
```

```
     */
    public function comments(): MorphMany
    {
        return $this->morphMany(Comment::class, 'commentable');
    }
}

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphMany;

class Video extends Model
{
    /**
     * 獲取此視訊的所有評論
     */
    public function comments(): MorphMany
    {
        return $this->morphMany(Comment::class, 'commentable');
    }
}
```

48.4.2.3 獲取關聯

一旦定義了資料庫表和模型，就可以通過模型訪問關聯。例如，可以使用 `comments` 動態屬性訪問文章的全部評論：

```
use App\Models\Post;

$post = Post::find(1);

foreach ($post->comments as $comment) {
    // ...
}
```

你還可以通過訪問執行對 `morphTo` 的呼叫的方法名來從多型模型獲取其所屬模型。在我們的例子中，這就是 `Comment` 模型上的 `commentable` 方法。因此，我們將以動態屬性的形式訪問該方法：

```
use App\Models\Comment;

$comment = Comment::find(1);

$commentable = $comment->commentable;
```

`Comment` 模型的 `commentable` 關聯將返回 `Post` 或 `Video` 實例，其結果取決於評論所屬的模型。

48.4.3 一對多檢索（多型）

有時一個模型可能有許多相關模型，要檢索關係的「最新」或「最舊」相關模型。 例如，一個 `User` 模型可能與許多 `Image` 模型相關，如果你想自訂一種方便的方式來與使用者上傳的最新圖像進行互動。 可以使用 `morphOne` 關係類型結合 `ofMany` 方法來完成此操作：

```
/**
 * 獲取使用者最近上傳的圖像。
 */
public function latestImage(): MorphOne
{
    return $this->morphOne(Image::class, 'imageable')->latestOfMany();
}
```

同樣，你也可以定義一個方法來檢索關係的「最早」或第一個相關模型：

```
/**
 * 獲取使用者最早上傳的圖像。
 */
public function oldestImage(): MorphOne
```

```
{
    return $this->morphOne(Image::class, 'imageable')->oldestOfMany();
}
```

默認情況下，`latestOfMany` 和 `oldestOfMany` 方法將基於模型的主鍵（必須可排序）檢索最新或最舊的相關模型。但是，有時你可能希望使用不同的排序條件從較大的關係中檢索單個模型。

例如，使用 `ofMany` 方法，可以檢索使用者點贊最高的圖像。`ofMany` 方法接受可排序列作為其第一個參數，以及在查詢相關模型時應用哪個聚合函數（`min` 或 `max`）：

```
/**
 * 獲取使用者最受歡迎的圖像。
 */
public function bestImage(): MorphOne
{
    return $this->morphOne(Image::class, 'imageable')->ofMany('likes', 'max');
}
```

提示 要建構更高級的「一對多」關係。 請查看 進階一對多檢索.

48.4.4 多對多（多型）

48.4.4.1 表結構

多對多多型關聯比 `morphOne` 和 `morphMany` 關聯略微複雜一些。例如，`Post` 和 `Video` 模型能夠共享關聯到 `Tag` 模型的多型關係。在這種情況下使用多對多多型關聯允許使用一個唯一標籤在部落格文章和視訊間共享。以下是多對多多型關聯的表結構：

```
posts
    id - integer
    name - string

videos
    id - integer
    name - string

tags
    id - integer
    name - string

taggables
    tag_id - integer
    taggable_id - integer
    taggable_type - string
```

提示 在深入研究多型多對多關係之前，閱讀 多對多關係 的文件會對你有幫助。

48.4.4.2 模型結構

接下來，我們可以定義模型之間的關聯。`Post` 和 `Video` 模型都將包含一個 `tags` 方法，該方法呼叫了基礎 Eloquent 模型類提供的 `morphToMany` 方法。

`morphToMany` 方法接受相關模型的名稱以及“關係名稱”。根據我們分配給中間表的名稱及其包含的鍵，我們將將關係稱為 「taggable」：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphToMany;
```

```
class Post extends Model
{
     /**
      * 獲取帖子的所有標籤。
      */
     public function tags(): MorphToMany
     {
          return $this->morphToMany(Tag::class, 'taggable');
     }
}
```

48.4.4.3 定義多對多（多型）反向關係

接下來, 在這個 `Tag` 模型中, 你應該為每個可能的父模型定義一個方法. 所以, 在這個例子中, 我們將會定義一個 `posts` 方法 和 一個 `videos` 方法. 這兩個方法都應該返回 `morphedByMany` 結果。

`morphedByMany` 方法接受相關模型的名稱以及「關係名稱」。根據我們分配給中間表名的名稱及其包含的鍵，我們將該關係稱為「taggable」：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphToMany;

class Tag extends Model
{
     /**
      * 獲取分配給此標籤的所有帖子。
      */
     public function posts(): MorphToMany
     {
          return $this->morphedByMany(Post::class, 'taggable');
     }

     /**
      * 獲取分配給此視訊的所有帖子.
      */
     public function videos(): MorphToMany
     {
          return $this->morphedByMany(Video::class, 'taggable');
     }
}
```

48.4.4.4 獲取關聯

一旦定義了資料庫表和模型，你就可以通過模型訪問關係。 例如，要訪問帖子的所有標籤，你可以使用 `tags` 動態關係屬性：

```
use App\Models\Post;

$post = Post::find(1);

foreach ($post->tags as $tag) {
     // ...
}
```

還可以訪問執行 `morphedByMany` 方法呼叫的方法名來從多型模型獲取其所屬模型。在這個示例中，就是 `Tag` 模型的 `posts` 或 `videos` 方法。可以像動態屬性一樣訪問這些方法：

```
use App\Models\Tag;

$tag = Tag::find(1);
```

```
foreach ($tag->posts as $post) {
    // ...
}

foreach ($tag->videos as $video) {
    // ...
}
```

48.4.5 自訂多型類型

默認情況下，Laravel 將使用完全限定的類名來儲存相關模型的「類型」。 例如，給定上面的一對多關係示例，其中 `Comment` 模型可能屬於 `Post` 或 `Video` 模型，默認的 `commentable_type` 將分別是 `App\Models\Post` 或 `App\Models\Video`。 但是，你可能希望將這些值與應用程式的內部結構解耦。

例如，我們可以使用簡單的字串，例如 `post` 和 `video`，而不是使用模型名稱作為「類型」。 通過這樣做，即使模型被重新命名，我們資料庫中的多型「類型」列值也將保持有效：

```
use Illuminate\Database\Eloquent\Relations\Relation;

Relation::enforceMorphMap([
    'post' => 'App\Models\Post',
    'video' => 'App\Models\Video',
]);
```

你可以在 `App\Providers\AppServiceProvider` 類的 `boot` 方法中呼叫 `enforceMorphMap` 方法，或者你也可以建立一個單獨的服務提供者。

你可以在執行階段使用 `getMorphClass` 方法確定給定模型的別名。相反，可以使用 `Relation::getMorphedModel` 方法來確定與別名相關聯的類名：

```
use Illuminate\Database\Eloquent\Relations\Relation;

$alias = $post->getMorphClass();

$class = Relation::getMorphedModel($alias);
```

> **注意** 向現有應用程式新增「變形對應」時，資料庫中仍包含完全限定類的每個可變形 `*_type` 列值都需要轉換為其「對應」名稱。

48.4.6 動態關聯

你可以使用 `resolveRelationUsing` 方法在執行階段定義 Eloquent 模型之間的關係。雖然通常不建議在常規應用程式開發中使用它，但是在開發 Laravel 軟體包時，這有時可能會很有用。

`resolveRelationUsing` 方法的第一個參數是關聯名稱。傳遞給該方法的第二個參數應該是一個閉包，閉包接受模型實例並返回一個有效的 Eloquent 關聯定義。通常情況下，你應該在服務提供器的啟動方法中組態動態關聯:

```
use App\Models\Order;
use App\Models\Customer;

Order::resolveRelationUsing('customer', function (Order $orderModel) {
    return $orderModel->belongsTo(Customer::class, 'customer_id');
});
```

> **注意**
> 定義動態關係時，始終為 Eloquent 關係方法提供顯式的鍵名參數。

48.5 查詢關聯

因為所有的 Eloquent 關聯都是通過方法定義的，你可以呼叫這些方法來獲取關聯的實例，而無需真實執行查詢來獲取相關的模型。此外，所有的 Eloquent 關聯也可以用作查詢構造器，允許你在最終對資料庫執行 SQL 查詢之前，繼續通過鏈式呼叫新增約束條件。

例如，假設有一個部落格系統，它的 `User` 模型有許多關聯的 `Post` 模型:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasMany;

class User extends Model
{
     /**
      * 獲取該使用者的所有文章.
      */
     public function posts(): HasMany
     {
          return $this->hasMany(Post::class);
     }
}
```

你可以查詢 `posts` 關聯，並給它新增額外的約束條件，如下例所示:

```
use App\Models\User;

$user = User::find(1);

$user->posts()->where('active', 1)->get();
```

你可以在關聯上使用任意的 查詢構造器 方法，所以一定要閱讀查詢構造器的文件，瞭解它的所有方法，這會對你非常有用。

48.5.1.1 在關聯之後鏈式新增 `orWhere` 子句

如上例所示，你可以在查詢關聯時，自由的給關聯新增額外的約束條件。但是，在將 `orWhere` 子句連結到關聯上時，一定要小心，因為 `orWhere` 子句將在邏輯上與關聯約束處於同一等級：

```
$user->posts()
         ->where('active', 1)
         ->orWhere('votes', '>=', 100)
         ->get();
```

上面的例子將生成以下 SQL。像你看到的那樣， 這個 `or` 子句的查詢指令，將返回大於 100 票的任一使用者，查詢不再限於特定的使用者：

```
select *
from posts
where user_id = ? and active = 1 or votes >= 100
```

在大多數情況下，你應該使用 邏輯分組 在括號中對條件檢查進行分組：

```
use Illuminate\Database\Eloquent\Builder;

$user->posts()
         ->where(function (Builder $query) {
             return $query->where('active', 1)
                          ->orWhere('votes', '>=', 100);
         })
         ->get();
```

上面的示例將生成以下 SQL。 請注意，邏輯分組已正確分組約束，並且查詢仍然受限於特定使用者：

```
select *
from posts
where user_id = ? and (active = 1 or votes >= 100)
```

48.5.2 關聯方法 VS 動態屬性

如果你不需要向 Eloquent 關聯查詢新增額外的約束，你可以像訪問屬性一樣訪問關聯。 例如，繼續使用我們的 `User` 和 `Post` 示例模型，我們可以像這樣訪問使用者的所有帖子：

```
use App\Models\User;

$user = User::find(1);

foreach ($user->posts as $post) {
    // ...
}
```

動態屬性是 「懶載入」 的，只有實際訪問到才會載入關聯資料。因此，通常用 預載入 來準備模型需要用到的關聯資料。預載入能大量減少因載入模型關聯執行的 SQL 語句。

48.5.3 基於存在的關聯查詢

在檢索模型記錄時，你可能希望基於關係的存在限制結果。例如，假設你想檢索至少有一條評論的所有部落格文章。為了實現這一點，你可以將關係名稱傳遞給 `has` 和 `orHas` 方法：

```
use App\Models\Post;

// 檢索所有至少有一條評論的文章...
$posts = Post::has('comments')->get();
```

也可以指定運算子和數量來進一步自訂查詢：

```
// 檢索所有有三條或更多評論的文章...
$posts = Post::has('comments', '>=', 3)->get();
```

可以使用用「.」語法構造巢狀的 `has` 語句。例如，你可以檢索包含至少一張圖片的評論的所有文章：

```
// 查出至少有一條帶圖片的評論的文章...
$posts = Post::has('comments.images')->get();
```

如果你需要更多的功能，你可以使用 `whereHas` 和 `orWhereHas` 方法在 `has` 查詢中定義額外的查詢約束，例如檢查評論的內容：

```
use Illuminate\Database\Eloquent\Builder;

// 檢索至少有一條評論包含類似於 code% 單詞的文章...
$posts = Post::whereHas('comments', function (Builder $query) {
    $query->where('content', 'like', 'code%');
})->get();

// 檢索至少有十條評論包含類似於 code% 單詞的文章...
$posts = Post::whereHas('comments', function (Builder $query) {
    $query->where('content', 'like', 'code%');
}, '>=', 10)->get();
```

注意 Eloquent 目前不支援跨資料庫查詢關係是否存在。 這些關係必須存在於同一資料庫中。

48.5.3.1 內聯關係存在查詢

如果你想使用附加到關係查詢簡單的 where 條件來確認關係是否存在，使用 `whereRelation`, `orWhereRelation`, `whereMorphRelation` 和 `orWhereMorphRelation` 方法更方便. 例如，查詢所有評

論未獲批准的帖子:

```
use App\Models\Post;

$posts = Post::whereRelation('comments', 'is_approved', false)->get();
```

當然，就像呼叫查詢建構器的 `where` 方法一樣，你也可以指定一個運算子：

```
$posts = Post::whereRelation(
    'comments', 'created_at', '>=', now()->subHour()
)->get();
```

48.5.4 查詢不存在的關聯

檢索模型記錄時，你可能會根據不存在關係來限制結果。例如，要檢索所有**沒有**任何評論的所有部落格文章。可以將關係的名稱傳遞給 `doesntHave` 和 `orDoesntHave` 方法：

```
use App\Models\Post;

$posts = Post::doesntHave('comments')->get();
```

如果需要更多功能，可以使用 `whereDoesntHave` 和 `orWhereDoesntHave` 方法將「where」 條件加到 `doesntHave` 查詢上。這些方法允許你向關聯加入自訂限制，比如檢測評論內容：

```
use Illuminate\Database\Eloquent\Builder;

$posts = Post::whereDoesntHave('comments', function (Builder $query) {
    $query->where('content', 'like', 'code%');
})->get();
```

你可以使用「點」符號對巢狀關係執行查詢。例如，以下查詢將檢索所有沒有評論的帖子；但是，有未被禁止的作者評論的帖子將包含在結果中:

```
use Illuminate\Database\Eloquent\Builder;

$posts = Post::whereDoesntHave('comments.author', function (Builder $query) {
    $query->where('banned', 0);
})->get();
```

48.5.5 查詢多型關聯

要查詢「多型關聯」的存在，可以使用 `whereHasMorph` 和 `whereDoesntHaveMorph` 方法。這些方法接受關聯名稱作為它們的第一個參數。接下來，這些方法接受你希望在查詢中包含的相關模型的名稱。最後，你可以提供一個閉包來自訂關聯查詢。

```
use App\Models\Comment;
use App\Models\Post;
use App\Models\Video;
use Illuminate\Database\Eloquent\Builder;

// 檢索與標題類似於 code% 的帖子或視訊相關的評論。
$comments = Comment::whereHasMorph(
    'commentable',
    [Post::class, Video::class],
    function (Builder $query) {
        $query->where('title', 'like', 'code%');
    }
)->get();

// 檢索與標題不類似於 code% 的帖子相關的評論。
$comments = Comment::whereDoesntHaveMorph(
    'commentable',
    Post::class,
    function (Builder $query) {
        $query->where('title', 'like', 'code%');
```

```
    }
)->get();
```

你可能需要根據相關多型模型的「類型」新增查詢約束。傳遞給 whereHasMorph 方法的閉包可以接收 $type 值作為其第二個參數。此參數允許你檢查正在建構的查詢的「類型」：

```
use Illuminate\Database\Eloquent\Builder;

$comments = Comment::whereHasMorph(
    'commentable',
    [Post::class, Video::class],
    function (Builder $query, string $type) {
        $column = $type === Post::class ? 'content' : 'title';

        $query->where($column, 'like', 'code%');
    }
)->get();
```

48.5.5.1 查詢所有相關模型

你可以使用萬用字元 * 代替多型模型的陣列，這將告訴 Laravel 從資料庫中檢索所有可能的多型類型。為了執行此操作，Laravel 將執行額外的查詢：

```
use Illuminate\Database\Eloquent\Builder;

$comments = Comment::whereHasMorph('commentable', '*', function (Builder $query) {
    $query->where('title', 'like', 'foo%');
})->get();
```

48.6 聚合相關模型

48.6.1 計算相關模型的數量

有時候你可能想要計算給定關係的相關模型的數量，而不實際載入模型。為了實現這一點，你可以使用 withCount 方法。withCount 方法將在生成的模型中放置一個 {relation}_count 屬性：

```
use App\Models\Post;

$posts = Post::withCount('comments')->get();

foreach ($posts as $post) {
    echo $post->comments_count;
}
```

通過將陣列傳遞給 withCount 方法，你可以同時新增多個關係的 “計數”，並向查詢新增其他約束：

```
use Illuminate\Database\Eloquent\Builder;

$posts = Post::withCount(['votes', 'comments' => function (Builder $query) {
    $query->where('content', 'like', 'code%');
}])->get();

echo $posts[0]->votes_count;
echo $posts[0]->comments_count;
```

你還可以給關係計數結果起別名，從而在同一關係上進行多個計數：

```
use Illuminate\Database\Eloquent\Builder;

$posts = Post::withCount([
    'comments',
    'comments as pending_comments_count' => function (Builder $query) {
        $query->where('approved', false);
    },
```

```
])->get();

echo $posts[0]->comments_count;
echo $posts[0]->pending_comments_count;
```

48.6.1.1 延遲計數載入

使用 `loadCount` 方法，你可以在獲取父模型後載入關係計數：

```
$book = Book::first();

$book->loadCount('genres');
```

如果你需要在計數查詢上設定其他查詢約束，你可以傳遞一個以你想要計數的關係為鍵的陣列。陣列的值應該是接收查詢建構器實例的閉包：

```
$book->loadCount(['reviews' => function (Builder $query) {
    $query->where('rating', 5);
}])
```

48.6.1.2 關聯計數和自訂查詢欄位

如果你的查詢同時包含 `withCount` 和 `select`，請確保 `withCount` 一定在 `select` 之後呼叫：

```
$posts = Post::select(['title', 'body'])
                ->withCount('comments')
                ->get();
```

48.6.2 其他聚合函數

除了 `withCount` 方法外，Eloquent 還提供了 `withMin`, `withMax`, `withAvg` 和 `withSum` 等聚合方法。 這些方法會通過 `{relation}_{function}_{column}` 的命名方式將聚合結果新增到獲取到的模型屬性中：

```
use App\Models\Post;

$posts = Post::withSum('comments', 'votes')->get();

foreach ($posts as $post) {
    echo $post->comments_sum_votes;
}
```

如果你想使用其他名稱訪問聚合函數的結果，可以自訂的別名：

```
$posts = Post::withSum('comments as total_comments', 'votes')->get();

foreach ($posts as $post) {
    echo $post->total_comments;
}
```

與 `loadCount` 方法類似，這些方法也有延遲呼叫的方法。這些延遲方法可在已獲取到的 Eloquent 模型上呼叫：

```
$post = Post::first();

$post->loadSum('comments', 'votes');
```

如果你將這些聚合方法和一個 `select` 語句組合在一起，確保你在 `select` 方法之後呼叫聚合方法:

```
$posts = Post::select(['title', 'body'])
                ->withExists('comments')
                ->get();
```

48.6.3 多型關聯計數

如果你想預載入多型關聯關係以及這個關聯關係關聯的其他關聯關係的計數統計，可以通過將 `with` 方法與

morphTo 關係和 morphWithCount 方法結合來實現。

在這個例子中，我們假設 Photo 和 Post 模型可以建立 ActivityFeed 模型。 我們將假設 ActivityFeed 模型定義了一個名為 parentable 的多型關聯關係，它允許我們為給定的 ActivityFeed 實例檢索父級 Photo 或 Post 模型。 此外，讓我們假設 Photo 模型有很多 Tag 模型、Post 模型有很多 Comment 模型。

假如我們想要檢索 ActivityFeed 實例並為每個 ActivityFeed 實例預先載入 parentable 父模型。 此外，我們想要檢索與每張父照片關聯的標籤數量以及與每個父帖子關聯的評論數量：

```
use Illuminate\Database\Eloquent\Relations\MorphTo;

$activities = ActivityFeed::with([
     'parentable' => function (MorphTo $morphTo) {
         $morphTo->morphWithCount([
              Photo::class => ['tags'],
              Post::class => ['comments'],
         ]);
     }])->get();
```

48.6.3.1 延遲計數載入

假設我們已經檢索了一組 ActivityFeed 模型，現在我們想要載入與活動提要關聯的各種 parentable 模型的巢狀關係計數。 可以使用 loadMorphCount 方法來完成此操作：

```
$activities = ActivityFeed::with('parentable')->get();

$activities->loadMorphCount('parentable', [
     Photo::class => ['tags'],
     Post::class => ['comments'],
]);
```

48.7 預載入

當將 Eloquent 關係作為屬性訪問時，相關模型是延遲載入的。 這意味著在你第一次訪問該屬性之前不會實際載入關聯資料。 但是，Eloquent 可以在查詢父模型時主動載入關聯關係。 預載入減輕了 N + 1 查詢問題。 為了說明 N + 1 查詢問題，請參考屬於 Author 模型的 Book 模型：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Book extends Model
{
     /**
      * 獲取寫了這本書的作者。
      */
     public function author(): BelongsTo
     {
          return $this->belongsTo(Author::class);
     }
}
```

現在，讓我們檢索所有書籍及其作者：

```
use App\Models\Book;

$books = Book::all();

foreach ($books as $book) {
     echo $book->author->name;
```

```
}
```

該循環將執行一個查詢以檢索資料庫表中的所有書籍，然後對每本書執行另一個查詢以檢索該書的作者。 因此，如果我們有 25 本書，上面的程式碼將運行 26 個查詢：一個查詢原本的書籍資訊，另外 25 個查詢來檢索每本書的作者。

值得慶幸的是，我們可以使用預載入將這個操作減少到兩個查詢。 在建構查詢時，可以使用 `with` 方法指定應該預載入哪些關係：

```
$books = Book::with('author')->get();

foreach ($books as $book) {
    echo $book->author->name;
}
```

對於此操作，將只執行兩個查詢 - 一個查詢檢索書籍，一個查詢檢索所有書籍的作者：

```
select * from books

select * from authors where id in (1, 2, 3, 4, 5, ...)
```

48.7.1.1 預載入多個關聯

有時，你可能需要在單一操作中預載入幾個不同的關聯。要達成此目的，只要向 `with` 方法傳遞多個關聯名稱構成的陣列參數：

```
$books = Book::with(['author', 'publisher'])->get();
```

48.7.1.2 巢狀預載入

可以使用 「.」 語法預載入巢狀關聯。比如在一個 Eloquent 語句中預載入所有書籍作者及其聯絡方式：

```
$books = Book::with('author.contacts')->get();
```

另外，你可以通過向 `with` 方法提供巢狀陣列來指定巢狀的預載入關係，這在預載入多個巢狀關係時非常方便。

```
$books = Book::with([
    'author' => [
        'contacts',
        'publisher',
    ],
])->get();
```

48.7.1.3 巢狀預載入 `morphTo` 關聯

如果你希望載入一個 `morphTo` 關係，以及該關係可能返回的各種實體的巢狀關係，可以將 `with` 方法與 `morphTo` 關係的 `morphWith` 方法結合使用。為了說明這種方法，讓我們參考以下模型：

```
<?php

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphTo;

class ActivityFeed extends Model
{
    /**
     * 獲取活動記錄的父記錄。
     */
    public function parentable(): MorphTo
    {
        return $this->morphTo();
    }
}
```

在這個例子中，我們假設 Event，Photo 和 Post 模型可以建立 ActivityFeed 模型。 另外，我們假設 Event 模型屬於 Calendar 模型，Photo 模型與 Tag 模型相關聯，Post 模型屬於 Author 模型。

使用這些模型定義和關聯，我們可以查詢 ActivityFeed 模型實例並預載入所有 parentable 模型及其各自的巢狀關係：

```
use Illuminate\Database\Eloquent\Relations\MorphTo;

$activities = ActivityFeed::query()
     ->with(['parentable' => function (MorphTo $morphTo) {
          $morphTo->morphWith([
               Event::class => ['calendar'],
               Photo::class => ['tags'],
               Post::class => ['author'],
          ]);
     }])->get();
```

48.7.1.4 預載入指定列

並不是總需要獲取關係的每一列。在這種情況下，Eloquent 允許你為關聯指定想要獲取的列:

```
$books = Book::with('author:id,name,book_id')->get();
```

注意 使用此功能時，應始終在要檢索的列列表中包括 id 列和任何相關的外部索引鍵列。

48.7.1.5 默認預載入

有時可能希望在查詢模型時始終載入某些關聯。 為此，你可以在模型上定義 $with 屬性

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Book extends Model
{
     /**
      * 默認預載入的關聯。
      *
      * @var array
      */
     protected $with = ['author'];

     /**
      * 獲取書籍作者。
      */
     public function author(): BelongsTo
     {
          return $this->belongsTo(Author::class);
     }

     /**
      * 獲取書籍類型。
      */
     public function genre(): BelongsTo
     {
          return $this->belongsTo(Genre::class);
     }
}
```

如果你想從單個查詢的 $with 屬性中刪除一個預載入，你可以使用 without 方法：

```
$books = Book::without('author')->get();
```

如果你想要覆蓋 `$with` 屬性中所有項，僅針對單個查詢，你可以使用 `withOnly` 方法：

```
$books = Book::withOnly('genre')->get();
```

48.7.2 約束預載入

有時，你可能希望預載入一個關聯，同時為預載入查詢新增額外查詢條件。你可以通過將一個關聯陣列傳遞給 `with` 方法來實現這一點，其中陣列鍵是關聯名稱，陣列值是一個閉包，它為預先載入查詢新增了額外的約束：

```
use App\Models\User;

$users = User::with(['posts' => function (Builder $query) {
    $query->where('title', 'like', '%code%');
}])->get();
```

在這個例子中，Eloquent 只會預載入帖子的 `title` 列包含單詞 `code` 的帖子。 你可以呼叫其他 查詢構造器 方法來自訂預載入操作：

```
$users = User::with(['posts' => function (Builder $query) {
    $query->orderBy('created_at', 'desc');
}])->get();
```

> **注意** 在約束預載入時，不能使用 limit 和 take 查詢構造器方法。

48.7.2.1 `morphTo` 關聯預載入新增約束

如果你在使用 Eloquent 進行 `morphTo` 關聯的預載入時，Eloquent 將運行多個查詢以獲取每種類型的相關模型。你可以使用 `MorphTo` 關聯的 `constrain` 方法向每個查詢新增附加約束條件：

```
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Relations\MorphTo;

$comments = Comment::with(['commentable' => function (MorphTo $morphTo) {
    $morphTo->constrain([
        Post::class => function (Builder $query) {
            $query->whereNull('hidden_at');
        },
        Video::class => function (Builder $query) {
            $query->where('type', 'educational');
        },
    ]);
}])->get();
```

在這個例子中，Eloquent 只會預先載入未被隱藏的帖子，並且視訊的 `type` 值為 `educational`。

48.7.2.2 基於存在限制預載入

有時候，你可能需要同時檢查關係的存在性並根據相同條件載入關係。例如，你可能希望僅查詢具有符合給定條件的子模型 `Post` 的 `User` 模型，同時也預載入匹配的文章。你可以使用 Laravel 中的 `withWhereHas` 方法來實現這一點。

```
use App\Models\User;
use Illuminate\Database\Eloquent\Builder;

$users = User::withWhereHas('posts', function (Builder $query) {
    $query->where('featured', true);
})->get();
```

48.7.3 延遲預載入

有時你可能需要在查詢父模型之後預載入關聯。例如，如果你需要動態地決定是否載入相關模型，則這可能非常有用：

```
use App\Models\Book;

$books = Book::all();

if ($someCondition) {
    $books->load('author', 'publisher');
}
```

如果要在渴求式載入的查詢語句中進行條件約束，可以通過陣列的形式去載入，鍵為對應的關聯關係，值為 `Closure` 閉包涵數，該閉包的參數為一個查詢實例：

```
$author->load(['books' => function (Builder $query) {
    $query->orderBy('published_date', 'asc');
}]);
```

如果希望僅載入未被載入的關聯關係時，你可以使用 `loadMissing` 方法：

```
$book->loadMissing('author');
```

48.7.3.1 巢狀延遲預載入 & `morphTo`

如果要預載入 `morphTo` 關係，以及該關係可能返回的各種實體上的巢狀關係，你可以使用 `loadMorph` 方法。

這個方法接受 `morphTo` 關係的名稱作為它的第一個參數，第二個參數接收模型陣列、關係陣列。例如:

```
<?php

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\MorphTo;

class ActivityFeed extends Model
{
    /**
     * 獲取活動提要記錄的父項。
     */
    public function parentable(): MorphTo
    {
        return $this->morphTo();
    }
}
```

在這個例子中，讓我們假設 `Event` 、`Photo` 和 `Post` 模型可以建立 `ActivityFeed` 模型。此外，讓我們假設 `Event` 模型屬於 `Calendar` 模型，`Photo` 模型與 `Tag` 模型相關聯，`Post` 模型屬於 `Author` 模型。

使用這些模型定義和關聯關係，我們方可以檢索 `ActivityFeed` 模型實例，並立即載入所有 `parentable` 模型及其各自的巢狀關係：

```
$activities = ActivityFeed::with('parentable')
    ->get()
    ->loadMorph('parentable', [
        Event::class => ['calendar'],
        Photo::class => ['tags'],
        Post::class => ['author'],
    ]);
```

48.7.4 防止延遲載入

如前所述，預載入關係可以為應用程式提供顯著的性能優勢。 但你也可以指示 Laravel 始終防止延遲載入關係。 你可以呼叫基本 Eloquent 模型類提供的 `preventLazyLoading` 方法。 通常，你應該在應用程式的 `AppServiceProvider` 類的 `boot` 方法中呼叫此方法。

`preventLazyLoading` 方法接受一個可選的布林值類型的參數，表示是否阻止延遲載入。例如，你可能希望只在非生產環境中停用延遲載入，這樣即使在生產環境程式碼中意外出現了延遲載入關係，你的生產環境也能繼續正常運行。

```
use Illuminate\Database\Eloquent\Model;

/**
 * 引導應用程式服務。
 */
public function boot(): void
{
    Model::preventLazyLoading(! $this->app->isProduction());
}
```

在阻止延遲載入之後，當你的應用程式嘗試延遲載入任何 Eloquent 關係時，Eloquent 將拋出 `Illuminate\Database\LazyLoadingViolationException` 異常。

你可以使用 `handleLazyLoadingViolationsUsing` 方法自訂延遲載入的違規行為。例如，使用此方法，你可以指示違規行為只被記錄，而不是使用異常中斷應用程式的執行：

```
Model::handleLazyLoadingViolationUsing(function (Model $model, string $relation) {
    $class = get_class($model);

    info("Attempted to lazy load [{$relation}] on model [{$class}].");
});
```

48.8 插入 & 更新關聯模型

48.8.1 save 方法

Eloquent 提供了向關係中新增新模型的便捷方法。例如，你可能需要向一篇文章（`Post` 模型）新增一條新的評論（`Comment` 模型），你不用手動設定 `Comment` 模型上的 `post_id` 屬性，你可以直接使用關聯模型中的 `save` 方法：

```
use App\Models\Comment;
use App\Models\Post;

$comment = new Comment(['message' => 'A new comment.']);

$post = Post::find(1);

$post->comments()->save($comment);
```

注意，我們沒有將 `comments` 關聯作為動態屬性訪問，相反，我們呼叫了 `comments` 方法來來獲得關聯實例， `save` 方法會自動新增適當的 `post_id` 值到新的 `Comment` 模型中。

如果需要保存多個關聯模型，你可以使用 `saveMany` 方法：

```
$post = Post::find(1);

$post->comments()->saveMany([
    new Comment(['message' => 'A new comment.']),
    new Comment(['message' => 'Another new comment.']),
]);
```

save 和 saveMany 方法不會將新模型（Comment）載入到父模型（Post) 上， 如果你計畫在使用 save 或 saveMany 方法後訪問該關聯模型（Comment），你需要使用 refresh 方法重新載入模型及其關聯，這樣你就可以訪問到所有評論，包括新保存的評論了：

```
$post->comments()->save($comment);

$post->refresh();

// 所有評論，包括新保存的評論...
$post->comments;
```

48.8.1.1 遞迴保存模型和關聯資料

如果你想 save 模型及其所有關聯資料，你可以使用 push 方法，在此示例中，將保存 Post 模型及其評論和評論作者：

```
$post = Post::find(1);

$post->comments[0]->message = 'Message';
$post->comments[0]->author->name = 'Author Name';

$post->push();
```

pushQuietly 方法可用於保存模型及其關聯關係，而不觸發任何事件：

```
$post->pushQuietly();
```

48.8.2 create 方法

除了 save 和 saveMany 方法外，你還可以使用 create 方法。它接受一個屬性陣列，同時會建立模型並插入到資料庫中。 還有， save 和 create 方法的不同之處在於， save 方法接受一個完整的 Eloquent 模型實例，而 create 則接受普通的 PHP 陣列：

```
use App\Models\Post;

$post = Post::find(1);

$comment = $post->comments()->create([
    'message' => 'A new comment.',
]);
```

你還可以使用 createMany 方法去建立多個關聯模型：

```
$post = Post::find(1);

$post->comments()->createMany([
    ['message' => 'A new comment.'],
    ['message' => 'Another new comment.'],
]);
```

還可以使用 createQuietly 和 createManyQuietly 方法建立模型，而無需調度任何事件：

```
$user = User::find(1);

$user->posts()->createQuietly([
    'title' => 'Post title.',
]);

$user->posts()->createManyQuietly([
    ['title' => 'First post.'],
    ['title' => 'Second post.'],
]);
```

你還可以使用 findOrNew, firstOrNew, firstOrCreate 和 updateOrCreate 方法來 建立和更新關係模型。

注意：在使用 `create` 方法前，請務必確保查看過本文件的 批次賦值 章節。

48.8.3 Belongs To 關聯

如果你想將子模型分配給新的父模型，你可以使用 `associate` 方法。在這個例子中，`User` 模型定義了一個與 `Account` 模型的 `belongsTo` 關係。 這個 `associate` 方法將在子模型上設定外部索引鍵：

```
use App\Models\Account;

$account = Account::find(10);

$user->account()->associate($account);

$user->save();
```

要從子模型中刪除父模型，你可以使用 `dissociate` 方法。此方法會將關聯外部索引鍵設定為 `null`：

```
$user->account()->dissociate();

$user->save();
```

48.8.4 多對多關聯

48.8.4.1 附加 / 分離

Eloquent 也提供了一些額外的輔助方法，使相關模型的使用更加方便。例如，我們假設一個使用者可以擁有多個角色，並且每個角色都可以被多個使用者共享。給某個使用者附加一個角色是通過向中間表插入一條記錄實現的，可以使用 `attach` 方法完成該操作：

```
use App\Models\User;

$user = User::find(1);

$user->roles()->attach($roleId);
```

在將關係附加到模型時，還可以傳遞一組要插入到中間表中的附加資料：

```
$user->roles()->attach($roleId, ['expires' => $expires]);
```

當然，有時也需要移除使用者的角色。可以使用 `detach` 移除多對多關聯記錄。`detach` 方法將會移除中間表對應的記錄。但是這兩個模型都將會保留在資料庫中:

```
// 移除使用者的一個角色...
$user->roles()->detach($roleId);

// 移除使用者的所有角色...
$user->roles()->detach();
```

為了方便起見，`attach` 和 `detach` 也允許傳遞一個 IDs 陣列：

```
$user = User::find(1);

$user->roles()->detach([1, 2, 3]);

$user->roles()->attach([
    1 => ['expires' => $expires],
    2 => ['expires' => $expires],
]);
```

48.8.4.2 同步關聯

你也可以使用 `sync` 方法建構多對多關聯。`sync` 方法接收一個 IDs 陣列以替換中間表的記錄。中間表記錄中，所有未在 IDs 陣列中的記錄都將會被移除。所以該操作結束後，只有給出陣列的 IDs 會被保留在中間表

中：

```
$user->roles()->sync([1, 2, 3]);
```

你也可以通過 IDs 傳遞額外的附加資料到中間表：

```
$user->roles()->sync([1 => ['expires' => true], 2, 3]);
```

如果你想為每個同步的模型 IDs 插入相同的中間表，你可以使用 `syncWithPivotValues` 方法：

```
$user->roles()->syncWithPivotValues([1, 2, 3], ['active' => true]);
```

如果你不想移除現有的 IDs，可以使用 `syncWithoutDetaching` 方法：

```
$user->roles()->syncWithoutDetaching([1, 2, 3]);
```

48.8.4.3 切換關聯

多對多關聯也提供了 `toggle` 方法用於「切換」給定 ID 陣列的附加狀態。 如果給定的 ID 已被附加在中間表中，那麼它將會被移除，同樣，如果給定的 ID 已被移除，它將會被附加：

```
$user->roles()->toggle([1, 2, 3]);
```

你還可以將附加的中間表值與 ID 一起傳遞：

```
$user->roles()->toggle([
    1 => ['expires' => true],
    2 => ['expires' => true],
]);
```

48.8.4.4 更新中間表上的記錄

如果你需要在中間表中更新一條已存在的記錄，可以使用 `updateExistingPivot` 方法。 此方法接收中間表的外部索引鍵與要更新的資料陣列進行更新：

```
$user = User::find(1);

$user->roles()->updateExistingPivot($roleId, [
    'active' => false,
]);
```

48.9 更新父級時間戳

當一個模型屬 `belongsTo` 或者 `belongsToMany` 另一個模型時， 例如 `Comment` 屬於 `Post` ，有時更新子模型導致更新父模型時間戳非常有用。

例如，當 `Comment` 模型被更新時，你需要自動「觸發」父級 `Post` 模型的 `updated_at` 時間戳的更新。`Eloquent` 讓它變得簡單。只要在子模型加一個包含關聯名稱的 `touches` 屬性即可：:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Comment extends Model
{
    /**
     * 需要觸發的所有關聯關係。
     *
     * @var array
     */
    protected $touches = ['post'];
```

```
    /**
     * 獲取評論所屬文章。
     */
    public function post(): BelongsTo
    {
        return $this->belongsTo(Post::class);
    }
}
```

注意：只有使用 `Eloquent` 的 `save` 方法更新子模型時，才會觸發更新父模型時間戳。

49 集合

49.1 介紹

所有以多個模型為查詢結果的 Eloquent 方法的返回值都是 `Illuminate\Database\Eloquent\Collection` 類的實例, 其中包括了通過 `get` 方法和關聯關係獲取的結果。Eloquent 集合對象擴展了 Laravel 的基礎集合類，因此它自然地繼承了許多用於流暢地處理 Eloquent 模型的底層陣列的方法。請務必查看 Laravel 集合文件以瞭解所有這些有用的方法！

所有的集合都可作為迭代器，你可以像遍歷普通的 PHP 陣列一樣來遍歷它們:

```
use App\Models\User;

$users = User::where('active', 1)->get();

foreach ($users as $user) {
    echo $user->name;
}
```

然而，正如前面所提到的，集合遠比陣列要強大，而且暴露了一系列直觀的、可用於鏈式呼叫的 map/reduce 方法。打個比方，我們可以刪除所有不活躍的模型，然後收集餘下的所有使用者的名字。

```
$names = User::all()->reject(function (User $user) {
    return $user->active === false;
})->map(function (User $user) {
    return $user->name;
});
```

49.1.1.1 Eloquent 集合轉換

在大多數 Eloquent 集合方法返回一個新的 Eloquent 集合實例的前提下，`collapse`，`flatten`，`flip`，`keys`，`pluck`，以及 `zip` 方法返回一個基礎集合類的實例。 如果一個 `map` 方法返回了一個不包含任何模型的 Eloquent 集合，它也會被轉換成一個基礎集合實例。

49.2 可用的方法

所有 Eloquent 的集合都繼承了 Laravel collection 對象；因此， 他們也繼承了所有集合基類提供的強大的方法。

另外， `Illuminate\Database\Eloquent\Collection` 類提供了一套上層的方法來幫你管理你的模型集合。大多數方法返回 `Illuminate\Database\Eloquent\Collection` 實例；然而，也會有一些方法， 例如 `modelKeys`， 它們會返回基於 `Illuminate\Support\Collection` 類的實例。

不列印可用的方法

49.3 自訂集合

如果你想在與模型互動時使用一個自訂的 `Collection` 對象，你可以通過在模型中定義 `newCollection` 方法來實現：

```
<?php

namespace App\Models;
```

```
use App\Support\UserCollection;
use Illuminate\Database\Eloquent\Collection;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 建立新的 Elquent Collection 實例。
     *
     * @param  array<int, \Illuminate\Database\Eloquent\Model>  $models
     * @return \Illuminate\Database\Eloquent\Collection<int, \Illuminate\Database\
Eloquent\Model>
     */
    public function newCollection(array $models = []): Collection
    {
        return new UserCollection($models);
    }
}
```

一旦在模型中定義了一個 `newCollection` 方法，每當 Eloquent 需要返回一個 `Illuminate\Database\Eloquent\Collection` 實例的時候，將會返回自訂集合的實例取代之。如果你想使每一個模型都使用自訂的集合，可以在一個模型基類中定義一個 `newCollection` 方法，然後讓其它模型派生於此基類。

50 屬性修改器

50.1 簡介

當你在 Eloquent 模型實例中獲取或設定某些屬性值時，訪問器和修改器允許你對 Eloquent 屬性值進行格式化。例如，你可能需要使用 Laravel 加密器 來加密保存在資料庫中的值，而在使用 Eloquent 模型訪問該屬性的時候自動進行解密其值。

或者，當通過 Eloquent 模型訪問儲存在資料庫的 JSON 字串時，你可能希望將其轉換為陣列。

50.2 訪問器 & 修改器

50.2.1 定義一個訪問器

訪問器會在訪問一個模型的屬性時轉換 Eloquent 值。要定義訪問器，請在模型中建立一個受保護的「駝峰式」方法來表示可訪問屬性。此方法名稱對應到真正的底層模型 屬性/資料庫欄位 的表示。

在本例中，我們將為 `first_name` 屬性定義一個訪問器。在嘗試檢索 `first_name` 屬性的值時，Eloquent 會自動呼叫訪問器。所有屬性訪問器 / 修改器方法必須聲明 `Illuminate\Database\Eloquent\Casts\Attribute` 的返回類型提示：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Casts\Attribute;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 獲取使用者的名字。
     */
    protected function firstName(): Attribute
    {
        return Attribute::make(
            get: fn (string $value) => ucfirst($value),
        );
    }
}
```

所有訪問器方法都返回一個 `Attribute` 實例，該實例定義了如何訪問該屬性以及如何改變該屬性。 在此示例中，我們僅定義如何訪問該屬性。 為此，我們將 `get` 參數提供給 `Attribute` 類建構函式。

如你所見，欄位的原始值被傳遞到訪問器中，允許你對它進行處理並返回結果。如果想獲取被修改後的值，你可以在模型實例上訪問 `first_name` 屬性：

```
use App\Models\User;

$user = User::find(1);

$firstName = $user->first_name;
```

> **注意：** 如果要將這些計算值新增到模型的 array / JSON 中表示，你需要追加它們.

50.2.1.1 從多個屬性建構值對象

有時你的訪問器可能需要將多個模型屬性轉換為單個「值對象」。 為此，你的 `get` 閉包可以接受 `$attributes` 的第二個參數，該參數將自動提供給閉包，並將包含模型所有當前屬性的陣列：

```
use App\Support\Address;
use Illuminate\Database\Eloquent\Casts\Attribute;

/**
 * 與使用者地址互動。
 */
protected function address(): Attribute
{
    return Attribute::make(
        get: fn (mixed $value, array $attributes) => new Address(
            $attributes['address_line_one'],
            $attributes['address_line_two'],
        ),
    );
}
```

50.2.1.2 訪問器快取

從訪問器返回值對象時，對值對象所做的任何更改都將在模型保存之前自動同步回模型。 這是可能的，因為 Eloquent 保留了訪問器返回的實例，因此每次呼叫訪問器時都可以返回相同的實例：

```
use App\Models\User;

$user = User::find(1);

$user->address->lineOne = 'Updated Address Line 1 Value';
$user->address->lineTwo = 'Updated Address Line 2 Value';

$user->save();
```

有時你可能希望為字串和布林值等原始值啟用快取，特別是當它們是計算密集型時。要實現這一點，你可以在定義訪問器時呼叫 `shouldCache` 方法：

```
protected function hash(): Attribute
{
    return Attribute::make(
        get: fn (string $value) => bcrypt(gzuncompress($value)),
    )->shouldCache();
}
```

如果要停用屬性的快取，可以在定義屬性時呼叫 `withoutObjectCaching` 方法：

```
/**
 * 與 user 的 address 互動。
 */
protected function address(): Attribute
{
    return Attribute::make(
        get: fn (mixed $value, array $attributes) => new Address(
            $attributes['address_line_one'],
            $attributes['address_line_two'],
        ),
    )->withoutObjectCaching();
}
```

50.2.2 定義修改器

修改器會在設定屬性時生效。要定義修改器，可以在定義屬性時提供 `set` 參數。讓我們為 `first_name` 屬性定義一個修改器。這個修改器將會在我們修改 `first_name` 屬性的值時自動呼叫：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Casts\Attribute;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 與 user 的 first name 互動。
     */
    protected function firstName(): Attribute
    {
        return Attribute::make(
            get: fn (string $value) => ucfirst($value),
            set: fn (string $value) => strtolower($value),
        );
    }
}
```

修改器的閉包會接收將要設定的值，並允許我們使用和返回該值。要使該修改器生效，只需在模型上設定 `first_name` 即可：

```
use App\Models\User;

$user = User::find(1);

$user->first_name = 'Sally';
```

在本例中，值 `Sally` 將會觸發 `set` 回呼。然後，修改器會使用 `strtolower` 函數處理姓名，並將結果值設定在模型的 `$attributes` 陣列中。

50.2.2.1 修改多個屬性

有時你的修改器可能需要修改底層模型的多個屬性。 為此，你的 `set` 閉包可以返回一個陣列，陣列中的每個鍵都應該與模型的屬性 / 資料庫列相對應：

```
use App\Support\Address;
use Illuminate\Database\Eloquent\Casts\Attribute;

/**
 * 與 user 模型的 address 互動。
 */
protected function address(): Attribute
{
    return Attribute::make(
        get: fn (mixed $value, array $attributes) => new Address(
            $attributes['address_line_one'],
            $attributes['address_line_two'],
        ),
        set: fn (Address $value) => [
            'address_line_one' => $value->lineOne,
            'address_line_two' => $value->lineTwo,
        ],
    );
}
```

50.3 屬性轉換

屬性轉換提供了類似於訪問器和修改器的功能，且無需在模型上定義任何其他方法。模型中的 `$casts` 屬性提供了一個便利的方法來將屬性轉換為常見的資料類型。

$casts 屬性應是一個陣列，且陣列的鍵是那些需要被轉換的屬性名稱，值則是你希望轉換的資料類型。支援轉換的資料類型有：

- array
- AsStringable::class
- boolean
- collection
- date
- datetime
- immutable_date
- immutable_datetime
- decimal:<precision>
- double
- encrypted
- encrypted:array
- encrypted:collection
- encrypted:object
- float
- integer
- object
- real
- string
- timestamp

示例， 讓我們把以整數（0 或 1）形式儲存在資料庫中的 is_admin 屬性轉成布林值：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 類型轉換。
     *
     * @var array
     */
    protected $casts = [
        'is_admin' => 'boolean',
    ];
}
```

現在當你訪問 is_admin 屬性時，雖然保存在資料庫裡的值是一個整數類型，但是返回值總是會被轉換成布林值類型：

```
$user = App\Models\User::find(1);

if ($user->is_admin) {
    // ...
}
```

如果需要在執行階段新增新的臨時強制轉換，可以使用 mergeCasts 這些強制轉換定義將新增到模型上已定義的任何強制轉換中：

```
$user->mergeCasts([
    'is_admin' => 'integer',
    'options' => 'object',
]);
```

> **注意**： 值屬性將不會被轉換。此外，禁止定義與關聯同名的類型轉換（或屬性）。

50.3.1.1 強制轉換

你可以用 Illuminate\Database\Eloquent\Casts\AsStringable 類將模型屬性強制轉換為

Illuminate\Support\Stringable 對象:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Casts\AsStringable;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 類型轉換。
     *
     * @var array
     */
    protected $casts = [
        'directory' => AsStringable::class,
    ];
}
```

50.3.2 陣列 & JSON 轉換

當你在資料庫儲存序列化的 JSON 的資料時， array 類型的轉換非常有用。比如：如果你的資料庫具有被序列化為 JSON 的 JSON 或 TEXT 欄位類型，並且在 Eloquent 模型中加入了 array 類型轉換，那麼當你訪問的時候就會自動被轉換為 PHP 陣列：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 類型轉換。
     *
     * @var array
     */
    protected $casts = [
        'options' => 'array',
    ];
}
```

一旦定義了轉換，你訪問 options 屬性時他會自動從 JSON 類型反序列化為 PHP 陣列。當你設定了 options 屬性的值時，給定的陣列也會自動序列化為 JSON 類型儲存：

```
use App\Models\User;

$user = User::find(1);

$options = $user->options;

$options['key'] = 'value';

$user->options = $options;

$user->save();
```

當使用 update 方法更新 JSON 屬性的單個欄位時，可以使用 -> 運算子讓語法更加簡潔：

```
$user = User::find(1);

$user->update(['options->key' => 'value']);
```

50.3.2.1 陣列對象 & 集合類型轉換

雖然標準的 `array` 類型轉換對於許多應用程式來說已經足夠了，但它確實有一些缺點。由於 `array` 類型轉換返回一個基礎類型，因此不可能直接改變陣列鍵的值。例如，以下程式碼將觸發一個 PHP 錯誤：

```
$user = User::find(1);

$user->options['key'] = $value;
```

為瞭解決這個問題，Laravel 提供了一個 `AsArrayObject` 類型轉換，它將 JSON 屬性轉換為一個 陣列對象 類。這個特性是使用 Laravel 的 自訂類型轉換 實現的，它允許 Laravel 智能地快取和轉換修改的對象，這樣可以在不觸發 PHP 錯誤的情況下修改各個鍵的值。要使用 `AsArrayObject` 類型轉換，只需將其指定給一個屬性即可：

```
use Illuminate\Database\Eloquent\Casts\AsArrayObject;

/**
 * 類型轉換。
 *
 * @var array
 */
protected $casts = [
    'options' => AsArrayObject::class,
];
```

類似的，Laravel 提供了一個 `AsCollection` 類型轉換，它將 JSON 屬性轉換為 Laravel 集合 實例：

```
use Illuminate\Database\Eloquent\Casts\AsCollection;

/**
 * 類型轉換。
 *
 * @var array
 */
protected $casts = [
    'options' => AsCollection::class,
];
```

50.3.3 Date 轉換

默認情況下，Eloquent 會將 `created_at` 和 `updated_at` 欄位轉換為 Carbon 實例，它繼承了 PHP 原生的 `DateTime` 類並提供了各種有用的方法。你可以通過在模型的 `$casts` 屬性陣列中定義額外的日期類型轉換，用來轉換其他的日期屬性。通常來說，日期應該使用 `datetime` 或 `immutable_datetime` 類型轉換來轉換。

當使用 `date` 或 `datetime` 類型轉換時，你也可以指定日期的格式。這種格式會被用在 模型序列化為陣列或者 JSON：

```
/**
 * 類型轉換。
 *
 * @var array
 */
protected $casts = [
    'created_at' => 'datetime:Y-m-d',
];
```

將列類型轉換為日期時，可以將其值設定為 UNIX 時間戳、日期字串（`Y-m-d`）、日期時間字串或 `DateTime` / `Carbon` 實例。日期值將會被精準的轉換並儲存在資料庫中。

通過在模型中定義 `serializeDate` 方法，你可以自訂所有模型日期的默認序列化格式。此方法不會影響日期在資料庫中儲存的格式：

```
/**
 * 為 array / JSON 序列化準備日期格式。
 */
protected function serializeDate(DateTimeInterface $date): string
{
    return $date->format('Y-m-d');
}
```

在模型上定義 `$dateFormat` 屬性後，模型的日期將會以你指定的格式實際儲存於資料庫中：

```
/**
 * 模型日期列的儲存格式。
 *
 * @var string
 */
protected $dateFormat = 'U';
```

50.3.3.1 日期轉換，序列化，& 時區

默認情況下，`date` 和 `datetime` 會序列化為 UTC ISO-8601 格式的（`1986-05-28T21:05:54.000000Z`）字串，並不會受到應用的 `timezone` 組態影響。強烈建議您始終使用此序列化格式，並不更改應用程式的 `timezone` 組態（默認 `UTC`）以將應用程式的日期儲存在 UTC 時區中。在整個應用程式中始終使用 UTC 時區，會使與其他 PHP 和 JavaScript 類庫的互操作性更高。

如果對 `date` 或 `datetime` 屬性自訂了格式，例如 `datetime:Y-m-d H:i:s`，那麼在日期序列化期間將使用 Carbon 實例的內部時區。通常，這是應用程式的 `timezone` 組態選項中指定的時區。

50.3.4 列舉轉換

Eloquent 還允許您將屬性值強制轉換為 PHP 的 列舉。 為此，可以在模型的 `$casts` 陣列屬性中指定要轉換的屬性和列舉：

```
use App\Enums\ServerStatus;

/**
 * 類型轉換。
 *
 * @var array
 */
protected $casts = [
    'status' => ServerStatus::class,
];
```

在模型上定義了轉換後，與屬性互動時，指定的屬性都將在列舉中強制轉換：

```
if ($server->status == ServerStatus::Provisioned) {
    $server->status = ServerStatus::Ready;

    $server->save();
}
```

50.3.4.1 轉換列舉陣列

有時，你可能需要模型在單個列中儲存列舉值的陣列。為此，你可以使用 Laravel 提供的 `AsEnumArrayObject` 或 `AsEnumCollection` 強制轉換：

```
use App\Enums\ServerStatus;
use Illuminate\Database\Eloquent\Casts\AsEnumCollection;

/**
 * 類型轉換。
 *
```

```
 * @var array
 */
protected $casts = [
    'statuses' => AsEnumCollection::class.':'.ServerStatus::class,
];
```

50.3.5 加密轉換

encrypted 轉換使用了 Laravel 的內建 encryption 功能加密模型的屬性值。 此外，encrypted:array、encrypted:collection、encrypted:object、AsEncryptedArrayObject 和 AsEncryptedCollection 類型轉換的工作方式與未加密的類型相同； 但是，正如您所料，底層值在儲存在資料庫中時是加密的。

由於加密文字的最終長度不可預測並且比其純文字長度要長，因此請確保關聯的資料庫列屬性是 TEXT 類型或更大。此外，由於資料庫中的值是加密的，您將無法查詢或搜尋加密的屬性值。

50.3.5.1 金鑰輪換

如你所知，Laravel 使用應用程式的 app 組態檔案中指定的 key 組態值對字串進行加密。通常，該值對應於 APP_KEY 環境變數的值。如果需要輪換應用程式的加密金鑰，則需要使用新金鑰手動重新加密加密屬性。

50.3.6 查詢時轉換

有時你可能需要在執行查詢時應用強制轉換，例如從表中選擇原始值時。 例如，考慮以下查詢：

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

$users = User::select([
    'users.*',
    'last_posted_at' => Post::selectRaw('MAX(created_at)')
            ->whereColumn('user_id', 'users.id')
])->get();
```

在該查詢獲取到的結果集中，last_posted_at 屬性將會是一個字串。假如我們在執行查詢時進行 datetime 類型轉換將更方便。你可以通過使用 withCasts 方法來完成上述操作：

```
$users = User::select([
    'users.*',
    'last_posted_at' => Post::selectRaw('MAX(created_at)')
            ->whereColumn('user_id', 'users.id')
])->withCasts([
    'last_posted_at' => 'datetime'
])->get();
```

50.4 自訂類型轉換

Laravel 有多種內建的、有用的類型轉換； 如果需要自訂強制轉換類型。要建立一個類型轉換，執行 make:cast 命令。 新的強制轉換類將被放置在你的 app/Casts 目錄中:

```
php artisan make:cast Json
```

所有自訂強制轉換類都實現了 CastsAttributes 介面。 實現這個介面的類必須定義一個 get 和 set 方法。get 方法負責將資料庫中的原始值轉換為轉換值，而 set 方法應將轉換值轉換為可以儲存在資料庫中的原始值。 作為示例，我們將內建的 json 類型轉換重新實現為自訂類型：

```
<?php
```

```
namespace App\Casts;

use Illuminate\Contracts\Database\Eloquent\CastsAttributes;
use Illuminate\Database\Eloquent\Model;

class Json implements CastsAttributes
{
    /**
     * 將取出的資料進行轉換。
     *
     * @param  array<string, mixed>  $attributes
     * @return array<string, mixed>
     */
    public function get(Model $model, string $key, mixed $value, array $attributes):
array
    {
        return json_decode($value, true);
    }

    /**
     * 轉換成將要進行儲存的值。
     *
     * @param  array<string, mixed>  $attributes
     */
    public function set(Model $model, string $key, mixed $value, array $attributes):
string
    {
        return json_encode($value);
    }
}
```

定義好自訂類型轉換後，可以使用其類名稱將其附加到模型屬性裡：

```
<?php

namespace App\Models;

use App\Casts\Json;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 應被強制轉換的屬性。
     *
     * @var array
     */
    protected $casts = [
        'options' => Json::class,
    ];
}
```

50.4.1 值對象轉換

你不僅可以將資料轉換成原生的資料類型，還可以將資料轉換成對象。兩種自訂類型轉換的定義方式非常類似。但是將資料轉換成對象的自訂轉換類中的 `set` 方法需要返回鍵值對陣列，用於設定原始、可儲存的值到對應的模型中。

例如，我們將定義一個自訂轉換類，將多個模型值轉換為單個 `Address` 值對象。 我們將假設 `Address` 值有兩個公共屬性：`lineOne` 和 `lineTwo`：

```
<?php

namespace App\Casts;
```

```
use App\ValueObjects\Address as AddressValueObject;
use Illuminate\Contracts\Database\Eloquent\CastsAttributes;
use Illuminate\Database\Eloquent\Model;
use InvalidArgumentException;

class Address implements CastsAttributes
{
    /**
     * 轉換給定的值。
     *
     * @param  array<string, mixed>  $attributes
     */
    public function get(Model $model, string $key, mixed $value, array $attributes):
AddressValueObject
    {
        return new AddressValueObject(
            $attributes['address_line_one'],
            $attributes['address_line_two']
        );
    }

    /**
     * 準備給定值以進行儲存。
     *
     * @param  array<string, mixed>  $attributes
     * @return array<string, string>
     */
    public function set(Model $model, string $key, mixed $value, array $attributes):
array
    {
        if (! $value instanceof AddressValueObject) {
            throw new InvalidArgumentException('The given value is not an Address
instance.');
        }

        return [
            'address_line_one' => $value->lineOne,
            'address_line_two' => $value->lineTwo,
        ];
    }
}
```

轉換為值對象時，對值對象所做的任何更改都將在模型保存之前自動同步回模型：

```
use App\Models\User;

$user = User::find(1);

$user->address->lineOne = 'Updated Address Value';

$user->save();
```

> 注意：如果你計畫將包含值對象的 Eloquent 模型序列化為 JSON 或陣列，那麼應該在值對象上實現 `Illuminate\Contracts\Support\Arrayable` 和 `JsonSerializable` 介面。

50.4.2 陣列 / JSON 序列化

當使用 `toArray` 和 `toJson` 方法將 Eloquent 模型轉換為陣列或 JSON 時，自訂轉換值對象通常會被序列化，只要它們實現 `Illuminate\Contracts\Support\Arrayable` 和 `JsonSerializable` 介面。 但是，在使用第三方庫提供的值對象時，你可能無法將這些介面新增到對象中。

因此，你可以指定你自訂的類型轉換類，它將負責序列化成值對象。為此，你自訂的類型轉換類需要實現 `Illuminate\Contracts\Database\Eloquent\SerializesCastableAttributes` 介面。此介面聲

明類應包含 `serialize` 方法，該方法應返回值對象的序列化形式：

```
/**
 * 獲取值的序列化表示形式。
 *
 * @param  array<string, mixed>  $attributes
 */
public function serialize(Model $model, string $key, mixed $value, array $attributes):
string
{
    return (string) $value;
}
```

50.4.3 入站轉換

有時候，你可能只需要對寫入模型的屬性值進行類型轉換而不需要對從模型中獲取的屬性值進行任何處理。

入站自訂強制轉換應該實現 `CastsInboundAttributes` 介面，該介面只需要定義一個 `set` 方法。`make:cast`Artisan 命令可以通過—`inbound` 選項呼叫來生成一個入站強制轉換類:

```
php artisan make:cast Hash --inbound
```

僅入站強制轉換的一個經典示例是「hashing」強制轉換。例如，我們可以定義一個類型轉換，通過給定的演算法雜湊入站值:

```
<?php

namespace App\Casts;

use Illuminate\Contracts\Database\Eloquent\CastsInboundAttributes;
use Illuminate\Database\Eloquent\Model;

class Hash implements CastsInboundAttributes
{
    /**
     * 建立一個新的強制轉換類實例。
     */
    public function __construct(
        protected string $algorithm = null,
    ) {}

    /**
     * 轉換成將要進行儲存的值
     *
     * @param  array<string, mixed>  $attributes
     */
    public function set(Model $model, string $key, mixed $value, array $attributes):
string
    {
        return is_null($this->algorithm)
                    ? bcrypt($value)
                    : hash($this->algorithm, $value);
    }
}
```

50.4.4 轉換參數

當將自訂類型轉換附加到模型時，可以指定傳入的類型轉換參數。傳入類型轉換參數需使用：將參數與類名分隔，多個參數之間使用逗號分隔。這些參數將會傳遞到類型轉換類的建構函式中：

```
/**
 * 應該強制轉換的屬性。
 *
```

```
 * @var array
 */
protected $casts = [
    'secret' => Hash::class.':sha256',
];
```

50.4.5 可轉換

如果要允許應用程式對象的值定義它們自訂轉換類。除了將自訂轉換類附加到你的模型之外，還可以附加一個實現 `Illuminate\Contracts\Database\Eloquent\Castable` 介面的值對象類：

```
use App\Models\Address;

protected $casts = [
    'address' => Address::class,
];
```

實現 `Castable` 介面的對象必須定義一個 `castUsing` 方法，此方法返回的是負責將 `Castable` 類進行自訂轉換的轉換器類名：

```
<?php

namespace App\Models;

use Illuminate\Contracts\Database\Eloquent\Castable;
use App\Casts\Address as AddressCast;

class Address implements Castable
{
    /**
     * 獲取轉換器的類名用以轉換當前類型轉換對象。
     *
     * @param  array<string, mixed>  $arguments
     */
    public static function castUsing(array $arguments): string
    {
        return AddressCast::class;
    }
}
```

使用 `Castable` 類時，仍然可以在 `$casts` 定義中提供參數。參數將傳遞給 `castUsing` 方法：

```
use App\Models\Address;

protected $casts = [
    'address' => Address::class.':argument',
];
```

50.4.5.1 可轉換 & 匿名類型轉換類

通過將 `castables` 與 PHP 的 匿名類 相結合，可以將值對象及其轉換邏輯定義為單個可轉換對象。為此，請從值對象的 `castUsing` 方法返回一個匿名類。匿名類應該實現 `CastsAttributes` 介面：

```
<?php

namespace App\Models;

use Illuminate\Contracts\Database\Eloquent\Castable;
use Illuminate\Contracts\Database\Eloquent\CastsAttributes;
use Illuminate\Database\Eloquent\Model;

class Address implements Castable
{
    // ...

    /**
```

```
     * 獲取轉換器類用以轉換當前類型轉換對象。
     *
     * @param  array<string, mixed>  $arguments
     */
    public static function castUsing(array $arguments): CastsAttributes
    {
        return new class implements CastsAttributes
        {
            public function get(Model $model, string $key, mixed $value, array
$attributes): Address
            {
                return new Address(
                    $attributes['address_line_one'],
                    $attributes['address_line_two']
                );
            }

            public function set(Model $model, string $key, mixed $value, array
$attributes): array
            {
                return [
                    'address_line_one' => $value->lineOne,
                    'address_line_two' => $value->lineTwo,
                ];
            }
        };
    }
}
```

51 API 資源

51.1 簡介

在建構 API 時，你往往需要一個轉換層來聯結你的 Eloquent 模型和實際返回給使用者的 JSON 響應。比如，你可能希望顯示部分使用者屬性而不是全部，或者你可能希望在模型的 JSON 中包括某些關係。Eloquent 的資源類能夠讓你以更直觀簡便的方式將模型和模型集合轉化成 JSON。

當然，你可以始終使用 Eloquent 模型或集合的 `toJson` 方法將其轉換為 JSON ；但是，Eloquent 的資源提供了對模型及其關係的 JSON 序列化更加精細和更加健壯的控制。

51.2 生成資源

你可以使用 `make:resource` artisan 命令來生成一個資源類。默認情況下，資源將放在應用程式的 `app/Http/Resources` 目錄下。資源繼承自 `Illuminate\Http\Resources\Json\JsonResource` 類：

```
php artisan make:resource UserResource
```

51.2.1.1 資源集合

除了生成轉換單個模型的資源之外，還可以生成負責轉換模型集合的資源。這允許你的 JSON 包含與給定資源的整個集合相關的其他資訊。

你應該在建立資源集合時使用 `--collection` 標誌來表明你要生成一個資源集合。或者，在資源名稱中包含 Collection 一詞將向 Laravel 表明它應該生成一個資源集合。資源集合繼承自 `Illuminate\Http\Resources\Json\ResourceCollection` 類：

```
php artisan make:resource User --collection

php artisan make:resource UserCollection
```

51.3 概念綜述

> **提示**
> 提示：這是對資源和資源集合的高度概述。強烈建議您閱讀本文件的其他部分，以深入瞭解如何更好地自訂和使用資源。

在深入瞭解如何定製化編寫你的資源之前，讓我們首先從高層次上瞭解 Laravel 中如何使用資源。一個資源類表示一個單一模型需要被轉換成 JSON 格式。例如，下面是一個簡單的 `UserResource` 資源類：

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Request;
use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    /**
     * 將資源轉換為陣列。
```

```
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
}
```

每個資源類都定義了一個 `toArray` 方法，當資源從路由或 controller 方法作為響應被呼叫返回時，該方法返回應該轉換為 JSON 的屬性陣列。

注意，我們可以直接使用 `$this` 變數訪問模型屬性。這是因為資源類將自動代理屬性和方法訪問到底層模型以方便訪問。一旦定義了資源，你可以從路由或 controller 中呼叫並返回它。資源通過其建構函式接受底層模型實例：

```
use App\Http\Resources\UserResource;
use App\Models\User;

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

51.3.1 資源集合

如果你要返回一個資源集合或一個分頁響應，你應該在路由或 controller 中建立資源實例時使用你的資源類提供的 `collection` 方法：

```
use App\Http\Resources\UserResource;
use App\Models\User;

Route::get('/users', function () {
    return UserResource::collection(User::all());
});
```

當然了，使用如上方法你將不能新增任何附加的中繼資料和集合一起返回。如果你需要自訂資源集合響應，你需要建立一個專用的資源來表示集合：

```
php artisan make:resource UserCollection
```

此時，你就可以輕鬆地自訂響應中應該包含的任何中繼資料：

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Request;
use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * 將資源集合轉換為陣列。
     *
     * @return array<int|string, mixed>
     */
    public function toArray(Request $request): array
    {
        return [
```

```
            'data' => $this->collection,
            'links' => [
                'self' => 'link-value',
            ],
        ];
    }
}
```

你可以在路由或者 controller 中返回已定義的資源集合：

```
use App\Http\Resources\UserCollection;
use App\Models\User;

Route::get('/users', function () {
    return new UserCollection(User::all());
});
```

51.3.1.1 保護集合的鍵

當從路由返回一個資源集合時，Laravel 會重設集合的鍵，使它們按數字順序排列。但是，你可以在資源類中新增 `preserveKeys` 屬性，以指示是否應該保留集合的原始鍵：

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    /**
     * 指示是否應保留資源的集合原始鍵。
     *
     * @var bool
     */
    public $preserveKeys = true;
}
```

如果 `preserveKeys` 屬性設定為 `true` ，那麼從路由或 controller 返回集合時，集合的鍵將會被保留：

```
use App\Http\Resources\UserResource;
use App\Models\User;

Route::get('/users', function () {
    return UserResource::collection(User::all()->keyBy->id);
});
```

51.3.1.2 自訂基礎資源類

通常，資源集合的 `$this->collection` 屬性會被自動填充，結果是將集合的每個項對應到其單個資源類。單個資源類被假定為資源的類名，但沒有類名末尾的 `Collection` 部分。 此外，根據您的個人偏好，單個資源類可以帶著後綴 `Resource` ，也可以不帶。

例如，`UserCollection` 會嘗試將給定的使用者實例對應到 `User` 或 `UserResource` 資源。想要自訂該行為，你可以重寫資源集合中的 `$collects` 屬性指定自訂的資源：

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * 自訂資源類名。
```

```
     *
     * @var string
     */
    public $collects = Member::class;
}
```

51.4 編寫資源

> **Note**
> 技巧：如果您還沒有閱讀 概念綜述，那麼在繼續閱讀本文件前，強烈建議您去閱讀一下，會更容易理解本節的內容。

從本質上說，資源的作用很簡單，它只需將一個給定的模型轉換為一個陣列。因此，每個資源都包含一個 `toArray` 方法，這個方法會將模型的屬性轉換為一個 API 友好的陣列，然後將該陣列通過路由或 controller 返回給使用者：

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Request;
use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    /**
     * 將資源轉換為陣列。
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
}
```

一旦資源被定義，它可以直接從路由或 controller 返回：

```
use App\Http\Resources\UserResource;
use App\Models\User;

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

51.4.1.1 關聯關係

如果你想在你的響應中包含關聯的資源，你可以將它們新增到你的資源的 `toArray` 方法返回的陣列中。在下面的例子中，我們將使用 `PostResource` 資源的 `collection` 方法來將使用者的部落格文章新增到資源響應中：

```
use App\Http\Resources\PostResource;
use Illuminate\Http\Request;

/**
 * 將資源轉換為陣列。
```

```
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'posts' => PostResource::collection($this->posts),
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
```

注意：如果你只希望在已經載入的關聯關係中包含它們，點這裡查看 條件關聯。

51.4.1.2 資源集合

當資源將單個模型轉換為陣列時，資源集合將模型集合轉換為陣列。當然，你並不是必須要為每個類都定義一個資源集合類，因為所有的資源都提供了一個 `collection` 方法來動態地生成一個「臨時」資源集合：

```
use App\Http\Resources\UserResource;
use App\Models\User;

Route::get('/users', function () {
    return UserResource::collection(User::all());
});
```

當然，如果你需要自訂資源集合返回的中繼資料，那就需要自己建立資源集合類：

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * 將資源集合轉換為陣列。
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array
    {
        return [
            'data' => $this->collection,
            'links' => [
                'self' => 'link-value',
            ],
        ];
    }
}
```

與單一資源一樣，資源集合可以直接從路由或 controller 返回:

```
use App\Http\Resources\UserCollection;
use App\Models\User;

Route::get('/users', function () {
    return new UserCollection(User::all());
});
```

51.4.2 封包裹

默認情況下，當資源響應被轉換為 JSON 時，最外層的資源被包裹在 `data` 鍵中。因此一個典型的資源收集響應如下所示:

```
{
    "data": [
        {
            "id": 1,
            "name": "Eladio Schroeder Sr.",
            "email": "therese28@example.com"
        },
        {
            "id": 2,
            "name": "Liliana Mayert",
            "email": "evandervort@example.com"
        }
    ]
}
```

如果你想使用自訂鍵而不是 `data`，你可以在資源類上定義一個 `$wrap` 屬性:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    /**
     * 應該應用的「資料」包裝器。
     *
     * @var string|null
     */
    public static $wrap = 'user';
}
```

如果你想停用最外層資源的包裹，你應該呼叫基類 `Illuminate\Http\Resources\Json\JsonResource` 的 `withoutWrapping` 方法。通常，你應該從你的 `AppServiceProvider` 或其他在程序每一個請求中都會被載入的 服務提供者 中呼叫這個方法:

```
<?php

namespace App\Providers;

use Illuminate\Http\Resources\Json\JsonResource;
use Illuminate\Support\ServiceProvider;

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
    {
        JsonResource::withoutWrapping();
```

```
    }
}
```

> **注意**
> `withoutWrapping` 方法只會停用最外層資源的包裹，不會刪除你手動新增到資源集合中的 `data` 鍵。

和單個資源一樣，你可以在路由或 controller 中直接返回資源集合：

```
use App\Http\Resources\UserCollection;
use App\Models\User;

Route::get('/users', function () {
    return new UserCollection(User::all());
});
```

51.4.3 封包裹

默認情況下，當資源響應被轉換為 JSON 時，最外層的資源被包裹在 `data` 鍵中。因此一個典型的資源收集響應如下所示：

```
{
    "data": [
        {
            "id": 1,
            "name": "Eladio Schroeder Sr.",
            "email": "therese28@example.com"
        },
        {
            "id": 2,
            "name": "Liliana Mayert",
            "email": "evandervort@example.com"
        }
    ]
}
```

如果你想使用自訂鍵而不是 `data`，你可以在資源類上定義一個 `$wrap` 屬性：

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    /**
     * 應該應用的「資料」包裝器。
     *
     * @var string|null
     */
    public static $wrap = 'user';
}
```

如果你想停用最外層資源的包裹，你應該呼叫基類 `Illuminate\Http\Resources\Json\JsonResource` 的 `withoutWrapping` 方法。通常，你應該從你的 `AppServiceProvider` 或其他在程序每一個請求中都會被載入的 服務提供者 中呼叫這個方法：

```
<?php

namespace App\Providers;

use Illuminate\Http\Resources\Json\JsonResource;
use Illuminate\Support\ServiceProvider;
```

```
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
    {
        JsonResource::withoutWrapping();
    }
}
```

> **注意** `withoutWrapping` 方法只會停用最外層資源的包裹，不會刪除你手動新增到資源集合中的 `data` 鍵。

51.4.3.1 包裹巢狀資源

你可以完全自由地決定資源關聯如何被包裹。如果你希望無論怎樣巢狀，所有的資源集合都包裹在一個 `data` 鍵中，你應該為每個資源定義一個資源集合類，並將返回的集合包裹在 `data` 鍵中。

你可能會擔心這是否會導致最外層的資源包裹在兩層 `data` 鍵中。別擔心， Laravel 永遠不會讓你的資源被雙層包裹，所以你不必擔心資源集合被多重巢狀的問題：

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class CommentsCollection extends ResourceCollection
{
    /**
     * 將資源集合轉換成陣列。
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array
    {
        return ['data' => $this->collection];
    }
}
```

51.4.3.2 封包裹和分頁

當通過資源響應返回分頁集合時，即使你呼叫了 `withoutWrapping` 方法，Laravel 也會將你的資源封包裹在 `data` 鍵中。這是因為分頁響應總會有 `meta` 和 `links` 鍵包含關於分頁狀態的資訊：

```
{
    "data": [
        {
            "id": 1,
            "name": "Eladio Schroeder Sr.",
            "email": "therese28@example.com"
        },
        {
```

```
            "id": 2,
            "name": "Liliana Mayert",
            "email": "evandervort@example.com"
        }
    ],
    "links":{
        "first": "http://example.com/pagination?page=1",
        "last": "http://example.com/pagination?page=1",
        "prev": null,
        "next": null
    },
    "meta":{
        "current_page": 1,
        "from": 1,
        "last_page": 1,
        "path": "http://example.com/pagination",
        "per_page": 15,
        "to": 10,
        "total": 10
    }
}
```

51.4.4 分頁

你可以將 Laravel 分頁實例傳遞給資源的 `collection` 方法或自訂資源集合：

```
use App\Http\Resources\UserCollection;
use App\Models\User;

Route::get('/users', function () {
    return new UserCollection(User::paginate());
});
```

分頁響應中總有 `meta` 和 `links` 鍵包含著分頁狀態資訊：

```
{
    "data": [
        {
            "id": 1,
            "name": "Eladio Schroeder Sr.",
            "email": "therese28@example.com"
        },
        {
            "id": 2,
            "name": "Liliana Mayert",
            "email": "evandervort@example.com"
        }
    ],
    "links":{
        "first": "http://example.com/pagination?page=1",
        "last": "http://example.com/pagination?page=1",
        "prev": null,
        "next": null
    },
    "meta":{
        "current_page": 1,
        "from": 1,
        "last_page": 1,
        "path": "http://example.com/pagination",
        "per_page": 15,
        "to": 10,
        "total": 10
    }
}
```

51.4.5 條件屬性

有些時候，你可能希望在給定條件滿足時新增屬性到資源響應裡。例如，你可能希望如果當前使用者是「管理員」時新增某個值到資源響應中。在這種情況下 Laravel 提供了一些輔助方法來幫助你解決問題。`when` 方法可以被用來有條件地向資源響應新增屬性：

```
/**
 * 將資源轉換成陣列。
 *
 * @return array<string, mixed>
 */
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'secret' => $this->when($request->user()->isAdmin(), 'secret-value'),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

在上面這個例子中，只有當 `isAdmin` 方法返回 `true` 時，`secret` 鍵才會最終在資源響應中被返回。如果該方法返回 `false` 鍵將會在資源響應被傳送給客戶端之前被刪除。 `when` 方法可以使你避免使用條件語句拼接陣列，轉而用更優雅的方式來編寫你的資源。

`when` 方法也接受閉包作為其第二個參數，只有在給定條件為 `true` 時，才從閉包中計算返回的值：

```
'secret' => $this->when($request->user()->isAdmin(), function () {
    return 'secret-value';
}),
```

`whenHas` 方法可以用來包含一個屬性，如果它確實存在於底層模型中：

```
'name' => $this->whenHas('name'),
```

此外， `whenNotNull` 方法可用於在資源響應中包含一個屬性，如果該屬性不為空：

```
'name' => $this->whenNotNull($this->name),
```

51.4.5.1 有條件地合併資料

有些時候，你可能希望在給定條件滿足時新增多個屬性到資源響應裡。在這種情況下，你可以使用 `mergeWhen` 方法在給定的條件為 `true` 時將多個屬性新增到響應中：

```
/**
 * 將資源轉換成陣列
 *
 * @return array<string, mixed>
 */
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        $this->mergeWhen($request->user()->isAdmin(), [
            'first-secret' => 'value',
            'second-secret' => 'value',
        ]),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
```

```
}
```

同理，如果給定的條件為 `false` 時，則這些屬性將會在資源響應被傳送給客戶端之前被移除。

> **注意**
> `mergeWhen` 方法不應該被使用在混合字串和數字鍵的陣列中。此外，它也不應該被使用在不按順序排列的數字鍵的陣列中。

51.4.6 條件關聯

除了有條件地載入屬性之外，你還可以根據模型關聯是否已載入來有條件地在你的資源響應中包含關聯。這允許你在 controller 中決定載入哪些模型關聯，這樣你的資源可以在模型關聯被載入後才新增它們。最終，這樣做可以使你的資源輕鬆避免「N+1」查詢問題。

如果屬性確實存在於模型中，可以用 `whenHas` 來獲取：

```
'name' => $this->whenHas('name'),
```

此外，`whenNotNull` 可用於在資源響應中獲取一個不為空的屬性：

```
'name' => $this->whenNotNull($this->name),
```

51.4.6.1 有條件地合併資料

有些時候，你可能希望在給定條件滿足時新增多個屬性到資源響應裡。在這種情況下，你可以使用 `mergeWhen` 方法在給定的條件為 `true` 時將多個屬性新增到響應中：

```
/**
 * 將資源轉換成陣列。
 *
 * @return array<string, mixed>
 */
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        $this->mergeWhen($request->user()->isAdmin(), [
            'first-secret' => 'value',
            'second-secret' => 'value',
        ]),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

同理，如果給定的條件為 `false` 時，則這些屬性將會在資源響應被傳送給客戶端之前被移除。

> **注意** `mergeWhen` 方法不應該被使用在混合字串和數字鍵的陣列中。此外，它也不應該被使用在不按順序排列的數字鍵的陣列中。

51.4.7 條件關聯

除了有條件地載入屬性之外，你還可以根據模型關聯是否已載入來有條件地在你的資源響應中包含關聯。這允許你在 controller 中決定載入哪些模型關聯，這樣你的資源可以在模型關聯被載入後才新增它們。最終，這樣做可以使你的資源輕鬆避免「N+1」查詢問題。

可以使用 `whenLoaded` 方法來有條件的載入關聯。為了避免載入不必要的關聯，此方法接受關聯的名稱而不

是關聯本身作為其參數：

```
use App\Http\Resources\PostResource;

/**
 * 將資源轉換成陣列。
 *
 * @return array<string, mixed>
 */
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'posts' => PostResource::collection($this->whenLoaded('posts')),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

在上面這個例子中，如果關聯沒有被載入，則 `posts` 鍵將會在資源響應被傳送給客戶端之前被刪除。

51.4.7.1 條件關係計數

除了有條件地包含關係之外，你還可以根據關係的計數是否已載入到模型上，有條件地包含資源響應中的關係「計數」：

```
new UserResource($user->loadCount('posts'));
```

`whenCounted` 方法可用於在資源響應中有條件地包含關係的計數。該方法避免在關係計數不存在時不必要地包含屬性:

```
/**
 * 將資源轉換為一個陣列。
 *
 * @return array<string, mixed>
 */
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'posts_count' => $this->whenCounted('posts'),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

在這個例子中，如果 `posts` 關係的計數還沒有載入，`posts_count` 鍵將在資源響應傳送到客戶端之前從資源響應中刪除。

51.4.7.2 條件中間表資訊

除了在你的資源響應中有條件地包含關聯外，你還可以使用 `whenPivotLoaded` 方法有條件地從多對多關聯的中間表中新增資料。`whenPivotLoaded` 方法接受的第一個參數為中間表的名稱。第二個參數是一個閉包，它定義了在模型上如果中間表資訊可用時要返回的值：

```
/**
 * 將資源轉換成陣列。
 *
 * @return array<string, mixed>
```

```
     */
    public function toArray(Request $request): array
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'expires_at' => $this->whenPivotLoaded('role_user', function () {
                return $this->pivot->expires_at;
            }),
        ];
    }
```

如果你的關聯使用的是 自訂中間表，你可以將中間表模型的實例作為 whenPivotLoaded 方法的第一個參數:

```
'expires_at' => $this->whenPivotLoaded(new Membership, function () {
    return $this->pivot->expires_at;
}),
```

如果你的中間表使用的是 pivot 以外的訪問器，你可以使用 whenPivotLoadedAs 方法：

```
/**
 * 將資源轉換成陣列。
 *
 * @return array<string, mixed>
 */
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'expires_at' => $this->whenPivotLoadedAs('subscription', 'role_user', function
() {
            return $this->subscription->expires_at;
        }),
    ];
}
```

51.4.8 新增中繼資料

一些 JSON API 標準需要你在資源和資源集合響應中新增中繼資料。這通常包括資源或相關資源的 links ，或一些關於資源本身的中繼資料。如果你需要返回有關資源的其他中繼資料，只需要將它們包含在 toArray 方法中即可。例如在轉換資源集合時你可能需要新增 link 資訊：

```
/**
 * 將資源轉換成陣列。
 *
 * @return array<string, mixed>
 */
public function toArray(Request $request): array
{
    return [
        'data' => $this->collection,
        'links' => [
            'self' => 'link-value',
        ],
    ];
}
```

當新增額外的中繼資料到你的資源中時，你不必擔心會覆蓋 Laravel 在返回分頁響應時自動新增的 links 或 meta 鍵。你新增的任何其他 links 會與分頁響應新增的 links 相合併。

51.4.8.1 頂層中繼資料

有時候，你可能希望當資源被作為頂層資源返回時新增某些中繼資料到資源響應中。這通常包括整個響應的元

資訊。你可以在資源類中新增 `with` 方法來定義中繼資料。此方法應返回一個中繼資料陣列，當資源被作為頂層資源渲染時，這個陣列將會被包含在資源響應中：

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * 將資源集合轉換成陣列。
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array
    {
        return parent::toArray($request);
    }

    /**
     * 返回應該和資源一起返回的其他資料陣列。
     *
     * @return array<string, mixed>
     */
    public function with(Request $request): array
    {
        return [
            'meta' => [
                'key' => 'value',
            ],
        ];
    }
}
```

51.4.8.2 構造資源時新增中繼資料

你還可以在路由或者 controller 中構造資源實例時新增頂層資料。所有資源都可以使用 `additional` 方法來接受應該被新增到資源響應中的資料陣列：

```
return (new UserCollection(User::all()->load('roles')))
                ->additional(['meta' => [
                    'key' => 'value',
                ]]);
```

51.5 響應資源

就像你知道的那樣，資源可以直接在路由和 controller 中被返回：

```
use App\Http\Resources\UserResource;
use App\Models\User;

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

但有些時候，在傳送給客戶端前你可能需要自訂 HTTP 響應。你有兩種辦法。第一，你可以鏈式呼叫 `response` 方法。此方法將會返回 `Illuminate\Http\JsonResponse` 實例，允許你自訂響應頭資訊：

```
use App\Http\Resources\UserResource;
use App\Models\User;

Route::get('/user', function () {
```

```
    return (new UserResource(User::find(1)))
                ->response()
                ->header('X-Value', 'True');
});
```

另外，你還可以在資源中定義一個 `withResponse` 方法。此方法將會在資源被作為頂層資源在響應時被呼叫：

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\JsonResponse;
use Illuminate\Http\Request;
use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource
{
    /**
     * 將資源轉換為陣列。
     *
     * @return array<string, mixed>
     */
    public function toArray(Request $request): array
    {
        return [
            'id' => $this->id,
        ];
    }

    /**
     * 自訂響應資訊。
     */
    public function withResponse(Request $request, JsonResponse $response): void
    {
        $response->header('X-Value', 'True');
    }
}
```

52 序列化

52.1 簡介

在使用 Laravel 建構 API 時，經常需要把模型和關聯轉化為陣列或 JSON。針對這些操作，Eloquent 提供了一些便捷方法，以及對序列化中的屬性控制。

> 技巧：想獲得更全面處理 Eloquent 的模型和集合 JSON 序列化的方法，請查看 Eloquent API 資源 文件。

52.2 序列化模型 & 集合

52.2.1 序列化為陣列

要轉化模型及其載入的 關聯 為陣列，可以使用 `toArray` 方法。這是一個遞迴的方法，因此所有的屬性和關聯（包括關聯的關聯）都將轉化成陣列：

```
use App\Models\User;

$user = User::with('roles')->first();

return $user->toArray();
```

`attributesToArray` 方法可用於將模型的屬性轉換為陣列，但不會轉換其關聯：

```
$user = User::first();

return $user->attributesToArray();
```

您還可以通過呼叫集合實例上的 `toArray` 方法，將模型的全部 集合 轉換為陣列：

```
$users = User::all();

return $users->toArray();
```

52.2.2 序列化為 JSON

您可以使用 `toJson` 方法將模型轉化成 JSON。和 `toArray` 一樣，`toJson` 方法也是遞迴的，因此所有屬性和關聯都會轉化成 JSON, 您還可以指定由 PHP 支援的任何 JSON 編碼選項：

```
use App\Models\User;

$user = User::find(1);

return $user->toJson();

return $user->toJson(JSON_PRETTY_PRINT);
```

或者，你也可以將模型或集合轉換為字串，模型或集合上的 `toJson` 方法會自動呼叫：

```
return (string) User::find(1);
```

由於模型和集合在轉化為字串的時候會轉成 JSON， 因此可以在應用的路由或 controller 中直接返回 Eloquent 對象。Laravel 會自動將 Eloquent 模型和集合序列化為 JSON：

```
Route::get('users', function () {
    return User::all();
});
```

52.2.2.1 關聯關係

當一個模型被轉化為 JSON 的時候，它載入的關聯關係也將自動轉化為 JSON 對象被包含進來。同時，通過「小駝峰」定義的關聯方法，關聯的 JSON 屬性將會是「蛇形」命名。

52.3 隱藏 JSON 屬性

有時要將模型陣列或 JSON 中的某些屬性進行隱藏，比如密碼。則可以在模型中新增 `$hidden` 屬性。模型序列化後， `$hidden` 陣列中列出的屬性將不會被顯示：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 陣列中的屬性會被隱藏。
     *
     * @var array
     */
    protected $hidden = ['password'];
}
```

注意 隱藏關聯時，需新增關聯的方法名到 `$hidden` 屬性中。

此外，也可以使用屬性 `visible` 定義一個模型陣列和 JSON 可見的「白名單」。轉化後的陣列或 JSON 不會出現其他的屬性：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 陣列中的屬性會被展示。
     *
     * @var array
     */
    protected $visible = ['first_name', 'last_name'];
}
```

52.3.1.1 臨時修改可見屬性

如果你想要在一個模型實例中顯示隱藏的屬性，可以使用 `makeVisible` 方法。`makeVisible` 方法返回模型實例：

```
return $user->makeVisible('attribute')->toArray();
```

相應地，如果你想要在一個模型實例中隱藏可見的屬性，可以使用 `makeHidden` 方法。

```
return $user->makeHidden('attribute')->toArray();
```

如果你想臨時覆蓋所有可見或隱藏的屬性，你可以分別使用 `setVisible` 和 `setHidden` 方法:

```
return $user->setVisible(['id', 'name'])->toArray();

return $user->setHidden(['email', 'password', 'remember_token'])->toArray();
```

52.4 追加 JSON 值

有時，需要在模型轉換為陣列或 JSON 時新增一些資料庫中不存在欄位的對應屬性。要實現這個功能，首先要定義一個 訪問器：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Casts\Attribute;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 判斷使用者是否是管理員。
     */
    protected function isAdmin(): Attribute
    {
        return new Attribute(
            get: fn () => 'yes',
        );
    }
}
```

如果你想附加屬性到模型中，可以使用模型屬性 appends 中新增該屬性名。注意，儘管訪問器使用「駝峰命名法」方式定義，但是屬性名通常以「蛇形命名法」的方式來引用：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 要附加到模型陣列表單的訪問器。
     *
     * @var array
     */
    protected $appends = ['is_admin'];
}
```

使用 appends 方法追加屬性後，它將包含在模型的陣列和 JSON 中。appends 陣列中的屬性也將遵循模型上組態的 visible 和 hidden 設定。

52.4.1.1 執行階段追加

在執行階段，你可以在單個模型實例上使用 append 方法來追加屬性。或者，使用 setAppends 方法來重寫整個追加屬性的陣列：

```
return $user->append('is_admin')->toArray();

return $user->setAppends(['is_admin'])->toArray();
```

52.5 日期序列化

52.5.1.1 自訂默認日期格式

你可以通過重寫 serializeDate 方法來自訂默認序列化格式。此方法不會影響日期在資料庫中儲存的格

式：

```
/**
 * 為 array / JSON 序列化準備日期格式
 */
protected function serializeDate(DateTimeInterface $date): string
{
    return $date->format('Y-m-d');
}
```

52.5.1.2 自訂默認日期格式

你可以在 Eloquent 的 屬性轉換 中單獨為日期屬性自訂日期格式：

```
protected $casts = [
    'birthday' => 'date:Y-m-d',
    'joined_at' => 'datetime:Y-m-d H:00',
];
```

53 資料工廠

53.1 介紹

當測試你的應用程式或向資料庫填充資料時，你可能需要插入一些記錄到資料庫中。Laravel 允許你使用模型工廠為每個 Eloquent 模型定義一組默認屬性，而不是手動指定每個列的值。

要查看如何編寫工廠的示例，請查看你的應用程式中的 `database/factories/UserFactory.php` 檔案。這個工廠已經包含在所有新的 Laravel 應用程式中，并包含以下工廠定義：

```
namespace Database\Factories;

use Illuminate\Support\Str;
use Illuminate\Database\Eloquent\Factories\Factory;

class UserFactory extends Factory
{
    /**
     * 定義模型的默認狀態
     *
     * @return array<string, mixed>
     */
    public function definition(): array
    {
        return [
            'name' => fake()->name(),
            'email' => fake()->unique()->safeEmail(),
            'email_verified_at' => now(),
            'password' =>
'$2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi', // password
            'remember_token' => Str::random(10),
        ];
    }
}
```

正如你所見，在其最基本的形式下，資料工廠是擴展 Laravel 基礎工廠類並定義一個 `definition` 方法的類。`definition` 方法返回在使用工廠建立模型時應用的默認屬性值集合。

通過 `fake` 輔助器，工廠可以訪問 Faker PHP 庫，它允許你方便地生成各種用於測試和填充的隨機資料。

> **注意** 你可以通過在 `config/app.php` 組態檔案中新增 `faker_locale` 選項來設定你應用程式的 Faker 區域設定。

53.2 定義模型工廠

53.2.1 建立工廠

要建立工廠，請執行 `make:factory` Artisan 命令:

```
php artisan make:factory PostFactory
```

新工廠類將被放置在你的 `database/factories` 目錄中。

53.2.1.1 模型和工廠的自動發現機制

一旦你定義了工廠，你可以使用 `Illuminate\Database\Eloquent\Factories\HasFactory` 特徵提供給模型的靜態 `factory` 方法來為該模型實例化工廠。

`HasFactory` 特徵的 `factory` 方法將使用約定來確定適用於特定模型的工廠。具體而言，該方法將在 `Database\Factories` 命名空間中尋找一個工廠，該工廠的類名與模型名稱匹配，並以 `Factory` 為後綴。如果這些約定不適用於你的特定應用程式或工廠，則可以在模型上覆蓋 `newFactory` 方法，以直接返回模型對應的工廠的實例：

```
use Illuminate\Database\Eloquent\Factories\Factory;
use Database\Factories\Administration\FlightFactory;

/**
 * 為模型建立一個新的工廠實例。
 */
protected static function newFactory(): Factory
{
    return FlightFactory::new();
}
```

接下來，定義相應工廠的 `model` 屬性：

```
use App\Administration\Flight;
use Illuminate\Database\Eloquent\Factories\Factory;

class FlightFactory extends Factory
{
    /**
     * 工廠對應的模型名稱。
     *
     * @var string
     */
    protected $model = Flight::class;
}
```

53.2.2 工廠狀態

狀態操作方法可以讓你定義離散的修改，這些修改可以在任意組合中應用於你的模型工廠。例如，你的 `Database\Factories\UserFactory` 工廠可能包含一個 `suspended` 狀態方法，該方法可以修改其默認屬性值之一。

狀態轉換方法通常會呼叫 Laravel 基礎工廠類提供的 `state` 方法。`state` 方法接受一個閉包函數，該函數將接收為工廠定義的原始屬性陣列，並應返回一個要修改的屬性陣列：

```
use Illuminate\Database\Eloquent\Factories\Factory;

/**
 * 表示使用者已被暫停。
 */
public function suspended(): Factory
{
    return $this->state(function (array $attributes) {
        return [
            'account_status' => 'suspended',
        ];
    });
}
```

53.2.2.1 「Trashed」狀態

如果你的 Eloquent 模型可以進行軟刪除，你可以呼叫內建的 `trashed` 狀態方法來表示建立的模型應該已經被

「軟刪除」。你不需要手動定義 `trashed` 狀態，因為它對所有工廠都是自動可用的：

```
use App\Models\User;

$user = User::factory()->trashed()->create();
```

53.2.3 工廠回呼函數

工廠回呼函數是使用 `afterMaking` 和 `afterCreating` 方法註冊的，它們允許你在建立或製造模型後執行其他任務。你應該通過在工廠類中定義一個 `configure` 方法來註冊這些回呼函數。當工廠被實例化時，Laravel 將自動呼叫這個方法：

```
namespace Database\Factories;

use App\Models\User;
use Illuminate\Database\Eloquent\Factories\Factory;
use Illuminate\Support\Str;

class UserFactory extends Factory
{
    /**
     * 組態模型工廠。
     */
    public function configure(): static
    {
        return $this->afterMaking(function (User $user) {
            // ...
        })->afterCreating(function (User $user) {
            // ...
        });
    }

    // ...
}
```

53.3 使用工廠建立模型

53.3.1 實例化模型

一旦你定義了工廠，你可以使用由 `Illuminate\Database\Eloquent\Factories\HasFactory` 特徵為你的模型提供的靜態 `factory` 方法來實例化該模型的工廠對象。讓我們看一些建立模型的示例。首先，我們將使用 `make` 方法建立模型，而不將其持久化到資料庫中：

```
use App\Models\User;

$user = User::factory()->make();
```

你可以使用 count 方法建立多個模型的集合：

```
$users = User::factory()->count(3)->make();
```

53.3.1.1 應用狀態

你也可以將任何狀態應用於這些模型。如果你想要對這些模型應用多個狀態轉換，只需直接呼叫狀態轉換方法即可：

```
$users = User::factory()->count(5)->suspended()->make();
```

53.3.1.2 覆蓋屬性

如果你想要覆蓋模型的一些預設值，可以將一個值陣列傳遞給 `make` 方法。只有指定的屬性將被替換，而其餘的屬性將保持設定為工廠指定的預設值：

```
$user = User::factory()->make([
    'name' => 'Abigail Otwell',
]);
```

或者，可以直接在工廠實例上呼叫 state 方法進行內聯狀態轉換：

```
$user = User::factory()->state([
    'name' => 'Abigail Otwell',
])->make();
```

> 注意：使用工廠建立模型時，批次賦值保護會自動被停用。

53.3.2 持久化模型

`create` 方法會實例化模型並使用 Eloquent 的 `save` 方法將它們持久化到資料庫中：

```
use App\Models\User;

// 建立單個 App\Models\User 實例。。。
$user = User::factory()->create();

// 建立三個 App\Models\User 實例。。。
$users = User::factory()->count(3)->create();
```

你可以通過將屬性陣列傳遞給 `create` 方法來覆蓋工廠的默認模型屬性：

```
$user = User::factory()->create([
    'name' => 'Abigail',
]);
```

53.3.3 序列

有時，你可能希望為每個建立的模型交替更改給定模型屬性的值。你可以通過將狀態轉換定義為序列來實現此目的。例如，你可能希望為每個建立的使用者在 `admin` 列中在 `Y` 和 `N` 之間交替更改：

```
use App\Models\User;
use Illuminate\Database\Eloquent\Factories\Sequence;

$users = User::factory()
                ->count(10)
                ->state(new Sequence(
                    ['admin' => 'Y'],
                    ['admin' => 'N'],
                ))
                ->create();
```

在這個例子中，將建立五個 `admin` 值為 `Y` 的使用者和五個 `admin` 值為 `N` 的使用者。

如果需要，你可以將閉包作為序列值包含在內。每次序列需要一個新值時，都會呼叫閉包：

```
use Illuminate\Database\Eloquent\Factories\Sequence;

$users = User::factory()
                ->count(10)
                ->state(new Sequence(
                    fn (Sequence $sequence) => ['role' => UserRoles::all()->random()],
                ))
                ->create();
```

在序列閉包內，你可以訪問注入到閉包中的序列實例上的 $index 或 $count 屬性。 $index 屬性包含到目前為止已經進行的序列迭代次數，而 $count 屬性包含序列將被呼叫的總次數：

```
$users = User::factory()
                  ->count(10)
                  ->sequence(fn (Sequence $sequence) => ['name' => 'Name '.$sequence-
>index])
                  ->create();
```

為了方便，序列也可以使用 sequence 方法應用，該方法只是在內部呼叫了 state 方法。 sequence 方法接受一個閉包或序列化屬性的陣列：

```
$users = User::factory()
                  ->count(2)
                  ->sequence(
                      ['name' => 'First User'],
                      ['name' => 'Second User'],
                  )
                  ->create();
```

53.4 工廠關聯

53.4.1 一對多關係

接下來，讓我們探討如何使用 Laravel 流暢的工廠方法建構 Eloquent 模型關係。首先，假設我們的應用程式有一個 App\Models\User 模型和一個 App\Models\Post 模型。同時，假設 User 模型定義了與 Post 的一對多關係。我們可以使用 Laravel 工廠提供的 has 方法建立一個有三篇文章的使用者。has 方法接受一個工廠實例：

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

$user = User::factory()
              ->has(Post::factory()->count(3))
              ->create();
```

按照約定，當將 Post 模型傳遞給 has 方法時，Laravel 將假定 User 模型必須有一個 posts 方法來定義關係。如果需要，你可以顯式指定你想要操作的關係名稱：

```
$user = User::factory()
              ->has(Post::factory()->count(3), 'posts')
              ->create();
```

當然，你可以對相關模型執行狀態操作。此外，如果你的狀態更改需要訪問父模型，你可以傳遞一個基於閉包的狀態轉換：

```
$user = User::factory()
              ->has(
                  Post::factory()
                          ->count(3)
                          ->state(function (array $attributes, User $user) {
                              return ['user_type' => $user->type];
                          })
              )
              ->create();
```

53.4.1.1 使用魔術方法

為了方便起見，你可以使用 Laravel 的魔術工廠關係方法來建構關係。例如，以下示例將使用約定來確定應該通過 User 模型上的 posts 關係方法建立相關模型：

```
$user = User::factory()
             ->hasPosts(3)
             ->create();
```

當使用魔術方法建立工廠關係時，你可以傳遞一個屬性陣列來覆蓋相關模型的屬性：

```
$user = User::factory()
             ->hasPosts(3, [
                 'published' => false,
             ])
             ->create();
```

如果你的狀態更改需要訪問父模型，你可以提供一個基於閉包的狀態轉換：

```
$user = User::factory()
             ->hasPosts(3, function (array $attributes, User $user) {
                 return ['user_type' => $user->type];
             })
             ->create();
```

53.4.2 反向關係

現在我們已經探討了如何使用工廠建構「一對多」關係，讓我們來探討關係的反向操作。`for` 方法可用於定義工廠建立的模型所屬的父模型。例如，我們可以建立三個 `App\Models\Post` 模型實例，這些實例屬於同一個使用者：

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

$posts = Post::factory()
             ->count(3)
             ->for(User::factory()->state([
                 'name' => 'Jessica Archer',
             ]))
             ->create();
```

如果你已經有一個應該與正在建立的模型關聯的父模型實例，則可以將該模型實例傳遞給 `for` 方法：

```
$user = User::factory()->create();

$posts = Post::factory()
             ->count(3)
             ->for($user)
             ->create();
```

53.4.2.1 使用魔術方法

為了方便起見，你可以使用 Laravel 的魔術工廠關係方法來定義「屬於」關係。例如，以下示例將使用慣例來確定這三篇文章應該屬於 `Post` 模型上的 `user` 關係：

```
$posts = Post::factory()
             ->count(3)
             ->forUser([
                 'name' => 'Jessica Archer',
             ])
             ->create();
```

53.4.3 多對多關係

與一對多關係一樣，可以使用 `has` 方法建立「多對多」關係：

```
use App\Models\Role;
use App\Models\User;
```

```
$user = User::factory()
             ->has(Role::factory()->count(3))
             ->create();
```

53.4.3.1 中間表屬性

如果需要定義應該在連結模型的透視表/中間表上設定的屬性，可以使用 `hasAttached` 方法。此方法接受透視表屬性名稱和值的陣列作為其第二個參數：

```
use App\Models\Role;
use App\Models\User;

$user = User::factory()
             ->hasAttached(
                 Role::factory()->count(3),
                 ['active' => true]
             )
             ->create();
```

如果你的狀態更改需要訪問相關模型，則可以提供基於閉包的狀態轉換：

```
$user = User::factory()
             ->hasAttached(
                 Role::factory()
                     ->count(3)
                     ->state(function (array $attributes, User $user) {
                         return ['name' => $user->name.' Role'];
                     }),
                 ['active' => true]
             )
             ->create();
```

如果你已經有要附加到正在建立的模型的模型實例，則可以將這些模型實例傳遞給 `hasAttached` 方法。在此示例中，相同的三個角色將附加到所有三個使用者：

```
$roles = Role::factory()->count(3)->create();

$user = User::factory()
             ->count(3)
             ->hasAttached($roles, ['active' => true])
             ->create();
```

53.4.3.2 使用魔術方法

為了方便，你可以使用 Laravel 的魔術工廠關係方法來定義多對多關係。例如，以下示例將使用約定確定應通過 `User` 模型上的 `roles` 關係方法建立相關模型：

```
$user = User::factory()
             ->hasRoles(1, [
                 'name' => 'Editor'
             ])
             ->create();
```

53.4.4 多型關聯

多型關聯也可以使用工廠函數建立。多型「morph many」關聯的建立方式與典型的「has many」關聯相同。例如，如果 `App\Models\Post` 模型與 `App\Models\Comment` 模型具有多型的 `morphMany` 關係：

```
use App\Models\Post;

$post = Post::factory()->hasComments(3)->create();
```

53.4.4.1 Morph To 關聯

不能使用魔術方法建立 morphTo 關聯。必須直接使用 for 方法，並明確提供關聯名稱。例如，假設 Comment 模型有一個 commentable 方法，該方法定義了一個 morphTo 關聯。在這種情況下，我們可以使用 for 方法直接建立屬於單個帖子的三個評論：

```
$comments = Comment::factory()->count(3)->for(
    Post::factory(), 'commentable'
)->create();
```

53.4.4.2 多型多對多關聯

多型「many to many」(morphToMany / morphedByMany) 關聯的建立方式與非多型 「many to many」 關聯相同：

```
use App\Models\Tag;
use App\Models\Video;

$videos = Video::factory()
             ->hasAttached(
                 Tag::factory()->count(3),
                 ['public' => true]
             )
             ->create();
```

當然，魔術方法 has 也可以用於建立多型「many to many」關係：

```
$videos = Video::factory()
             ->hasTags(3, ['public' => true])
             ->create();
```

53.4.5 在工廠中定義關係

在模型工廠中定義關係時，通常會將一個新的工廠實例分配給關係的外部索引鍵。這通常是針對「反向」關係，例如 belongsTo 和 morphTo 關係。例如，如果你想在建立帖子時建立一個新使用者，則可以執行以下操作：

```
use App\Models\User;

/**
 * 定義模型的默認狀態.
 *
 * @return array<string, mixed>
 */
public function definition(): array
{
    return [
        'user_id' => User::factory(),
        'title' => fake()->title(),
        'content' => fake()->paragraph(),
    ];
}
```

如果關係的列依賴於定義它的工廠，你可以將閉包分配給屬性。該閉包將接收工廠計算出的屬性陣列

```
/**
 * 定義模型的默認狀態.
 *
 * @return array<string, mixed>
 */
public function definition(): array
{
    return [
        'user_id' => User::factory(),
```

```
            'user_type' => function (array $attributes) {
                return User::find($attributes['user_id'])->type;
            },
            'title' => fake()->title(),
            'content' => fake()->paragraph(),
        ];
}
```

53.4.6 在關係中重複使用現有模型

如果你有多個模型與另一個模型共享一個公共關係，則可以使用 `recycle` 方法來確保相關模型的單個實例在工廠建立的所有關係中被重複使用。

例如，假設你有 `Airline`、`Flight` 和 `Ticket` 模型，其中機票屬於一個航空公司和一個航班，而航班也屬於一個航空公司。在建立機票時，你可能希望將同一航空公司用於機票和航班，因此可以將一個航空公司實例傳遞給 `recycle` 方法：

```
Ticket::factory()
    ->recycle(Airline::factory()->create())
    ->create();
```

如果你的模型屬於一個公共使用者或團隊，則可以發現 `recycle` 方法特別有用。

`recycle` 方法還接受一組現有模型。當一組集合提供給 `recycle` 方法時，當工廠需要該類型的模型時，將從集合中選擇一個隨機模型：

```
Ticket::factory()
    ->recycle($airlines)
    ->create();
```

54 測試入門

54.1 介紹

Laravel 在建構時考慮到了測試。實際上，對 PHPUnit 測試的支援是開箱即用的，並且已經為你的應用程式設定了一個 phpunit.xml 檔案。 Laravel 還附帶了方便的幫助方法，允許你對應用程式進行富有表現力的測試。

默認情況下，你應用程式的 tests 目錄下包含兩個子目錄：Feature 和 Unit。**單元測試**（Unit）是針對你的程式碼中非常少，而且相對獨立的一部分程式碼來進行的測試。實際上，大部分單元測試都是針對單個方法進行的。在你的 Unit 測試目錄中進行測試，不會啟動你的 Laravel 應用程式，因此無法訪問你的應用程式的資料庫或其他框架服務。

功能測試（Feature）能測試你的大部分程式碼，包括多個對象如何相互互動，甚至是對 JSON 端點的完整 HTTP 請求。 **通常，你的大多數測試應該是功能測試。這些類型的測試可以最大程度地確保你的系統作為一個整體按預期運行**。

Feature 和 Unit 測試目錄中都提供了一個 ExampleTest.php 檔案。 安裝新的 Laravel 應用程式後，執行 vendor/bin/phpunit 或 php artisan test 命令來運行你的測試。

54.2 環境

運行測試時，由於 phpunit.xml 檔案中定義了 環境變數 ，Laravel 會自動組態環境變數為 testing。Laravel 還會在測試時自動將 session 和快取組態到 array 驅動程式，這意味著在測試時不會持久化 session 或快取資料。

你可以根據需要自由定義其他測試環境組態值。 testing 環境變數可以在應用程式的 phpunit.xml 檔案中組態，但請確保在運行測試之前使用 config:clear Artisan 命令清除組態快取！

54.2.1.1 .env.testing 環境組態檔案

此外，你可以在項目的根目錄中建立一個 .env.testing 檔案。 當運行 PHPUnit 測試或使用 --env=testing 選項執行 Artisan 命令時，將不會使用 .env 檔案，而是使用此檔案。

54.2.1.2 CreatesApplication Trait

Laravel 包含一個 CreatesApplication Trait，該 Trait 應用於應用程式的基類 TestCase 。 這個 trait 包含一個 createApplication 方法，它在運行測試之前引導 Laravel 應用程式。 重要的是，應將此 trait 保留在其原始位置，因為某些功能（例如 Laravel 的平行測試功能）依賴於它。

54.3 建立測試

要建立新的測試用例，請使用 Artisan 命令： make:test 。 默認情況下，測試將放置在 tests/Feature 目錄中：

```
php artisan make:test UserTest
```

如果想在 tests/Unit 目錄中建立一個測試，你可以在執行 make:test 命令時使用 --unit 選項：

```
php artisan make:test UserTest --unit
```

如果想建立一個 Pest PHP 測試, 你可以為 `make:test` 命令提供 `--pest` 選項：

```
php artisan make:test UserTest --pest
php artisan make:test UserTest --unit --pest
```

> **技巧**
> 可以使用 Stub 定製來自訂測試。

生成測試後，你可以像通常使用 PHPUnit 那樣定義測試方法。要運行測試，請從終端執行 `vendor/bin/phpunit` 或 `php artisan test` 命令：

```
<?php

namespace Tests\\Unit;

use PHPUnit\\Framework\\TestCase;

class ExampleTest extends TestCase
{
    /**
     * 基礎測試樣例
     *
     * @return void
     */
    public function test_basic_test()
    {
        $this->assertTrue(true);
    }
}
```

> 注意：如果你在測試類中定義自己的 `setUp` 或 `tearDown` 方法，請務必在父類上呼叫各自的 `parent::setUp()` 或 `parent::tearDown()` 方法。

****運行測試****

正如前面提到的，編寫測試後，可以使用 `phpunit` 命令來執行測試：

```
./vendor/bin/phpunit
```

除了 `phpunit` 命令，你還可以使用 `test` Artisan 命令來運行你的測試。 Artisan 測試運行器提供了詳細的測試報告，以簡化開發和偵錯：

```
php artisan test
```

任何可以傳遞給 `phpunit` 命令的參數也可以傳遞給 Artisan `test` 命令：

```
php artisan test --testsuite=Feature --stop-on-failure
```

****平行運行測試****

默認情況下，`Laravel` 和 `PHPUnit` 在執行測試時，是在單處理程序中按照先後順序執行的。除此之外，通過多個處理程序同時運行測試，則可以大大減少運行測試所需的時間。首先，請確保你的應用程式已依賴於 `^5.3` 或更高版本的 `nunomaduro/collision` 依賴包。然後，在執行 `test` Artisan 命令時，請加入 `--parallel` 選項：

```
php artisan test --parallel
```

默認情況下，`Laravel` 將建立與電腦上可用 CPU 核心數量一樣多的處理程序。但是，你可以使用 `--processes` 選項來調整處理程序數：

```
php artisan test --parallel --processes=4
```

> 注意：在平行測試時，某些 PHPUnit 選項（例如 `--do-not-cache-result`）可能不可用。

54.3.1 平行測試和資料庫

Laravel 在執行平行測試時，自動為每個處理程序建立並遷移生成一個測試資料庫。這些測試資料庫將以每個處理程序唯一的處理程序令牌作為後綴。例如，如果你有兩個平行的測試處理程序，Laravel 將建立並使用 your_db_test_1 和 your_db_test_2 測試資料庫。

默認情況下，在多次呼叫 test Artisan 命令時，上一次的測試資料庫依然存在，以便下一次的 test 命令可以再次使用它們。但是，你可以使用 --recreate-databases 選項重新建立它們：

```
php artisan test --parallel --recreate-databases
```

54.3.2 平行測試鉤子

有時，你可能需要為應用程式測試準備某些資源，以便可以將它們安全地用於多個測試處理程序。

使用 ParallelTesting 門面，你就可以在處理程序或測試用例的 setUp 和 tearDown 上指定要執行的程式碼。給定的閉包將分別接收包含處理程序令牌和當前測試用例的 $token 和 $testCase 變數：

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Artisan;
use Illuminate\Support\Facades\ParallelTesting;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * 引導任何應用程式服務。
     *
     * @return void
     */
    public function boot()
    {
        ParallelTesting::setUpProcess(function ($token) {
            // ...
        });

        ParallelTesting::setUpTestCase(function ($token, $testCase) {
            // ...
        });

        // 在建立測試資料庫時執行……
        ParallelTesting::setUpTestDatabase(function ($database, $token) {
            Artisan::call('db:seed');
        });

        ParallelTesting::tearDownTestCase(function ($token, $testCase) {
            // ...
        });

        ParallelTesting::tearDownProcess(function ($token) {
            // ...
        });
    }
}
```

54.3.3 訪問平行測試令牌

如果你想從應用程式的測試程式碼中的任何其他位置訪問當前的平行處理程序的 token，則可以使用 token

方法。該令牌（token）是單個測試處理程序的唯一字串識別碼，可用於在平行測試過程中劃分資源。例如，Laravel 自動用此令牌值作為每個平行測試處理程序建立的測試資料庫名的後綴：

```
$token = ParallelTesting::token();
```

54.3.4 報告測試覆蓋率

> 注意：這個功能需要 Xdebug 或 PCOV。

在運行測試時，你可能需要確定測試用例是否真的測到了某些程式碼，以及在運行測試時究竟使用了多少應用程式程式碼。要實現這一點，你可以在呼叫 test 命令時，增加一個 --coverage 選項：

```
php artisan test --coverage
```

54.3.5 最小覆蓋率閾值限制

你可以使用 --min 選項來為你的應用程式定義一個最小測試覆蓋率閾值。如果不滿足此閾值，測試套件將失敗：

```
php artisan test --coverage --min=80.3
```

54.3.6 測試性能分析

Artisan 測試運行器還提供了一個方便的機制用於列出你的應用程式中最慢的測試。使用--profile 選項呼叫測試命令，可以看到 10 個最慢的測試列表，這可以讓你很容易地識別哪些測試可以被改進，以加快你的測試套件。

```
php artisan test --profile
```

55 HTTP 測試

55.1 簡介

Laravel 提供了一個非常流暢的 API，用於嚮應用程序發出 HTTP 請求並檢查響應。例如，看看下面定義的特性測試：

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * 基本功能測試示例。
     */
    public function test_a_basic_request(): void
    {
        $response = $this->get('/');

        $response->assertStatus(200);
    }
}
```

`get` 方法嚮應用程序發出 `Get` 請求，而 `assertStatus` 方法則斷言返回的響應應該具有給定的 HTTP 狀態程式碼。除了這個簡單的斷言之外，Laravel 還包含各種用於檢查響應頭、內容、JSON 結構等的斷言。

55.2 建立請求

要嚮應用程序發出請求，可以在測試中呼叫 `get`、`post`、`put`、`patch` 或 `delete` 方法。這些方法實際上不會嚮應用程序發出「真正的」HTTP 請求。相反，整個網路請求是在內部模擬的。

測試請求方法不返回 `Illuminate\Http\Response` 實例，而是返回 `Illuminate\Testing\TestResponse` 實例，該實例提供各種有用的斷言,允許你檢查應用程式的響應：

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * 基本功能測試示例。
     */
    public function test_a_basic_request(): void
    {
        $response = $this->get('/');

        $response->assertStatus(200);
    }
```

```
}
```

通常，你的每個測試應該只向你的應用發出一個請求。如果在單個測試方法中執行多個請求，則可能會出現意外行為。

> **技巧** 為了方便起見，運行測試時會自動停用 CSRF 中介軟體。

55.2.1 自訂要求標頭

你可以使用此 `withHeaders` 方法自訂請求的標頭，然後再將其傳送到應用程式。這使你可以將任何想要的自訂標頭新增到請求中：

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * 基本功能測試示例。
     */
    public function test_interacting_with_headers(): void
    {
        $response = $this->withHeaders([
            'X-Header' => 'Value',
        ])->post('/user', ['name' => 'Sally']);

        $response->assertStatus(201);
    }
}
```

55.2.2 Cookies

在傳送請求前你可以使用 `withCookie` 或 `withCookies` 方法設定 cookie。`withCookie` 接受 cookie 的名稱和值這兩個參數，而 `withCookies` 方法接受一個名稱 / 值對陣列：

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_interacting_with_cookies(): void
    {
        $response = $this->withCookie('color', 'blue')->get('/');

        $response = $this->withCookies([
            'color' => 'blue',
            'name' => 'Taylor',
        ])->get('/');
    }
}
```

55.2.3 session (Session) / 認證 (Authentication)

Laravel 提供了幾個可在 HTTP 測試時使用 Session 的輔助函數。首先，你需要傳遞一個陣列給 `withSession` 方法來設定 session 資料。這樣在應用程式的測試請求傳送之前，就會先去給資料載入 session：

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_interacting_with_the_session(): void
    {
        $response = $this->withSession(['banned' => false])->get('/');
    }
}
```

Laravel 的 session 通常用於維護當前已驗證使用者的狀態。因此，actingAs 方法提供了一種將給定使用者作為當前使用者進行身份驗證的便捷方法。例如，我們可以使用一個工廠模式來生成和認證一個使用者：

```
<?php

namespace Tests\Feature;

use App\Models\User;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_an_action_that_requires_authentication(): void
    {
        $user = User::factory()->create();

        $response = $this->actingAs($user)
                         ->withSession(['banned' => false])
                         ->get('/');
    }
}
```

你也可以通過傳遞看守器名稱作為 actingAs 方法的第二參數以指定使用者通過哪種看守器來認證。提供給 actingAs 方法的防護也將成為測試期間的默認防護。

```
$this->actingAs($user, 'web')
```

55.2.4 偵錯響應

在向你的應用程式發出測試請求之後，可以使用 dump、dumpHeaders 和 dumpSession 方法來檢查和偵錯響應內容：

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * 基本功能測試示例。
     */
    public function test_basic_test(): void
    {
        $response = $this->get('/');

        $response->dumpHeaders();

        $response->dumpSession();

        $response->dump();
```

```
    }
}
```

或者，你可以使用 dd、ddHeaders 和 ddSession 方法轉儲有關響應的資訊，然後停止執行：

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * 基本功能測試示例。
     */
    public function test_basic_test(): void
    {
        $response = $this->get('/');

        $response->ddHeaders();

        $response->ddSession();

        $response->dd();
    }
}
```

55.2.5 異常處理

有時你可能想要測試你的應用程式是否引發了特定異常。為了確保異常不會被 Laravel 的異常處理程序捕獲並作為 HTTP 響應返回，可以在發出請求之前呼叫 withoutExceptionHandling 方法：

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

此外，如果想確保你的應用程式沒有使用 PHP 語言或你的應用程式正在使用的庫已棄用的功能，你可以在發出請求之前呼叫 withoutDeprecationHandling 方法。停用棄用處理時，棄用警告將轉換為異常，從而導致你的測試失敗：

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

55.3 測試 JSON APIs

Laravel 也提供了幾個輔助函數來測試 JSON APIs 和其響應。例
如，json、getJson、postJson、putJson、patchJson、deleteJson 以及 optionsJson 可以被用於傳送各種 HTTP 動作。你也可以輕鬆地將資料和要求標頭傳遞到這些方法中。首先，讓我們實現一個測試示例，傳送 POST 請求到 /api/user，並斷言返回的期望資料：

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * 基本功能測試示例。
     */
    public function test_making_an_api_request(): void
    {
        $response = $this->postJson('/api/user', ['name' => 'Sally']);
```

```
        $response
            ->assertStatus(201)
            ->assertJson([
                'created' => true,
            ]);
    }
}
```

此外，JSON 響應資料可以作為響應上的陣列變數進行訪問，從而使你可以方便地檢查 JSON 響應中返回的各個值：

```
$this->assertTrue($response['created']);
```

> **技巧** `assertJson` 方法將響應轉換為陣列，並利用 `PHPUnit::assertArraySubset` 驗證給定陣列是否存在於應用程式返回的 JSON 響應中。因此，如果 JSON 響應中還有其他屬性，則只要存在給定的片段，此測試仍將通過。

55.3.1.1 驗證 JSON 完全匹配

如前所述，`assertJson` 方法可用於斷言 JSON 響應中存在 JSON 片段。如果你想驗證給定陣列是否與應用程式返回的 JSON **完全匹配**，則應使用 `assertExactJson` 方法：

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * 基本功能測試示例。
     */
    public function test_asserting_an_exact_json_match(): void
    {
        $response = $this->postJson('/user', ['name' => 'Sally']);

        $response
            ->assertStatus(201)
            ->assertExactJson([
                'created' => true,
            ]);
    }
}
```

55.3.1.2 驗證 JSON 路徑

如果你想驗證 JSON 響應是否包含指定路徑上的某些給定資料，可以使用 `assertJsonPath` 方法：

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    /**
     * 基本功能測試示例。
     */
    public function test_asserting_a_json_paths_value(): void
    {
        $response = $this->postJson('/user', ['name' => 'Sally']);
```

```
        $response
            ->assertStatus(201)
            ->assertJsonPath('team.owner.name', 'Darian');
    }
}
```

`assertJsonPath` 方法也接受一個閉包，可以用來動態地確定斷言是否應該通過。

```
$response->assertJsonPath('team.owner.name', fn (string $name) => strlen($name) >= 3);
```

55.3.2 JSON 流式測試

Laravel 還提供了一種漂亮的方式來流暢地測試應用程式的 JSON 響應。首先，將閉包傳遞給 `assertJson` 方法。這個閉包將使用 `Illuminate\Testing\Fluent\AssertableJson` 的實例呼叫，該實例可用於對應用程式返回的 JSON 進行斷言。 `where` 方法可用於對 JSON 的特定屬性進行斷言，而 `missing` 方法可用於斷言 JSON 中缺少特定屬性：

```
use Illuminate\Testing\Fluent\AssertableJson;

/**
 * 基本功能測試示例。
 */
public function test_fluent_json(): void
{
    $response = $this->getJson('/users/1');

    $response
        ->assertJson(fn (AssertableJson $json) =>
            $json->where('id', 1)
                 ->where('name', 'Victoria Faith')
                 ->where('email', fn (string $email) => str($email)-
>is('victoria@gmail.com'))
                 ->whereNot('status', 'pending')
                 ->missing('password')
                 ->etc()
        );
}
```

55.3.2.1 瞭解 `etc` 方法

在上面的例子中, 你可能已經注意到我們在斷言鏈的末端呼叫了 `etc` 方法. 這個方法通知 Laravel，在 JSON 對象上可能還有其他的屬性存在。如果沒有使用 `etc` 方法, 如果你沒有對 JSON 對象的其他屬性進行斷言, 測試將失敗.

這種行為背後的意圖是保護你不會在你的 JSON 響應中無意地暴露敏感資訊，因為它迫使你明確地對該屬性進行斷言或通過 `etc` 方法明確地允許額外的屬性。

然而，你應該知道，在你的斷言鏈中不包括 `etc` 方法並不能確保額外的屬性不會被新增到巢狀在 JSON 對象中的陣列。`etc` 方法只能確保在呼叫 `etc` 方法的巢狀層中不存在額外的屬性。

55.3.2.2 斷言屬性存在/不存在

要斷言屬性存在或不存在，可以使用 `has` 和 `missing` 方法：

```
$response->assertJson(fn (AssertableJson $json) =>
    $json->has('data')
         ->missing('message')
);
```

此外，`hasAll` 和 `missingAll` 方法允許同時斷言多個屬性的存在或不存在：

```
$response->assertJson(fn (AssertableJson $json) =>
```

```
    $json->hasAll(['status', 'data'])
         ->missingAll(['message', 'code'])
);
```

你可以使用 hasAny 方法來確定是否存在給定屬性列表中的至少一個：

```
$response->assertJson(fn (AssertableJson $json) =>
    $json->has('status')
         ->hasAny('data', 'message', 'code')
);
```

55.3.2.3 斷言反對 JSON 集合

通常，你的路由將返回一個 JSON 響應，其中包含多個項目，例如多個使用者：

```
Route::get('/users', function () {
    return User::all();
});
```

在這些情況下，我們可以使用 fluent JSON 對象的 has 方法對響應中包含的使用者進行斷言。例如，讓我們斷言 JSON 響應包含三個使用者。接下來，我們將使用 first 方法對集合中的第一個使用者進行一些斷言。first 方法接受一個閉包，該閉包接收另一個可斷言的 JSON 字串，我們可以使用它來對 JSON 集合中的第一個對象進行斷言：

```
$response
    ->assertJson(fn (AssertableJson $json) =>
        $json->has(3)
             ->first(fn (AssertableJson $json) =>
                $json->where('id', 1)
                     ->where('name', 'Victoria Faith')
                     ->where('email', fn (string $email) => str($email)-
>is('victoria@gmail.com'))
                     ->missing('password')
                     ->etc()
             )
    );
```

55.3.2.4 JSON 集合範圍斷言

有時，你的應用程式的路由將返回分配有命名鍵的 JSON 集合：

```
Route::get('/users', function () {
    return [
        'meta' => [...],
        'users' => User::all(),
    ];
})
```

在測試這些路由時，你可以使用 has 方法來斷言集合中的項目數。此外，你可以使用 has 方法來確定斷言鏈的範圍：

```
$response
    ->assertJson(fn (AssertableJson $json) =>
        $json->has('meta')
             ->has('users', 3)
             ->has('users.0', fn (AssertableJson $json) =>
                $json->where('id', 1)
                     ->where('name', 'Victoria Faith')
                     ->where('email', fn (string $email) => str($email)-
>is('victoria@gmail.com'))
                     ->missing('password')
                     ->etc()
             )
    );
```

但是，你可以進行一次呼叫，提供一個閉包作為其第三個參數，而不是對 has 方法進行兩次單獨呼叫來斷言 users 集合。這樣做時，將自動呼叫閉包並將其範圍限定為集合中的第一項：

```
$response
     ->assertJson(fn (AssertableJson $json) =>
          $json->has('meta')
                ->has('users', 3, fn (AssertableJson $json) =>
                   $json->where('id', 1)
                         ->where('name', 'Victoria Faith')
                         ->where('email', fn (string $email) => str($email)-
>is('victoria@gmail.com'))
                         ->missing('password')
                         ->etc()
                )
     );
```

55.3.2.5 斷言 JSON 類型

你可能只想斷言 JSON 響應中的屬性屬於某種類型。 Illuminate\Testing\Fluent\AssertableJson 類提供了 whereType 和 whereAllType 方法來做到這一點：

```
$response->assertJson(fn (AssertableJson $json) =>
     $json->whereType('id', 'integer')
           ->whereAllType([
              'users.0.name' => 'string',
              'meta' => 'array'
          ])
);
```

你可以使用 | 字元指定多種類型，或者將類型陣列作為第二個參數傳遞給 whereType 方法。如果響應值為任何列出的類型，則斷言將成功：

```
$response->assertJson(fn (AssertableJson $json) =>
     $json->whereType('name', 'string|null')
          ->whereType('id', ['string', 'integer'])
);
```

whereType 和 whereAllType 方法識別以下類型：string、integer、double、boolean、array 和 null。

55.4 測試檔案上傳

Illuminate\Http\UploadedFile 提供了一個 fake 方法用於生成虛擬的檔案或者圖像以供測試之用。它可以和 Storage facade 的 fake 方法相結合，大幅度簡化了檔案上傳測試。舉個例子，你可以結合這兩者的功能非常方便地進行頭像上傳表單測試：

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Http\UploadedFile;
use Illuminate\Support\Facades\Storage;
use Tests\TestCase;

class ExampleTest extends TestCase
{
     public function test_avatars_can_be_uploaded(): void
     {
          Storage::fake('avatars');

          $file = UploadedFile::fake()->image('avatar.jpg');
```

```
        $response = $this->post('/avatar', [
            'avatar' => $file,
        ]);

        Storage::disk('avatars')->assertExists($file->hashName());
    }
}
```

如果你想斷言一個給定的檔案不存在，則可以使用由 `Storage` facade 提供的 `AssertMissing` 方法：

```
Storage::fake('avatars');

// ...

Storage::disk('avatars')->assertMissing('missing.jpg');
```

55.4.1.1 虛擬檔案定製

當使用 `UploadedFile` 類提供的 `fake` 方法建立檔案時，你可以指定圖片的寬度、高度和大小（以千位元組為單位），以便更好地測試你的應用程式的驗證規則。

```
UploadedFile::fake()->image('avatar.jpg', $width, $height)->size(100);
```

除建立圖像外，你也可以用 `create` 方法建立其他類型的檔案：

```
UploadedFile::fake()->create('document.pdf', $sizeInKilobytes);
```

如果需要，可以向該方法傳遞一個 `$mimeType` 參數，以顯式定義檔案應返回的 MIME 類型：

```
UploadedFile::fake()->create(
    'document.pdf', $sizeInKilobytes, 'application/pdf'
);
```

55.5 測試檢視

Laravel 允許在不嚮應用程序發出模擬 HTTP 請求的情況下獨立呈現檢視。為此，可以在測試中使用 `view` 方法。`view` 方法接受檢視名稱和一個可選的資料陣列。這個方法返回一個 `Illuminate\Testing\TestView` 的實例，它提供了幾個方法來方便地斷言檢視的內容：

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_a_welcome_view_can_be_rendered(): void
    {
        $view = $this->view('welcome', ['name' => 'Taylor']);

        $view->assertSee('Taylor');
    }
}
```

`TestView` 對象提供了以下斷言方法：
`assertSee`、`assertSeeInOrder`、`assertSeeText`、`assertSeeTextInOrder`、`assertDontSee` 和 `assertDontSeeText`。

如果需要，你可以通過將 `TestView` 實例轉換為一個字串獲得原始的檢視內容：

```
$contents = (string) $this->view('welcome');
```

55.5.1.1 共享錯誤

一些檢視可能依賴於 Laravel 提供的 全域錯誤包 中共享的錯誤。要在錯誤包中生成錯誤消息，可以使用 `withViewErrors` 方法：

```
$view = $this->withViewErrors([
    'name' => ['Please provide a valid name.']
])->view('form');

$view->assertSee('Please provide a valid name.');
```

55.5.2 渲染範本 & 元件

必要的話，你可以使用 `blade` 方法來計算和呈現原始的 Blade 字串。與 `view` 方法一樣，`blade` 方法返回的是 `Illuminate\Testing\TestView` 的實例：

```
$view = $this->blade(
    '<x-component :name="$name" />',
    ['name' => 'Taylor']
);

$view->assertSee('Taylor');
```

你可以使用 `component` 方法來評估和渲染 Blade 元件。類似於 `view` 方法，`component` 方法返回一個 `Illuminate\Testing\TestView` 的實例：

```
$view = $this->component(Profile::class, ['name' => 'Taylor']);

$view->assertSee('Taylor');
```

55.6 可用斷言

55.6.1 響應斷言

Laravel 的 `Illuminate\Testing\TestResponse` 類提供了各種自訂斷言方法，你可以在測試應用程式時使用它們。可以在由 `json`、`get`、`post`、`put` 和 `delete` 方法返回的響應上訪問這些斷言：

assertCookie assertCookieExpired assertCookieNotExpired assertCookieMissing assertCreated assertDontSee assertDontSeeText assertDownload assertExactJson assertForbidden assertHeader assertHeaderMissing assertJson assertJsonCount assertJsonFragment assertJsonIsArray assertJsonIsObject assertJsonMissing assertJsonMissingExact assertJsonMissingValidationErrors assertJsonPath assertJsonMissingPath assertJsonStructure assertJsonValidationErrors assertJsonValidationErrorFor assertLocation assertContent assertNoContent assertStreamedContent assertNotFound assertOk assertPlainCookie assertRedirect assertRedirectContains assertRedirectToRoute assertRedirectToSignedRoute assertSee assertSeeInOrder assertSeeText assertSeeTextInOrder assertSessionHas assertSessionHasInput assertSessionHasAll assertSessionHasErrors assertSessionHasErrorsIn assertSessionHasNoErrors assertSessionDoesntHaveErrors assertSessionMissing assertStatus assertSuccessful assertUnauthorized assertUnprocessable assertValid assertInvalid assertViewHas assertViewHasAll assertViewIs assertViewMissing

55.6.1.1 assertCookie

斷言響應中包含給定的 cookie：

```
$response->assertCookie($cookieName, $value = null);
```

55.6.1.2 assertCookieExpired

斷言響應包含給定的過期的 cookie：

```
$response->assertCookieExpired($cookieName);
```

55.6.1.3 assertCookieNotExpired

斷言響應包含給定的未過期的 cookie：

```
$response->assertCookieNotExpired($cookieName);
```

55.6.1.4 assertCookieMissing

斷言響應不包含給定的 cookie:

```
$response->assertCookieMissing($cookieName);
```

55.6.1.5 assertCreated

斷言做狀態程式碼為 201 的響應：

```
$response->assertCreated();
```

55.6.1.6 assertDontSee

斷言給定的字串不包含在響應中。除非傳遞第二個參數 `false`，否則此斷言將給定字串進行轉義後匹配：

```
$response->assertDontSee($value, $escaped = true);
```

55.6.1.7 assertDontSeeText

斷言給定的字串不包含在響應文字中。除非你傳遞第二個參數 `false`，否則該斷言將自動轉義給定的字串。該方法將在做出斷言之前將響應內容傳遞給 PHP 的 `strip_tags` 函數：

```
$response->assertDontSeeText($value, $escaped = true);
```

55.6.1.8 assertDownload

斷言響應是「下載」。通常，這意味著返迴響應的呼叫路由返回了 `Response::download` 響應、`BinaryFileResponse` 或 `Storage::download` 響應：

```
$response->assertDownload();
```

如果你願意，你可以斷言可下載的檔案被分配了一個給定的檔案名稱：

```
$response->assertDownload('image.jpg');
```

55.6.1.9 assertExactJson

斷言響應包含與給定 JSON 資料的完全匹配：

```
$response->assertExactJson(array $data);
```

55.6.1.10 assertForbidden

斷言響應中有禁止訪問 (403) 狀態碼：

```
$response->assertForbidden();
```

55.6.1.11 assertHeader

斷言給定的 header 在響應中存在：

```
$response->assertHeader($headerName, $value = null);
```

55.6.1.12 assertHeaderMissing

斷言給定的 header 在響應中不存在：

```
$response->assertHeaderMissing($headerName);
```

55.6.1.13 assertJson

斷言響應包含給定的 JSON 資料：

```
$response->assertJson(array $data, $strict = false);
```

`AssertJson` 方法將響應轉換為陣列，並利用 `PHPUnit::assertArraySubset` 驗證給定陣列是否存在於應用程式返回的 JSON 響應中。因此，如果 JSON 響應中還有其他屬性，則只要存在給定的片段，此測試仍將通過。

55.6.1.14 assertJsonCount

斷言響應 JSON 中有一個陣列，其中包含給定鍵的預期元素數量：

```
$response->assertJsonCount($count, $key = null);
```

55.6.1.15 assertJsonFragment

斷言響應包含給定 JSON 片段：

```
Route::get('/users', function () {
    return [
        'users' => [
            [
                'name' => 'Taylor Otwell',
            ],
        ],
    ];
});

$response->assertJsonFragment(['name' => 'Taylor Otwell']);
```

55.6.1.16 assertJsonIsArray

斷言響應的 JSON 是一個陣列。

```
$response->assertJsonIsArray();
```

55.6.1.17 assertJsonIsObject

斷言響應的 JSON 是一個對象。

```
$response->assertJsonIsObject();
```

55.6.1.18 assertJsonMissing

斷言響應未包含給定的 JSON 片段：

```
$response->assertJsonMissing(array $data);
```

55.6.1.19 assertJsonMissingExact

斷言響應不包含確切的 JSON 片段：

```
$response->assertJsonMissingExact(array $data);
```

55.6.1.20 assertJsonMissingValidationErrors

斷言響應響應對於給定的鍵沒有 JSON 驗證錯誤：

```
$response->assertJsonMissingValidationErrors($keys);
```

提示 更通用的 assertValid 方法可用於斷言響應沒有以 JSON 形式返回的驗證錯誤**並且**沒有錯誤被閃現到 session 儲存中。

55.6.1.21 assertJsonPath

斷言響應包含指定路徑上的給定資料：

```
$response->assertJsonPath($path, $expectedValue);
```

例如，如果你的應用程式返回的 JSON 響應包含以下資料：

```
{
     "user": {
          "name": "Steve Schoger"
     }
}
```

你可以斷言 `user` 對象的 `name` 屬性匹配給定值，如下所示：

```
$response->assertJsonPath('user.name', 'Steve Schoger');
```

55.6.1.22 assertJsonMissingPath

斷言響應具有給定的 JSON 結構：

```
$response->assertJsonMissingPath($path);
```

例如，如果你的應用程式返回的 JSON 響應包含以下資料：

```
{
     "user": {
          "name": "Steve Schoger"
     }
}
```

你可以斷言它不包含 `user` 對象的 `email` 屬性。

```
$response->assertJsonMissingPath('user.email');
```

55.6.1.23 assertJsonStructure

斷言響應具有給定的 JSON 結構：

```
$response->assertJsonStructure(array $structure);
```

例如，如果你的應用程式返回的 JSON 響應包含以下資料：

```
{
     "user": {
          "name": "Steve Schoger"
     }
}
```

你可以斷言 JSON 結構符合你的期望，如下所示：

```
$response->assertJsonStructure([
     'user' => [
          'name',
     ]
]);
```

有時，你的應用程式返回的 JSON 響應可能包含對象陣列：

```
{
     "user": [
          {
               "name": "Steve Schoger",
               "age": 55,
               "location": "Earth"
```

```
        },
        {
            "name": "Mary Schoger",
            "age": 60,
            "location": "Earth"
        }
    ]
}
```

在這種情況下，你可以使用 * 字元來斷言陣列中所有對象的結構：

```
$response->assertJsonStructure([
    'user' => [
        '*' => [
            'name',
            'age',
            'location'
        ]
    ]
]);
```

55.6.1.24 assertJsonValidationErrors

斷言響應具有給定鍵的給定 JSON 驗證錯誤。在斷言驗證錯誤作為 JSON 結構返回而不是閃現到 session 的響應時，應使用此方法：

```
$response->assertJsonValidationErrors(array $data, $responseKey = 'errors');
```

> **技巧** 更通用的 assertInvalid 方法可用於斷言響應具有以 JSON 形式返回的驗證錯誤**或**錯誤已快閃記憶體到 session 儲存。

55.6.1.25 assertJsonValidationErrorFor

斷言響應對給定鍵有任何 JSON 驗證錯誤：

```
$response->assertJsonValidationErrorFor(string $key, $responseKey = 'errors');
```

55.6.1.26 assertLocation

斷言響應在 `Location` 頭部中具有給定的 URI 值：

```
$response->assertLocation($uri);
```

55.6.1.27 assertContent

斷言給定的字串與響應內容匹配。

```
$response->assertContent($value);
```

55.6.1.28 assertNoContent

斷言響應具有給定的 HTTP 狀態碼且沒有內容：

```
$response->assertNoContent($status = 204);
```

55.6.1.29 assertStreamedContent

斷言給定的字串與流式響應的內容相匹配。

```
$response->assertStreamedContent($value);
```

55.6.1.30 assertNotFound

斷言響應具有未找到（404）HTTP 狀態碼：

```
$response->assertNotFound();
```

55.6.1.31 assertOk

斷言響應有 200 狀態碼：

```
$response->assertOk();
```

55.6.1.32 assertPlainCookie

斷言響應包含給定的 cookie（未加密）：

```
$response->assertPlainCookie($cookieName, $value = null);
```

55.6.1.33 assertRedirect

斷言響應會重新導向到給定的 URI：

```
$response->assertRedirect($uri);
```

55.6.1.34 assertRedirectContains

斷言響應是否重新導向到包含給定字串的 URI：

```
$response->assertRedirectContains($string);
```

55.6.1.35 assertRedirectToRoute

斷言響應是對給定的命名路由的重新導向。

```
$response->assertRedirectToRoute($name = null, $parameters = []);
```

55.6.1.36 assertRedirectToSignedRoute

斷言響應是對給定簽名路由的重新導向：

```
$response->assertRedirectToSignedRoute($name = null, $parameters = []);
```

55.6.1.37 assertSee

斷言給定的字串包含在響應中。除非傳遞第二個參數 `false`，否則此斷言將給定字串進行轉義後匹配：

```
$response->assertSee($value, $escaped = true);
```

55.6.1.38 assertSeeInOrder

斷言給定的字串按順序包含在響應中。除非傳遞第二個參數 `false`，否則此斷言將給定字串進行轉義後匹配：

```
$response->assertSeeInOrder(array $values, $escaped = true);
```

55.6.1.39 assertSeeText

斷言給定字串包含在響應文字中。除非傳遞第二個參數 `false`，否則此斷言將給定字串進行轉義後匹配。在做出斷言之前，響應內容將被傳遞到 PHP 的 `strip_tags` 函數：

```
$response->assertSeeText($value, $escaped = true);
```

55.6.1.40 assertSeeTextInOrder

斷言給定的字串按順序包含在響應的文字中。除非傳遞第二個參數 `false`，否則此斷言將給定字串進行轉義後匹配。在做出斷言之前，響應內容將被傳遞到 PHP 的 `strip_tags` 函數：

```
$response->assertSeeTextInOrder(array $values, $escaped = true);
```

55.6.1.41 assertSessionHas

斷言 Session 包含給定的資料段：

```
$response->assertSessionHas($key, $value = null);
```

如果需要，可以提供一個閉包作為 `assertSessionHas` 方法的第二個參數。如果閉包返回 `true`，則斷言將通過：

```
$response->assertSessionHas($key, function (User $value) {
    return $value->name === 'Taylor Otwell';
});
```

55.6.1.42 assertSessionHasInput

session 在 快閃記憶體輸入陣列 中斷言具有給定值：

```
$response->assertSessionHasInput($key, $value = null);
```

如果需要，可以提供一個閉包作為 `assertSessionHasInput` 方法的第二個參數。如果閉包返回 `true`，則斷言將通過：

```
$response->assertSessionHasInput($key, function (string $value) {
    return Crypt::decryptString($value) === 'secret';
});
```

55.6.1.43 assertSessionHasAll

斷言 Session 中具有給定的鍵 / 值對列表：

```
$response->assertSessionHasAll(array $data);
```

例如，如果你的應用程式 session 包含 `name` 和 `status` 鍵，則可以斷言它們存在並且具有指定的值，如下所示：

```
$response->assertSessionHasAll([
    'name' => 'Taylor Otwell',
    'status' => 'active',
]);
```

55.6.1.44 assertSessionHasErrors

斷言 session 包含給定 `$keys` 的 Laravel 驗證錯誤。如果 `$keys` 是關聯陣列，則斷言 session 包含每個欄位（key）的特定錯誤消息（value）。測試將快閃記憶體驗證錯誤到 session 的路由時，應使用此方法，而不是將其作為 JSON 結構返回：

```
$response->assertSessionHasErrors(
    array $keys, $format = null, $errorBag = 'default'
);
```

例如，要斷言 `name` 和 `email` 欄位具有已快閃記憶體到 session 的驗證錯誤消息，可以呼叫 `assertSessionHasErrors` 方法，如下所示：

```
$response->assertSessionHasErrors(['name', 'email']);
```

或者，你可以斷言給定欄位具有特定的驗證錯誤消息：

```
$response->assertSessionHasErrors([
    'name' => 'The given name was invalid.'
]);
```

> **注意** 更加通用的 assertInvalid 方法可以用來斷言一個響應有驗證錯誤，以 JSON 形式返回，**或** 將錯誤被快閃記憶體到 session 儲存中。

55.6.1.45 assertSessionHasErrorsIn

斷言 session 在特定的錯誤包中包含給定 `$keys` 的錯誤。如果 `$keys` 是一個關聯陣列，則斷言該 session 在錯誤包內包含每個欄位（鍵）的特定錯誤消息（值）：

```
$response->assertSessionHasErrorsIn($errorBag, $keys = [], $format = null);
```

55.6.1.46 assertSessionHasNoErrors

斷言 session 沒有驗證錯誤：

```
$response->assertSessionHasNoErrors();
```

55.6.1.47 assertSessionDoesntHaveErrors

斷言 session 對給定鍵沒有驗證錯誤：

```
$response->assertSessionDoesntHaveErrors($keys = [], $format = null, $errorBag =
'default');
```

> **注意** 更加通用的 assertValid 方法可以用來斷言一個響應沒有以 JSON 形式返回的驗證錯誤，**同時** 不會將錯誤被快閃記憶體到 session 儲存中。

55.6.1.48 assertSessionMissing

斷言 session 中缺少指定的 $key：

```
$response->assertSessionMissing($key);
```

55.6.1.49 assertStatus

斷言響應指定的 http 狀態碼：

```
$response->assertStatus($code);
```

55.6.1.50 assertSuccessful

斷言響應一個成功的狀態碼 (>= 200 且 < 300) :

```
$response->assertSuccessful();
```

55.6.1.51 assertUnauthorized

斷言一個未認證的狀態碼 (401)：

```
$response->assertUnauthorized();
```

55.6.1.52 assertUnprocessable

斷言響應具有不可處理的實體 (422) HTTP 狀態程式碼：

```
$response->assertUnprocessable();
```

55.6.1.53 assertValid

斷言響應對給定鍵沒有驗證錯誤。此方法可用於斷言驗證錯誤作為 JSON 結構返回或驗證錯誤已閃現到 session 的響應：

```
// 斷言不存在驗證錯誤...
$response->assertValid();

// 斷言給定的鍵沒有驗證錯誤...
$response->assertValid(['name', 'email']);
```

55.6.1.54 assertInvalid

斷言響應對給定鍵有驗證錯誤。此方法可用於斷言驗證錯誤作為 JSON 結構返回或驗證錯誤已快閃記憶體到 session 的響應：

```
$response->assertInvalid(['name', 'email']);
```

你還可以斷言給定鍵具有特定的驗證錯誤消息。這樣做時，你可以提供整條消息或僅提供一部分消息：

```
$response->assertInvalid([
    'name' => 'The name field is required.',
    'email' => 'valid email address',
]);
```

55.6.1.55 assertViewHas

斷言為響應檢視提供了一個鍵值對資料：

```
$response->assertViewHas($key, $value = null);
```

將閉包作為第二個參數傳遞給 `assertViewHas` 方法將允許你檢查並針對特定的檢視資料做出斷言：

```
$response->assertViewHas('user', function (User $user) {
    return $user->name === 'Taylor';
});
```

此外，檢視資料可以作為陣列變數訪問響應，讓你可以方便地檢查它：

```
$this->assertEquals('Taylor', $response['name']);
```

55.6.1.56 assertViewHasAll

斷言響應檢視具有給定的資料列表：

```
$response->assertViewHasAll(array $data);
```

該方法可用於斷言該檢視僅包含與給定鍵匹配的資料：

```
$response->assertViewHasAll([
    'name',
    'email',
]);
```

或者，你可以斷言該檢視資料存在並且具有特定值：

```
$response->assertViewHasAll([
    'name' => 'Taylor Otwell',
    'email' => 'taylor@example.com,',
]);
```

55.6.1.57 assertViewIs

斷言當前路由返回的的檢視是給定的檢視：

```
$response->assertViewIs($value);
```

55.6.1.58 assertViewMissing

斷言給定的資料鍵不可用於應用程式響應中返回的檢視：

```
$response->assertViewMissing($key);
```

55.6.2 身份驗證斷言

Laravel 還提供了各種與身份驗證相關的斷言，你可以在應用程式的功能測試中使用它們。請注意，這些方法是在測試類本身上呼叫的，而不是由諸如 `get` 和 `post` 等方法返回的 `Illuminate\Testing\TestResponse` 實例。

55.6.2.1 assertAuthenticated

斷言使用者已通過身份驗證：

```
$this->assertAuthenticated($guard = null);
```

55.6.2.2 assertGuest

斷言使用者未通過身份驗證：

```
$this->assertGuest($guard = null);
```

55.6.2.3 assertAuthenticatedAs

斷言特定使用者已通過身份驗證：

```
$this->assertAuthenticatedAs($user, $guard = null);
```

55.7 驗證斷言

Laravel 提供了兩個主要的驗證相關的斷言，你可以用它來確保在你的請求中提供的資料是有效或無效的。

55.7.1.1 assertValid

斷言響應對於給定的鍵沒有驗證錯誤。該方法可用於斷言響應中的驗證錯誤是以 JSON 結構返回的，或者驗證錯誤已經閃現到 session 中。

```
// 斷言沒有驗證錯誤存在...
$response->assertValid();

//斷言給定的鍵沒有驗證錯誤...
$response->assertValid(['name', 'email']);
```

55.7.1.2 assertInvalid

斷言響應對給定的鍵有驗證錯誤。這個方法可用於斷言響應中的驗證錯誤是以 JSON 結構返回的，或者驗證錯誤已經被閃現到 session 中。

```
$response->assertInvalid(['name', 'email']);
```

你也可以斷言一個給定的鍵有一個特定的驗證錯誤資訊。當這樣做時，你可以提供整個消息或只提供消息的一小部分。

```
$response->assertInvalid([
    'name' => 'The name field is required.',
    'email' => 'valid email address',
]);
```

56 命令列測試

56.1 介紹

除了簡化 HTTP 測試之外，Laravel 還提供了一個簡單的 API 來測試應用程式的 自訂控制台命令。

56.2 期望成功/失敗

首先，讓我們探索如何對 Artisan 命令的退出程式碼進行斷言。為此，我們將使用 `artisan` 方法從我們的測試中呼叫 Artisan 命令。然後，我們將使用 `assertExitCode` 方法斷言該命令以給定的退出程式碼完成：

```
/**
 * 測試控制台命令。
 */
public function test_console_command(): void
{
    $this->artisan('inspire')->assertExitCode(0);
}
```

你可以使用 `assertNotExitCode` 方法斷言命令沒有以給定的退出程式碼退出：

```
$this->artisan('inspire')->assertNotExitCode(1);
```

當然，所有終端命令通常在成功時以 `0` 狀態碼退出，在不成功時以非零退出碼退出。因此，為方便起見，你可以使用 `assertSuccessful` 和 `assertFailed` 斷言來斷言給定命令是否以成功的退出程式碼退出：

```
$this->artisan('inspire')->assertSuccessful();

$this->artisan('inspire')->assertFailed();
```

56.3 期望輸入/輸出

Laravel 允許你使用 `expectsQuestion` 方法輕鬆 「mock」控制台命令的使用者輸入。此外，你可以使用 `assertExitCode` 和 `expectsOutput` 方法指定你希望通過控制台命令輸出的退出程式碼和文字。例如，考慮以下控制台命令：

```
Artisan::command('question', function () {
    $name = $this->ask('What is your name?');

    $language = $this->choice('Which language do you prefer?', [
        'PHP',
        'Ruby',
        'Python',
    ]);

    $this->line('Your name is '.$name.' and you prefer '.$language.'.');
});
```

你可以用下面的測試來測試這個命令，該測試利用了 `expectsQuestion`、`expectsOutput`、`doesntExpectOutput`、`expectsOutputToContain`、`doesntExpectOutputToContain` 和 `assertExitCode` 方法。

```
/**
 * 測試控制台命令。
 */
public function test_console_command(): void
```

```
{
    $this->artisan('question')
          ->expectsQuestion('What is your name?', 'Taylor Otwell')
          ->expectsQuestion('Which language do you prefer?', 'PHP')
          ->expectsOutput('Your name is Taylor Otwell and you prefer PHP.')
          ->doesntExpectOutput('Your name is Taylor Otwell and you prefer Ruby.')
          ->expectsOutputToContain('Taylor Otwell')
          ->doesntExpectOutputToContain('you prefer Ruby')
          ->assertExitCode(0);
}
```

56.3.1.1 確認期望

當編寫一個期望以「是」或「否」答案形式確認的命令時，你可以使用 `expectsConfirmation` 方法：

```
$this->artisan('module:import')
    ->expectsConfirmation('Do you really wish to run this command?', 'no')
    ->assertExitCode(1);
```

56.3.1.2 表格期望

如果你的命令使用 Artisan 的 `table` 方法顯示資訊表，則為整個表格編寫輸出預期會很麻煩。相反，你可以使用 `expectsTable` 方法。此方法接受表格的標題作為它的第一個參數和表格的資料作為它的第二個參數：

```
$this->artisan('users:all')
    ->expectsTable([
        'ID',
        'Email',
    ], [
        [1, 'taylor@example.com'],
        [2, 'abigail@example.com'],
    ]);
```

57 Laravel Dusk

57.1 介紹

Laravel Dusk 提供了一套富有表現力、易於使用的瀏覽器自動化和測試 API。默認情況下，Dusk 不需要在本地電腦上安裝 JDK 或 Selenium。相反，Dusk 使用一個獨立的 ChromeDriver 安裝包。你可以自由地使用任何其他相容 Selenium 的驅動程式。

57.2 安裝

為了開始使用，你需要先安裝 Google Chrome 並將 `laravel/dusk` Composer 依賴新增到你的項目中：

```
composer require --dev laravel/dusk
```

> **警告** 如果你手動註冊 Dusk 的服務提供者，在生產環境中 **絕不要** 註冊，因為這可能導致任意使用者能夠認證你的應用程式。

安裝 Dusk 包後，執行 `dusk:install` Artisan 命令。`dusk:install` 命令將會建立一個 `tests/Browser` 目錄，一個示例 Dusk 測試，並為你的作業系統安裝 Chrome 驅動程式二進制檔案：

```
php artisan dusk:install
```

接下來，在應用程式的 `.env` 檔案中設定 `APP_URL` 環境變數。該值應該與你用於在瀏覽器中訪問應用程式的 URL 匹配。

> **注意** 如果你正在使用 Laravel Sail 管理你的本地開發環境，請參閱 Sail 文件中有關組態和運行 Dusk 測試的內容。

57.2.1 管理 ChromeDriver 安裝

如果你想安裝與 Laravel Dusk 通過 `dusk:install` 命令安裝的不同版本的 ChromeDriver，則可以使用 `dusk:chrome-driver` 命令：

```
# 為你的作業系統安裝最新版本的 ChromeDriver...
php artisan dusk:chrome-driver

# 為你的作業系統安裝指定版本的 ChromeDriver...
php artisan dusk:chrome-driver 86

# 為所有支援的作業系統安裝指定版本的 ChromeDriver...
php artisan dusk:chrome-driver --all

# 為你的作業系統安裝與 Chrome / Chromium 檢測到的版本匹配的 ChromeDriver...
php artisan dusk:chrome-driver --detect
```

> **警告** Dusk 需要 `chromedriver` 二進制檔案可執行。如果你無法運行 Dusk，你應該使用以下命令確保二進制檔案可執行：`chmod -R 0755 vendor/laravel/dusk/bin/`。

57.2.2 使用其他瀏覽器

默認情況下，Dusk 使用 Google Chrome 和獨立的 ChromeDriver 安裝來運行你的瀏覽器測試。但是，你可以啟動自己的 Selenium 伺服器，並運行你希望的任何瀏覽器來運行測試。

要開始，請打開你的 tests/DuskTestCase.php 檔案，該檔案是你的應用程式的基本 Dusk 測試用例。在這個檔案中，你可以刪除對 startChromeDriver 方法的呼叫。這將停止 Dusk 自動啟動 ChromeDriver：

```
/**
 * 準備執行 Dusk 測試。
 *
 * @beforeClass
 */
public static function prepare(): void
{
    // static::startChromeDriver();
}
```

接下來，你可以修改 driver 方法來連接到你選擇的 URL 和連接埠。此外，你可以修改應該傳遞給 WebDriver 的“期望能力”：

```
use Facebook\WebDriver\Remote\RemoteWebDriver;

/**
 * 建立 RemoteWebDriver 實例。
 */
protected function driver(): RemoteWebDriver
{
    return RemoteWebDriver::create(
        'http://localhost:4444/wd/hub', DesiredCapabilities::phantomjs()
    );
}
```

57.3 入門

57.3.1 生成測試

要生成 Dusk 測試，請使用 dusk:make Artisan 命令。生成的測試將放在 tests/Browser 目錄中：

```
php artisan dusk:make LoginTest
```

57.3.2 在每次測試後重設資料庫

你編寫的大多數測試將與從應用程式資料庫檢索資料的頁面互動；然而，你的 Dusk 測試不應該使用 RefreshDatabase trait。RefreshDatabase trait 利用資料庫事務，這些事務將不適用或不可用於 HTTP 請求。相反，你有兩個選項：DatabaseMigrations trait 和 DatabaseTruncation trait。

57.3.2.1 使用資料庫遷移

DatabaseMigrations trait 會在每次測試之前運行你的資料庫遷移。但是，為了每次測試而刪除和重新建立資料庫表通常比截斷表要慢：

```
<?php

namespace Tests\Browser;

use App\Models\User;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Chrome;
use Tests\DuskTestCase;

class ExampleTest extends DuskTestCase
{
    use DatabaseMigrations;
}
```

警告 當執行 Dusk 測試時，不能使用 SQLite 記憶體資料庫。由於瀏覽器在其自己的處理程序中執行，因此它將無法訪問其他處理程序的記憶體資料庫。

57.3.2.2 使用資料庫截斷

在使用 `DatabaseTruncation` trait 之前，你必須使用 Composer 包管理器安裝 `doctrine/dbal` 包：

```
composer require --dev doctrine/dbal
```

`DatabaseTruncation` trait 將在第一次測試時遷移你的資料庫，以確保你的資料庫表已經被正確建立。但是，在後續測試中，資料庫表將僅被截斷 - 相比重新運行所有的資料庫遷移，這樣做可以提高速度：

```
<?php

namespace Tests\Browser;

use App\Models\User;
use Illuminate\Foundation\Testing\DatabaseTruncation;
use Laravel\Dusk\Chrome;
use Tests\DuskTestCase;

class ExampleTest extends DuskTestCase
{
    use DatabaseTruncation;
}
```

默認情況下，此 trait 將截斷除 `migrations` 表以外的所有表。如果你想自訂應該截斷的表，則可以在測試類上定義 `$tablesToTruncate` 屬性：

```
/**
 * 表示應該截斷哪些表。
 *
 * @var array
 */
protected $tablesToTruncate = ['users'];
```

或者，你可以在測試類上定義 `$exceptTables` 屬性，以指定應該從截斷中排除的表：

```
/**
 * 表示應該從截斷中排除哪些表。
 *
 * @var array
 */
protected $exceptTables = ['users'];
```

為了指定需要清空表格的資料庫連接，你可以在測試類中定義一個 `$connectionsToTruncate` 屬性：

```
/**
 * 表示哪些連接需要清空表格。
 *
 * @var array
 */
protected $connectionsToTruncate = ['mysql'];
```

57.3.3 運行測試

要運行瀏覽器測試，執行 `dusk` Artisan 命令：

```
php artisan dusk
```

如果上一次運行 `dusk` 命令時出現了測試失敗，你可以通過 `dusk:fails` 命令先重新運行失敗的測試，以節省時間：

```
php artisan dusk:fails
```

`dusk` 命令接受任何 PHPUnit 測試運行器通常接受的參數，例如你可以只運行給定組的測試：

```
php artisan dusk --group=foo
```

> **注意** 如果你正在使用 Laravel Sail 來管理本地開發環境，請參考 Sail 文件中有關組態和運行 Dusk 測試的部分。

57.3.3.1 手動啟動 ChromeDriver

默認情況下，Dusk 會自動嘗試啟動 ChromeDriver。如果對於你的特定系統無法自動啟動，你可以在運行 `dusk` 命令之前手動啟動 ChromeDriver。如果你選擇手動啟動 ChromeDriver，則應該註釋掉 `tests/DuskTestCase.php` 檔案中的以下程式碼：

```
/**
 * 為 Dusk 測試執行做準備。
 *
 * @beforeClass
 */
public static function prepare(): void
{
    // static::startChromeDriver();
}
```

此外，如果你在連接埠 9515 以外的連接埠上啟動 ChromeDriver，你需要修改同一類中的 `driver` 方法以反映正確的連接埠：

```
use Facebook\WebDriver\Remote\RemoteWebDriver;

/**
 * 建立 RemoteWebDriver 實例。
 */
protected function driver(): RemoteWebDriver
{
    return RemoteWebDriver::create(
        'http://localhost:9515', DesiredCapabilities::chrome()
    );
}
```

57.3.4 環境處理

如果要在運行測試時強制 Dusk 使用自己的環境檔案，請在項目根目錄中建立一個 `.env.dusk.{當前環境}` 檔案。例如，如果你將從你的 `local` 環境啟動 `dusk` 命令，你應該建立一個 `.env.dusk.local` 檔案。

在運行測試時，Dusk 將備份你的 `.env` 檔案，並將你的 Dusk 環境重新命名為 `.env`。測試完成後，會將你的 `.env` 檔案還原。

57.4 瀏覽器基礎知識

57.4.1 建立瀏覽器

為了開始學習，我們編寫一個測試，驗證我們能否登錄到我們的應用程式。生成測試後，我們可以修改它以導航到登錄頁面，輸入一些憑據並點選“登錄”按鈕。為了建立一個瀏覽器實例，你可以在 Dusk 測試中呼叫 `browse` 方法：

```
<?php

namespace Tests\Browser;

use App\Models\User;
use Illuminate\Foundation\Testing\DatabaseMigrations;
```

```
use Laravel\Dusk\Browser;
use Laravel\Dusk\Chrome;
use Tests\DuskTestCase;

class ExampleTest extends DuskTestCase
{
     use DatabaseMigrations;

     /**
      * 一個基本的瀏覽器測試示例。
      */
     public function test_basic_example(): void
     {
          $user = User::factory()->create([
               'email' => 'taylor@laravel.com',
          ]);

          $this->browse(function (Browser $browser) use ($user) {
               $browser->visit('/login')
                       ->type('email', $user->email)
                       ->type('password', 'password')
                       ->press('Login')
                       ->assertPathIs('/home');
          });
     }
}
```

如上面的例子所示，`browse` 方法接受一個閉包。瀏覽器實例將由 Dusk 自動傳遞給此閉包，並且是與應用程式互動和進行斷言的主要對象。

57.4.1.1 建立多個瀏覽器

有時你可能需要多個瀏覽器來正確地進行測試。例如，測試與 WebSockets 互動的聊天螢幕可能需要多個瀏覽器。要建立多個瀏覽器，只需將更多的瀏覽器參數新增到傳遞給 `browse` 方法的閉包簽名中即可：

```
$this->browse(function (Browser $first, Browser $second) {
     $first->loginAs(User::find(1))
            ->visit('/home')
            ->waitForText('Message');

     $second->loginAs(User::find(2))
             ->visit('/home')
             ->waitForText('Message')
             ->type('message', 'Hey Taylor')
             ->press('Send');

     $first->waitForText('Hey Taylor')
            ->assertSee('Jeffrey Way');
});
```

57.4.2 導航

`visit` 方法可用於在應用程式中導航到給定的 URI：

```
$browser->visit('/login');
```

你可以使用 `visitRoute` 方法來導航到 命名路由：

```
$browser->visitRoute('login');
```

你可以使用 `back` 和 `forward` 方法來導航「後退」和「前進」：

```
$browser->back();

$browser->forward();
```

你可以使用 refresh 方法來刷新頁面：

```
$browser->refresh();
```

57.4.3 調整瀏覽器窗口大小

你可以使用 resize 方法來調整瀏覽器窗口的大小：

```
$browser->resize(1920, 1080);
```

你可以使用 maximize 方法來最大化瀏覽器窗口：

```
$browser->maximize();
```

fitContent 方法將調整瀏覽器窗口的大小以匹配其內容的大小：

```
$browser->fitContent();
```

當測試失敗時，Dusk 將在擷取螢幕截圖之前自動調整瀏覽器大小以適合內容。你可以在測試中呼叫 disableFitOnFailure 方法來停用此功能：

```
$browser->disableFitOnFailure();
```

你可以使用 move 方法將瀏覽器窗口移動到螢幕上的其他位置：

```
$browser->move($x = 100, $y = 100);
```

57.4.4 瀏覽器宏

如果你想定義一個可以在各種測試中重複使用的自訂瀏覽器方法，可以在 Browser 類中使用 macro 方法。通常，你應該從服務提供者的 boot 方法中呼叫它：

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Laravel\Dusk\Browser;

class DuskServiceProvider extends ServiceProvider
{
    /**
     * 註冊 《Dusk》 的瀏覽器宏。
     */
    public function boot(): void
    {
        Browser::macro('scrollToElement', function (string $element = null) {
            $this->script("$('html, body').animate({ scrollTop: $
('$element').offset().top }, 0);");

            return $this;
        });
    }
}
```

該 macro 函數接收方法名作為其第一個參數，並接收閉包作為其第二個參數。 將宏作為 Browser 實現上的方法呼叫宏時，將執行宏的閉包：

```
$this->browse(function (Browser $browser) use ($user) {
    $browser->visit('/pay')
            ->scrollToElement('#credit-card-details')
            ->assertSee('Enter Credit Card Details');
});
```

57.4.5 使用者認證

我們經常會測試需要身份驗證的頁面，你可以使用 Dusk 的 `loginAs` 方法來避免在每次測試期間與登錄頁面進行互動。該 `loginAs` 方法接收使用者 ID 或者使用者模型實例

```
use App\Models\User;
use Laravel\Dusk\Browser;

$this->browse(function (Browser $browser) {
    $browser->loginAs(User::find(1))
          ->visit('/home');
});
```

注意 使用 `loginAs` 方法後，使用者 session 在檔案中的所有測試被維護。

57.4.6 Cookies

你可以使用 `cookie` 方法來獲取或者設定加密過的 cookie 的值：

```
$browser->cookie('name');

$browser->cookie('name', 'Taylor');
```

使用 `plainCookie` 則可以獲取或者設定未加密過的 cookie 的值：

```
$browser->plainCookie('name');

$browser->plainCookie('name', 'Taylor');
```

你可以使用 `deleteCookie` 方法刪除指定的 cookie：

```
$browser->deleteCookie('name');
```

57.4.7 運行 JavaScript

可以使用 `script` 方法在瀏覽器中執行任意 JavaScript 語句：

```
$browser->script('document.documentElement.scrollTop = 0');

$browser->script([
    'document.body.scrollTop = 0',
    'document.documentElement.scrollTop = 0',
]);

$output = $browser->script('return window.location.pathname');
```

57.4.8 獲取截圖

你可以使用 `screenshot` 方法來截圖並將其指定檔案名稱儲存，所有截圖都將存放在 `tests/Browser/screenshots` 目錄下：

```
$browser->screenshot('filename');
```

`responsiveScreenshots` 方法可用於在不同斷點處擷取一系列截圖:

```
$browser->responsiveScreenshots('filename');
```

57.4.9 控制台輸出結果保存到硬碟

你可以使用 `storeConsoleLog` 方法將控制台輸出指定檔案名稱並寫入磁碟，控制台輸出默認存放在 `tests/Browser/console` 目錄下：

```
$browser->storeConsoleLog('filename');
```

57.4.10 頁面原始碼保存到硬碟

你可以使用 `storeSource` 方法將頁面當前原始碼指定檔案名稱並寫入磁碟，頁面原始碼默認會存放到 `tests/Browser/source` 目錄：

```
$browser->storeSource('filename');
```

57.5 與元素互動

57.5.1 Dusk 選擇器

編寫 Dusk 測試最困難的部分之一就是選擇良好的 CSS 選擇器與元素進行互動。 隨著時間的推移，前端的更改可能會導致如下所示的 CSS 選擇器無法通過測試：

```
// HTML...

<button>Login</button>

// Test...

$browser->click('.login-page .container div > button');
```

Dusk 選擇器可以讓你專注於編寫有效的測試，而不必記住 CSS 選擇器。要定義一個選擇器，你需要新增一個 `dusk` 屬性在 HTML 元素中。然後在選擇器前面加上@用來在 Dusk 測試中操作元素：

```
// HTML...

<button dusk="login-button">Login</button>

// Test...

$browser->click('@login-button');
```

57.5.2 文字、值 & 屬性

57.5.2.1 獲取 & 設定值

Dusk 提供了多個方法用於和頁面元素的當前顯示文字、值和屬性進行互動，例如，要獲取匹配指定選擇器的元素的「值」，使用 `value` 方法：

```
// 獲取值...
$value = $browser->value('selector');

// 設定值...
$browser->value('selector', 'value');
```

你可以使用 `inputValue` 方法來獲取包含指定欄位名稱的輸入元素的「值」：

```
$value = $browser->inputValue('field');
```

57.5.2.2 獲取文字

該 `text` 方法可以用於獲取匹配指定選擇器元素文字：

```
$text = $browser->text('selector');
```

57.5.2.3 獲取屬性

最後，該 attribute 方法可以用於獲取匹配指定選擇器元素屬性：

```
$attribute = $browser->attribute('selector', 'value');
```

57.5.3 使用表單

57.5.3.1 輸入值

Dusk 提供了多種方法來與表單和輸入元素進行互動。首先，讓我們看一個在欄位中輸入值的示例：

```
$browser->type('email', 'taylor@laravel.com');
```

注意，儘管該方法在需要時接收，但是我們不需要將 CSS 選擇器傳遞給 type 方法。如果沒有提供 CSS 選擇器，Dusk 會搜尋包含指定 name 屬性的 input 或 textarea 欄位。

要想將文字附加到一個欄位之後而且不清除其內容， 你可以使用 append 方法：

```
$browser->type('tags', 'foo')
        ->append('tags', ', bar, baz');
```

你可以使用 clear 方法清除輸入值：

```
$browser->clear('email');
```

你可以使用 typeSlowly 方法指示 Dusk 緩慢鍵入。 默認情況下，Dusk 在兩次按鍵之間將暫停 100 毫秒。 要自訂按鍵之間的時間量，你可以將適當的毫秒數作為方法的第二個參數傳遞：

```
$browser->typeSlowly('mobile', '+1 (202) 555-5555');

$browser->typeSlowly('mobile', '+1 (202) 555-5555', 300);
```

你可以使用 appendSlowly 方法緩慢新增文字：

```
$browser->type('tags', 'foo')
        ->appendSlowly('tags', ', bar, baz');
```

57.5.3.2 下拉菜單

需要在下拉菜單中選擇值，你可以使用 select 方法。 類似於 type 方法， 該 select 方法並不是一定要傳入 CSS 選擇器。 當使用 select 方法時，你應該傳遞選項實際的值而不是它的顯示文字：

```
$browser->select('size', 'Large');
```

你也可以通過省略第二個參數來隨機選擇一個選項：

```
$browser->select('size');
```

通過將陣列作為 select 方法的第二個參數，可以指示該方法選擇多個選項：

```
$browser->select('categories', ['Art', 'Music']);
```

57.5.3.3 複選框

使用「check」 複選框時，你可以使用 check 方法。 像其他許多與 input 相關的方法，並不是必須傳入 CSS 選擇器。 如果精準的選擇器無法找到的時候，Dusk 會搜尋能夠與 name 屬性匹配的複選框：

```
$browser->check('terms');
```

該 uncheck 方法可用於「取消選中」複選框輸入：

```
$browser->uncheck('terms');
```

57.5.3.4 單選按鈕

使用 「select」中單選按鈕選項時，你可以使用 `radio` 這個方法。 像很多其他的與輸入相關的方法一樣， 它也並不是必須傳入 CSS 選擇器。如果精準的選擇器無法被找到的時候， Dusk 會搜尋能夠與 `name` 屬性或者 `value` 屬性相匹配的 `radio` 單選按鈕：

```
$browser->radio('size', 'large');
```

57.5.4 附件

該 `attach` 方法可以附加一個檔案到 `file`input 元素中。 像很多其他的與輸入相關的方法一樣，他也並不是必須傳入 CSS 選擇器。如果精準的選擇器沒有被找到的時候，Dusk 會搜尋與 `name` 屬性匹配的檔案輸入框：

```
$browser->attach('photo', __DIR__.'/photos/mountains.png');
```

> **注意** attach 方法需要使用 PHP`Zip` 擴展，你的伺服器必須安裝了此擴展。

57.5.5 點選按鈕

可以使用 `press` 方法來點選頁面上的按鈕元素。該 `press` 方法的第一個參數可以是按鈕的顯示文字，也可以是 CSS/ Dusk 選擇器：

```
$browser->press('Login');
```

提交表單時，許多應用程式在按下表單後會停用表單的提交按鈕，然後在表單提交的 HTTP 請求完成後重新啟用該按鈕。要按下按鈕並等待按鈕被重新啟用，可以使用 `pressAndWaitFor` 方法：

```
// 按下按鈕並等待最多 5 秒，它將被啟用…
$browser->pressAndWaitFor('Save');

// 按下按鈕並等待最多 1 秒，它將被啟用…
$browser->pressAndWaitFor('Save', 1);
```

57.5.6 點選連結

要點選連結，可以在瀏覽器實例下使用 `clickLink` 方法。該 `clickLink` 方法將點選指定文字的連結：

```
$browser->clickLink($linkText);
```

你可以使用 `seeLink` 方法來確定具有給定顯示文字的連結在頁面上是否可見：

```
if ($browser->seeLink($linkText)) {
    // ...
}
```

> **注意** 這些方法與 jQuery 互動。 如果頁面上沒有 jQuery，Dusk 會自動將其注入到頁面中，以便在測試期間可用。

57.5.7 使用鍵盤

該 `keys` 方法讓你可以再指定元素中輸入比 `type` 方法更加複雜的輸入序列。例如，你可以在輸入值的同時按下按鍵。在這個例子中，輸入 `taylor` 時，`shift` 鍵也同時被按下。當 `taylor` 輸入完之後， 將會輸入 `swift` 而不會按下任何按鍵：

```
$browser->keys('selector', ['{shift}', 'taylor'], 'swift');
```

`keys` 方法的另一個有價值的用例是向你的應用程式的主要 CSS 選擇器傳送「鍵盤快速鍵」組合：

```
$browser->keys('.app', ['{command}', 'j']);
```

技巧 所有修飾符鍵如{command}都包裹在{}字元中，並且與在 Facebook\WebDriver\WebDriverKeys 類中定義的常數匹配，該類可以在 GitHub 上找到.

57.5.8 使用滑鼠

57.5.8.1 點選元素

該 click 方法可用於「點選」與給定選擇器匹配的元素：

```
$browser->click('.selector');
```

該 clickAtXPath 方法可用於「點選」與給定 XPath 表示式匹配的元素：

```
$browser->clickAtXPath('//div[@class = "selector"]');
```

該 clickAtPoint 方法可用於「點選」相對於瀏覽器可視區域的給定坐標對上的最高元素：

```
$browser->clickAtPoint($x = 0, $y = 0);
```

該 doubleClick 方法可用於模擬滑鼠的連按兩下：

```
$browser->doubleClick();
```

該 rightClick 方法可用於模擬滑鼠的右擊：

```
$browser->rightClick();

$browser->rightClick('.selector');
```

該 clickAndHold 方法可用於模擬被點選並按住的滑鼠按鈕。 隨後呼叫 releaseMouse 方法將撤消此行為並釋放滑鼠按鈕：

```
$browser->clickAndHold()
        ->pause(1000)
        ->releaseMouse();
```

57.5.8.2 滑鼠懸停

該 mouseover 方法可用於與給定選擇器匹配的元素的滑鼠懸停動作：

```
$browser->mouseover('.selector');
```

57.5.8.3 拖放

該 drag 方法用於將與指定選擇器匹配的元素拖到其它元素：

```
$browser->drag('.from-selector', '.to-selector');
```

或者，可以在單一方向上拖動元素：

```
$browser->dragLeft('.selector', $pixels = 10);
$browser->dragRight('.selector', $pixels = 10);
$browser->dragUp('.selector', $pixels = 10);
$browser->dragDown('.selector', $pixels = 10);
```

最後，你可以將元素拖動給定的偏移量：

```
$browser->dragOffset('.selector', $x = 10, $y = 10);
```

57.5.9 JavaScript 對話方塊

Dusk 提供了各種與 JavaScript 對話方塊進行互動的方法。例如，你可以使用 waitForDialog 方法來等待 JavaScript 對話方塊的出現。此方法接受一個可選參數，該參數指示等待對話方塊出現多少秒：

```
$browser->waitForDialog($seconds = null);
```

該 assertDialogOpened 方法，斷言對話方塊已經顯示，並且其消息與給定值匹配：

```
$browser->assertDialogOpened('Dialog message');
```

typeInDialog 方法，在打開的 JavaScript 提示對話方塊中輸入給定值：

```
$browser->typeInDialog('Hello World');
```

acceptDialog 方法，通過點選確定按鈕關閉打開的 JavaScript 對話方塊：

```
$browser->acceptDialog();
```

dismissDialog 方法，通過點選取消按鈕關閉打開的 JavaScript 對話方塊（僅對確認對話方塊有效）：

```
$browser->dismissDialog();
```

57.5.10 選擇器作用範圍

有時可能希望在給定的選擇器範圍內執行多個操作。比如，可能想要斷言表格中存在某些文字，然後點選表格中的一個按鈕。那麼你可以使用 with 方法實現此需求。在傳遞給 with 方法的閉包內執行的所有操作都將限於原始選擇器：

```
$browser->with('.table', function (Browser $table) {
    $table->assertSee('Hello World')
          ->clickLink('Delete');
});
```

你可能偶爾需要在當前範圍之外執行斷言。 你可以使用 elsewhere 和 elsewhereWhenAvailable 方法來完成此操作：

```
 $browser->with('.table', function ($table) {
    // 當前範圍是 `body .table`...

    $browser->elsewhere('.page-title', function ($title) {
        // 當前範圍是 `body .page-title`...
        $title->assertSee('Hello World');
    });

    $browser->elsewhereWhenAvailable('.page-title', function ($title) {
        // 當前範圍是 `body .page-title`...
        $title->assertSee('Hello World');
    });
 });
```

57.5.11 等待元素

在測試大面積使用 JavaScript 的應用時，在進行測試之前，通常有必要 「等待」 某些元素或資料可用。Dusk 可輕鬆實現。使用一系列方法，可以等到頁面元素可用，甚至給定的 JavaScript 表示式執行結果為 true。

57.5.11.1 等待

如果需要測試暫停指定的毫秒數， 使用 pause 方法：

```
$browser->pause(1000);
```

如果你只需要在給定條件為 true 時暫停測試，請使用 pauseIf 方法:

```
$browser->pauseIf(App::environment('production'), 1000);
```

同樣地，如果你需要暫停測試，除非給定的條件是 true，你可以使用 pauseUnless 方法:

```
$browser->pauseUnless(App::environment('testing'), 1000);
```

57.5.11.2 等待選擇器

該 waitFor 方法可以用於暫停執行測試，直到頁面上與給定 CSS 選擇器匹配的元素被顯示。默認情況下，將在暫停超過 5 秒後拋出異常。如有必要，可以傳遞自訂超時時長作為其第二個參數：

```
// 等待選擇器不超過 5 秒...
$browser->waitFor('.selector');

// 等待選擇器不超過 1 秒...
$browser->waitFor('.selector', 1);
```

你也可以等待選擇器顯示給定文字：

```
//  等待選擇器不超過 5 秒包含給定文字...
$browser->waitForTextIn('.selector', 'Hello World');

//  等待選擇器不超過 1 秒包含給定文字...
$browser->waitForTextIn('.selector', 'Hello World', 1);
```

你也可以等待指定選擇器從頁面消失:

```
// 等待不超過 5 秒 直到選擇器消失...
$browser->waitUntilMissing('.selector');

// 等待不超過 1 秒 直到選擇器消失...
$browser->waitUntilMissing('.selector', 1);
```

或者，你可以等待與給定選擇器匹配的元素被啟用或停用：

```
// 最多等待 5 秒鐘，直到選擇器啟用...
$browser->waitUntilEnabled('.selector');

// 最多等待 1 秒鐘，直到選擇器啟用...
$browser->waitUntilEnabled('.selector', 1);

// 最多等待 5 秒鐘，直到選擇器被停用...
$browser->waitUntilDisabled('.selector');

// 最多等待 1 秒鐘，直到選擇器被停用...
$browser->waitUntilDisabled('.selector', 1);
```

57.5.11.3 限定範疇範圍（可用時）

有時，你或許希望等待給定選擇器出現，然後與匹配選擇器的元素進行互動。例如，你可能希望等到模態窗口可用，然後在模態窗口中點選「確定」按鈕。在這種情況下，可以使用 whenAvailable 方法。給定回呼內的所有要執行的元素操作都將被限定在起始選擇器上:

```
$browser->whenAvailable('.modal', function (Browser $modal) {
    $modal->assertSee('Hello World')
          ->press('OK');
});
```

57.5.11.4 等待文字

waitForText 方法可以用於等待頁面上給定文字被顯示：

```
// 等待指定文字不超過 5 秒...
$browser->waitForText('Hello World');

// 等待指定文字不超過 1 秒...
$browser->waitForText('Hello World', 1);
```

你可以使用 waitUntilMissingText 方法來等待，直到顯示的文字已從頁面中刪除為止:

```
// 等待 5 秒刪除文字...
$browser->waitUntilMissingText('Hello World');
```

```
// 等待 1 秒刪除文字...
$browser->waitUntilMissingText('Hello World', 1);
```

57.5.11.5 等待連結

`waitForLink` 方法用於等待給定連結文字在頁面上顯示:

```
// 等待連結 5 秒...
$browser->waitForLink('Create');

// 等待連結 1 秒...
$browser->waitForLink('Create', 1);
```

57.5.11.6 等待輸入

`waitForInput` 方法可用於等待，直到給定的輸入欄位在頁面上可見:

```
// 等待 5 秒的輸入…
$browser->waitForInput($field);

// 等待 1 秒的輸入…
$browser->waitForInput($field, 1);
```

57.5.11.7 等待頁面跳轉

當給出類似`$browser->assertPathIs('/home')`的路徑斷言時，如果`window.location.pathname`被非同步更新，斷言就會失敗。可以使用 `waitForLocation` 方法等待頁面跳轉到給定路徑：

```
$browser->waitForLocation('/secret');
```

`waitForLocation` 方法還可用於等待當前窗口位置成為完全限定的 URL：

```
$browser->waitForLocation('https://example.com/path');
```

還可以使用被命名的路由等待跳轉：

```
$browser->waitForRoute($routeName, $parameters);
```

57.5.11.8 等待頁面重新載入

如果要在頁面重新載入後斷言，可以使用 `waitForReload` 方法：

```
use Laravel\Dusk\Browser;

$browser->waitForReload(function (Browser $browser) {
    $browser->press('Submit');
})
->assertSee('Success!');
```

由於需要等待頁面重新載入通常發生在點選按鈕之後，為了方便起見，你可以使用 `clickAndWaitForReload` 方法：

```
$browser->clickAndWaitForReload('.selector')
         ->assertSee('something');
```

57.5.11.9 等待 JavaScript 表示式

有時候會希望暫停測試的執行，直到給定的 JavaScript 表示式執行結果為 `true`。可以使用 `waitUntil` 方法輕鬆地達成此目的。 通過這個方法執行表示式，不需要包含 `return` 關鍵字或者結束分號：

```
// 等待表示式為 true 5 秒時間...
$browser->waitUntil('App.data.servers.length > 0');

// 等待表示式為 true 1 秒時間...
$browser->waitUntil('App.data.servers.length > 0', 1);
```

57.5.11.10 等待 Vue 表示式

`waitUntilVue` 和 `waitUntilVueIsNot` 方法可以一直等待，直到 Vue 元件 的屬性包含給定的值：

```
// 一直等待，直到元件屬性包含給定的值...
$browser->waitUntilVue('user.name', 'Taylor', '@user');

// 一直等待，直到元件屬性不包含給定的值...
$browser->waitUntilVueIsNot('user.name', null, '@user');
```

57.5.11.11 等待 JavaScript 事件

`waitForEvent` 方法可用於暫停測試的執行，直到 JavaScript 事件發生:

```
$browser->waitForEvent('load');
```

事件監聽器附加到當前範疇，默認情況下是 `body` 元素。當使用範圍選擇器時，事件監聽器將被附加到匹配的元素上:

```
$browser->with('iframe', function (Browser $iframe) {
    // 等待 iframe 的載入事件…
    $iframe->waitForEvent('load');
});
```

你也可以提供一個選擇器作為 `waitForEvent` 方法的第二個參數，將事件監聽器附加到特定的元素上:

```
$browser->waitForEvent('load', '.selector');
```

你也可以等待 `document` 和 `window` 對象上的事件:

```
// 等待文件被滾動…
$browser->waitForEvent('scroll', 'document');

// 等待 5 秒，直到窗口大小被調整…
$browser->waitForEvent('resize', 'window', 5);
```

57.5.11.12 等待回呼

Dusk 中的許多 「wait」 方法都依賴於底層方法 waitUsing。你可以直接用這個方法去等待一個回呼函數返回 `waitUsing`。你可以直接用這個方法去等待一個回呼函數返回 `true`。該 `waitUsing` 方法接收一個最大的等待秒數，閉包執行間隔時間，閉包，以及一個可選的失敗資訊：

```
$browser->waitUsing(10, 1, function () use ($something) {
    return $something->isReady();
}, "有些東西沒有及時準備好。");
```

57.5.12 滾動元素到檢視中

有時你可能無法點選某個元素，因為該元素在瀏覽器的可見區域之外。該 `scrollIntoView` 方法可以將元素滾動到瀏覽器可視窗口內：

```
$browser->scrollIntoView('.selector')
        ->click('.selector');
```

57.6 可用的斷言

Dusk 提供了各種你可以對應用使用的斷言。所有可用的斷言羅列如下：

註：不列印

57.7 Pages

有時，測試需要按順序執行幾個複雜的操作。這會使測試程式碼更難閱讀和理解。 Dusk Pages 允許你定義語義化的操作，然後可以通過單一方法在給定頁面上執行這些操作。Pages 還可以為應用或單個頁面定義通用選擇器的快捷方式。

57.7.1 生成 Pages

`dusk:page`Artisan 命令可以生成頁面對象。所有的頁面對象都位於 `tests/Browser/Pages` 目錄：

```
php artisan dusk:page Login
```

57.7.2 組態 Pages

默認情況下，頁面具有三種方法：`url`、`assert` 和 `elements`。我們現在將討論 `url` 和 `assert` 方法。`elements` 方法將在下面更詳細地討論。

57.7.2.1 `url` 方法

`url` 方法應該返回表示頁面 URL 的路徑。 Dusk 將會在瀏覽器中使用這個 URL 來導航到具體頁面：

```
/**
 * 獲取頁面的 URL。
 *
 * @return string
 */
public function url()
{
    return '/login';
}
```

57.7.2.2 `assert` 方法

`assert` 方法可以作出任何斷言來驗證瀏覽器是否在指定頁面上。實際上沒有必要在這個方法中放置任何東西；但是，你可以按自己的需求來做出這些斷言。導航到頁面時，這些斷言將自動運行：

```
/**
 * 斷言瀏覽器當前處於指定頁面。
 */
public function assert(Browser $browser): void
{
    $browser->assertPathIs($this->url());
}
```

57.7.3 導航至頁面

一旦頁面定義好之後，你可以使用 `visit` 方法導航至頁面：

```
use Tests\Browser\Pages\Login;

$browser->visit(new Login);
```

有時你可能已經在給定的頁面上，需要將頁面的選擇器和方法「載入」到當前的測試上下文中。 這在通過按鈕重新導向到指定頁面而沒有明確導航到該頁面時很常見。 在這種情況下，你可以使用 on 方法載入頁面：

```
use Tests\Browser\Pages\CreatePlaylist;

$browser->visit('/dashboard')
        ->clickLink('Create Playlist')
```

```
        ->on(new CreatePlaylist)
        ->assertSee('@create');
```

57.7.4 選擇器簡寫

該 `elements` 方法允許你為頁面中的任何 CSS 選擇器定義簡單易記的簡寫。例如，讓我們為應用登錄頁中的 email 輸入框定義一個簡寫：

```
/**
 * 獲取頁面元素的簡寫。
 *
 * @return array<string, string>
 */
public function elements(): array
{
    return [
        '@email' => 'input[name=email]',
    ];
}
```

一旦定義了簡寫，你就可以用這個簡寫來代替之前在頁面中使用的完整 CSS 選擇器：

```
$browser->type('@email', 'taylor@laravel.com');
```

57.7.4.1 全域的選擇器簡寫

安裝 Dusk 之後，`Page` 基類存放在你的 `tests/Browser/Pages` 目錄。該類中包含一個 `siteElements` 方法，這個方法可以用來定義全域的選擇器簡寫，這樣在你應用中每個頁面都可以使用這些全域選擇器簡寫了：

```
/**
 * 獲取站點全域的選擇器簡寫。
 *
 * @return array<string, string>
 */
public static function siteElements(): array
{
    return [
        '@element' => '#selector',
    ];
}
```

57.7.5 頁面方法

除了頁面中已經定義的默認方法之外，你還可以定義在整個測試過程中會使用到的其他方法。例如，假設我們正在開發一個音樂管理應用，在應用中每個頁面都可能需要一個公共的方法來建立播放列表，而不是在每一個測試類中都重寫一遍建立播放列表的邏輯，這時候你可以在你的頁面類中定義一個 `createPlaylist` 方法：

```
<?php

namespace Tests\Browser\Pages;

use Laravel\Dusk\Browser;

class Dashboard extends Page
{
    // 其他頁面方法...

    /**
     * 建立一個新的播放列表。
     */
    public function createPlaylist(Browser $browser, string $name): void
    {
        $browser->type('name', $name)
```

```
                ->check('share')
                ->press('Create Playlist');
    }
}
```

方法被定義之後，你可以在任何使用到該頁的測試中使用它了。瀏覽器實例會自動作為第一個參數傳遞給自訂頁面方法：

```
use Tests\Browser\Pages\Dashboard;

$browser->visit(new Dashboard)
        ->createPlaylist('My Playlist')
        ->assertSee('My Playlist');
```

57.8 元件

元件類似於 Dusk 的 「頁面對象」，不過它更多的是貫穿整個應用程式中頻繁重用的 UI 和功能片斷，比如說導覽列或資訊通知彈窗。因此，元件並不會繫結於某個明確的 URL。

57.8.1 生成元件

使用 `dusk:component` Artisan 命令即可生成元件。新生成的元件位於 `tests/Browser/Components` 目錄下：

```
php artisan dusk:component DatePicker
```

如上所示，這是生成一個「日期選擇器」（date picker）元件的示例，這個元件可能會貫穿使用在你應用程式的許多頁面中。在整個測試套件的大量測試頁面中，手動編寫日期選擇的瀏覽器自動化邏輯會非常麻煩。 更方便的替代辦法是，定義一個表示日期選擇器的 Dusk 元件，然後把自動化邏輯封裝在該元件內：

```
<?php

namespace Tests\Browser\Components;

use Laravel\Dusk\Browser;
use Laravel\Dusk\Component as BaseComponent;

class DatePicker extends BaseComponent
{
    /**
     * 獲取元件的 root selector。
     */
    public function selector(): string
    {
        return '.date-picker';
    }

    /**
     * 斷言瀏覽器包含元件。
     */
    public function assert(Browser $browser): void
    {
        $browser->assertVisible($this->selector());
    }

    /**
     * 讀取元件的元素簡寫。
     *
     * @return array<string, string>
     */
    public function elements(): array
    {
```

```
        return [
            '@date-field' => 'input.datepicker-input',
            '@year-list' => 'div > div.datepicker-years',
            '@month-list' => 'div > div.datepicker-months',
            '@day-list' => 'div > div.datepicker-days',
        ];
    }

    /**
     * 選擇給定日期。
     */
    public function selectDate(Browser $browser, int $year, int $month, int $day): void
    {
        $browser->click('@date-field')
                ->within('@year-list', function (Browser $browser) use ($year) {
                    $browser->click($year);
                })
                ->within('@month-list', function (Browser $browser) use ($month) {
                    $browser->click($month);
                })
                ->within('@day-list', function (Browser $browser) use ($day) {
                    $browser->click($day);
                });
    }
}
```

57.8.2 使用元件

當元件被定義了之後，我們就可以輕鬆的在任意測試頁面通過日期選擇器選擇一個日期。並且，如果選擇日期的邏輯發生了變化，我們只需要更新元件即可：

```
<?php

namespace Tests\Browser;

use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Browser;
use Tests\Browser\Components\DatePicker;
use Tests\DuskTestCase;

class ExampleTest extends DuskTestCase
{
    /**
     * 一個基礎的元件測試用例.
     */
    public function test_basic_example(): void
    {
        $this->browse(function (Browser $browser) {
            $browser->visit('/')
                    ->within(new DatePicker, function (Browser $browser) {
                        $browser->selectDate(2019, 1, 30);
                    })
                    ->assertSee('January');
        });
    }
}
```

57.9 持續整合

> **注意** 大多數 Dusk 持續整合組態都希望你的 Laravel 應用程式使用連接埠 8000 上的內建 PHP 開發伺服器提供服務。因此，你應該確保持續整合環境有一個值為 `http://127.0.0.1:8000` 的 `APP_URL` 環境變數。

57.9.1 Heroku CI

要在 Heroku CI 中運行 Dusk 測試，請將以下 Google Chrome buildpack 和 指令碼新增到 Heroku 的 `app.json` 檔案中：

```
{
  "environments": {
    "test": {
      "buildpacks": [
        { "url": "heroku/php" },
        { "url": "https://github.com/heroku/heroku-buildpack-google-chrome" }
      ],
      "scripts": {
        "test-setup": "cp .env.testing .env",
        "test": "nohup bash -c './vendor/laravel/dusk/bin/chromedriver-linux >
/dev/null 2>&1 &' && nohup bash -c 'php artisan serve --no-reload > /dev/null 2>&1 &' &&
php artisan dusk"
      }
    }
  }
}
```

57.9.2 Travis CI

要在 Travis CI 運行 Dusk 測試，可以使用下面這個 `.travis.yml` 組態。由於 Travis CI 不是一個圖形化的環境，我們還需要一些額外的步驟以便啟動 Chrome 瀏覽器。此外，我們將會使用 `php artisan serve` 來啟動 PHP 自帶的 Web 伺服器：

```
language: php

php:
  - 7.3

addons:
  chrome: stable

install:
  - cp .env.testing .env
  - travis_retry composer install --no-interaction --prefer-dist
  - php artisan key:generate
  - php artisan dusk:chrome-driver

before_script:
  - google-chrome-stable --headless --disable-gpu --remote-debugging-port=9222
http://localhost &
  - php artisan serve --no-reload &

script:
  - php artisan dusk
```

57.9.3 GitHub Actions

如果你正在使用 Github Actions 來運行你的 Dusk 測試，你應該使用以下這份組態檔案為範本。像 TravisCI 一樣，

我們使用 `php artisan serve` 命令來啟動 PHP 的內建 Web 服務：

```
name: CI
on: [push]
jobs:

  dusk-php:
    runs-on: ubuntu-latest
    env:
      APP_URL: "http://127.0.0.1:8000"
      DB_USERNAME: root
      DB_PASSWORD: root
      MAIL_MAILER: log
    steps:
      - uses: actions/checkout@v3
      - name: Prepare The Environment
        run: cp .env.example .env
      - name: Create Database
        run: |
          sudo systemctl start mysql
          mysql --user="root" --password="root" -e "CREATE DATABASE \`my-database\`
character set UTF8mb4 collate utf8mb4_bin;"
      - name: Install Composer Dependencies
        run: composer install --no-progress --prefer-dist --optimize-autoloader
      - name: Generate Application Key
        run: php artisan key:generate
      - name: Upgrade Chrome Driver
        run: php artisan dusk:chrome-driver --detect
      - name: Start Chrome Driver
        run: ./vendor/laravel/dusk/bin/chromedriver-linux &
      - name: Run Laravel Server
        run: php artisan serve --no-reload &
      - name: Run Dusk Tests
        run: php artisan dusk
      - name: Upload Screenshots
        if: failure()
        uses: actions/upload-artifact@v2
        with:
          name: screenshots
          path: tests/Browser/screenshots
      - name: Upload Console Logs
        if: failure()
        uses: actions/upload-artifact@v2
        with:
          name: console
          path: tests/Browser/console
```

58 資料庫測試

58.1 介紹

Laravel 提供了各種有用的工具和斷言，從而讓測試資料庫驅動變得更加容易。除此之外，Laravel 模型工廠和 Seeders 可以輕鬆地使用應用程式的 Eloquent 模型和關係來建立測試資料庫記錄。

58.1.1 每次測試後重設資料庫

在進行測試之前，讓我們討論一下如何在每次測試後重設資料庫，以便讓先前測試的資料不會干擾後續測試。Laravel 包含的 `Illuminate\Foundation\Testing\RefreshDatabase` trait 會為你解決這個問題。只需在您的測試類上使用該 Trait：

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    use RefreshDatabase;

    /**
     * 一個基本的功能測試例子。
     */
    public function test_basic_example(): void
    {
        $response = $this->get('/');

        // ...
    }
}
```

如果你的資料庫結構是最新的，那麼這個 Trait`Illuminate\Foundation\Testing\RefreshDatabase` 並不會遷移資料庫。相反，它只會在一個資料庫事務中執行測試。因此，任何由測試用例新增到資料庫的記錄，如果不使用這個 Trait，可能仍然存在於資料庫中。

如果你想使用遷移來完全重設資料庫，可以使用這個 Trait `Illuminate\Foundation\Testing\DatabaseMigrations` 來替代。然而，`DatabaseMigrations` 這個 Trait 明顯比 `RefreshDatabase` Trait 要慢。

58.2 模型工廠

當我們測試的時候，可能需要在執行測試之前向資料庫插入一些資料。Laravel 允許你使用 模型工廠 為每個 Eloquent 模型 定義一組預設值，而不是在建立測試資料時手動指定每一列的值。

要學習有關建立和使用模型工廠來建立模型的更多資訊，請參閱完整的 模型工廠文件。定義模型工廠後，你可以在測試中使用該工廠來建立模型：

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

```
public function test_models_can_be_instantiated(): void
{
    $user = User::factory()->create();

    // ...
}
```

58.3 運行 seeders

如果你在功能測試時希望使用 資料庫 seeders 來填充你的資料庫， 你可以呼叫 `seed` 方法。 默認情況下，`seed` 方法將會執行 `DatabaseSeeder`， 它將會執行你的所有其他 seeders。或者，你傳遞指定的 seeder 類名給 `seed` 方法：

```
<?php

namespace Tests\Feature;

use Database\Seeders\OrderStatusSeeder;
use Database\Seeders\TransactionStatusSeeder;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    use RefreshDatabase;

    /**
     * 測試建立新訂單。
     */
    public function test_orders_can_be_created(): void
    {
        // 運行 DatabaseSeeder...
        $this->seed();

        // 運行指定 seeder...
        $this->seed(OrderStatusSeeder::class);

        // ...

        // 運行指定的 seeders...
        $this->seed([
            OrderStatusSeeder::class,
            TransactionStatusSeeder::class,
            // ...
        ]);
    }
}
```

或者通過 `RefreshDatabase` trait 在每次測試之前自動為資料庫填充資料。你也可以通過在測試類上定義 `$seed` 屬性來實現：

```
<?php

namespace Tests;

use Illuminate\Foundation\Testing\TestCase as BaseTestCase;

abstract class TestCase extends BaseTestCase
{
    use CreatesApplication;

    /**
     * Indicates whether the default seeder should run before each test.
     *
```

```
     * @var bool
     */
    protected $seed = true;
}
```

當 `$seed` 屬性為 `true` 時，這個測試將在每個使用 `RefreshDatabase` trait 的測試之前運行 `Database\Seeders\DatabaseSeeder` 類。但是，你可以通過在測試類上定義 `$seeder` 屬性來指定要執行的 seeder：

```
use Database\Seeders\OrderStatusSeeder;

/**
 * 在測試前指定要運行的 seeder
 *
 * @var string
 */
protected $seeder = OrderStatusSeeder::class;
```

58.4 可用的斷言

Laravel 為你的 PHPUnit 功能測試提供了幾個資料庫斷言。我們將在下面逐個討論。

58.4.1.1 assertDatabaseCount

斷言資料庫中的表包含給定數量的記錄：

```
$this->assertDatabaseCount('users', 5);
```

58.4.1.2 assertDatabaseHas

斷言資料庫中的表包含給定鍵/值查詢約束的記錄：

```
$this->assertDatabaseHas('users', [
    'email' => 'sally@example.com',
]);
```

58.4.1.3 assertDatabaseMissing

斷言資料庫中的表不包含給定鍵/值查詢約束的記錄：

```
$this->assertDatabaseMissing('users', [
    'email' => 'sally@example.com',
]);
```

58.4.1.4 assertSoftDeleted

`assertSoftDeleted` 方法斷言給定的 Eloquent 模型已被「軟刪除」的記錄：

```
$this->assertSoftDeleted($user);
```

58.4.1.5 assertNotSoftDeleted

`assertNotSoftDeleted` 方法斷言給定的 Eloquent 模型沒有被「軟刪除」的記錄：

```
$this->assertNotSoftDeleted($user);
```

58.4.1.6 assertModelExists

斷言資料庫中存在給定的模型：

```
use App\Models\User;

$user = User::factory()->create();
```

```
$this->assertModelExists($user);
```

58.4.1.7 assertModelMissing

斷言資料庫中不存在給定的模型：

```
use App\Models\User;

$user = User::factory()->create();

$user->delete();

$this->assertModelMissing($user);
```

58.4.1.8 expectsDatabaseQueryCount

可以在測試開始時呼叫 `expectsDatabaseQueryCount` 方法，以指定你希望在測試期間運行的資料庫查詢總數。如果實際執行的查詢數量與這個預期不完全匹配，那麼測試將失敗：

```
$this->expectsDatabaseQueryCount(5);

// Test...
```

59 Mocking

59.1 介紹

在 Laravel 應用程式測試中，你可能希望「模擬」應用程式的某些功能的行為，從而避免該部分在測試中真正執行。例如：在 controller 執行過程中會觸發事件，您可能希望模擬事件監聽器，從而避免該事件在測試時真正執行。這允許你在僅測試 controller HTTP 響應的情況時，而不必擔心觸發事件，因為事件偵聽器可以在它們自己的測試用例中進行測試。

Laravel 針對事件、任務和 Facades 的模擬，提供了開箱即用的輔助函數。這些函數基於 `Mockery` 封裝而成，使用非常方便，無需手動呼叫複雜的 `Mockery` 函數。

59.2 模擬對象

當模擬一個對象將通過 Laravel 的 服務容器 注入到應用中時，你將需要將模擬實例作為 `instance` 繫結到容器中。這將告訴容器使用對象的模擬實例，而不是構造對象的本身：

```
use App\Service;
use Mockery;
use Mockery\MockInterface;

public function test_something_can_be_mocked(): void
{
     $this->instance(
          Service::class,
          Mockery::mock(Service::class, function (MockInterface $mock) {
               $mock->shouldReceive('process')->once();
          })
     );
}
```

為了讓以上過程更便捷，你可以使用 Laravel 的基本測試用例類提供的 `mock` 方法。例如，下面的例子跟上面的例子的執行效果是一樣的：

```
use App\Service;
use Mockery\MockInterface;

$mock = $this->mock(Service::class, function (MockInterface $mock) {
     $mock->shouldReceive('process')->once();
});
```

當你只需要模擬對象的幾個方法時，可以使用 `partialMock` 方法。 未被模擬的方法將在呼叫時正常執行：

```
use App\Service;
use Mockery\MockInterface;

$mock = $this->partialMock(Service::class, function (MockInterface $mock) {
     $mock->shouldReceive('process')->once();
});
```

同樣，如果你想 監控 一個對象，Laravel 的基本測試用例類提供了一個便捷的 `spy` 方法作為 `Mockery::spy` 的替代方法。`spies` 與模擬類似。但是，`spies` 會記錄 `spy` 與被測試程式碼之間的所有互動，從而允許您在執行程式碼後做出斷言：

```
use App\Service;

$spy = $this->spy(Service::class);

// ...
```

```
$spy->shouldHaveReceived('process');
```

59.3 Facades 模擬

與傳統靜態方法呼叫不同的是，facades (包含的 real-time facades) 也可以被模擬。相較傳統的靜態方法而言，它具有很大的優勢，同時提供了與傳統依賴注入相同的可測試性。在測試中，你可能想在 controller 中模擬對 Laravel Facade 的呼叫。比如下面 controller 中的行為：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

class UserController extends Controller
{
    /**
     * 顯示該應用程式的所有使用者的列表。
     */
    public function index(): array
    {
        $value = Cache::get('key');

        return [
            // ...
        ];
    }
}
```

我們可以使用 `shouldReceive` 方法模擬對 `Cache` Facade 的呼叫，該方法將返回一個 Mockery 模擬的實例。由於 Facades 實際上是由 Laravel 服務容器 解析和管理的，因此它們比傳統的靜態類具有更好的可測試性。例如，讓我們模擬對 `Cache` Facade 的 `get` 方法的呼叫：

```
<?php

namespace Tests\Feature;

use Illuminate\Support\Facades\Cache;
use Tests\TestCase;

class UserControllerTest extends TestCase
{
    public function test_get_index(): void
    {
        Cache::shouldReceive('get')
                    ->once()
                    ->with('key')
                    ->andReturn('value');

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

        // ...
    }
}
```

> **注意** 你不應該模擬 `Request` facade。相反，在運行測試時將你想要的輸入傳遞到 HTTP 測試方法 中，例如 `get` 和 `post` 方法。同樣也不要模擬 `Config` facade，而是在測試中呼叫 `Config::set` 方法。

59.3.1 Facade Spies

如果你想 監控 一個 facade，你可以在相應的 facade 上呼叫 `spy` 方法。`spies` 類似於 `mocks`；但是，`spies` 會

記錄 spy 和被測試程式碼之間的所有互動，允許你在程式碼執行後做出斷言：

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

public function test_values_are_be_stored_in_cache(): void
{
    Cache::spy();

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

    $response->assertStatus(200);

    Cache::shouldHaveReceived('put')->once()->with('name', 'Taylor', 10);
}
```

59.4 設定時間

當我們測試時，有時可能需要修改諸如 now 或 Illuminate\Support\Carbon::now() 之類的助手函數返回的時間。 Laravel 的基本功能測試類包中包含了一些助手函數，可以讓你設定當前時間：

```
use Illuminate\Support\Carbon;

public function test_time_can_be_manipulated(): void
{
    // 設定未來的時間...
    $this->travel(5)->milliseconds();
    $this->travel(5)->seconds();
    $this->travel(5)->minutes();
    $this->travel(5)->hours();
    $this->travel(5)->days();
    $this->travel(5)->weeks();
    $this->travel(5)->years();

    // 設定過去的時間...
    $this->travel(-5)->hours();

    // 設定一個確切的時間...
    $this->travelTo(now()->subHours(6));

    // 返回現在的時間...
    $this->travelBack();
}
```

你還可以為各種設定時間方法寫一個閉包。閉包將在指定的時間被凍結呼叫。一旦閉包執行完畢，時間將恢復正常:

```
$this->travel(5)->days(function () {
    // 在 5 天之後測試...
});

$this->travelTo(now()->subDays(10), function () {
    // 在指定的時間測試...
});
```

freezeTime 方法可用於凍結當前時間。與之類似地，freezeSecond 方法也可以秒為單位凍結當前時間：

```
use Illuminate\Support\Carbon;

// 凍結時間並在完成後恢復正常時間...
$this->freezeTime(function (Carbon $time) {
    // ...
});

// 凍結以秒為單位的時間並在完成後恢復正常時間...
$this->freezeSecond(function (Carbon $time) {
    // ...
```

```
})
```

正如你期望的一樣，上面討論的所有方法都主要用於測試對時間敏感的應用程式的行為，比如鎖定論壇上不活躍的帖子:

```
use App\Models\Thread;

public function test_forum_threads_lock_after_one_week_of_inactivity()
{
     $thread = Thread::factory()->create();

     $this->travel(1)->week();

     $this->assertTrue($thread->isLockedByInactivity());
}
```

60 Envoy 部署工具

60.1 簡介

Laravel Envoy 是一套在遠端伺服器上執行日常任務的工具。 使用 Blade 風格語法，你可以輕鬆地組態部署任務、Artisan 命令的執行等。目前， Envoy 僅支援 Mac 和 Linux 作業系統。但是， Windows 上可以使用 WSL2 來實現支援。

60.2 安裝

首先，運行 Composer 將 Envoy 安裝到你的項目中：

```
composer require laravel/envoy --dev
```

安裝 Envoy 之後， Envoy 的可執行檔案將出現在項目的 `vendor/bin` 目錄下：

```
php vendor/bin/envoy
```

60.3 編寫任務

60.3.1 定義任務

任務是 Envoy 的基礎建構元素。任務定義了你想在遠端伺服器上當任務被呼叫時所執行的 Shell 命令。例如，你定義了一個任務：在佇列伺服器上執行 `php artisan queue:restart` 命令。

你所有的 Envoy 任務都應該在項目根目錄中的 `Envoy.blade.php` 檔案中定義。下面是一個幫助你入門的例子：

```
@servers(['web' => ['user@192.168.1.1'], 'workers' => ['user@192.168.1.2']])

@task('restart-queues', ['on' => 'workers'])
    cd /home/user/example.com
    php artisan queue:restart
@endtask
```

正如你所見，在檔案頂部定義了一個 `@servers` 陣列，使你可以通過任務聲明的 `on` 選項引用這些伺服器。`@servers` 聲明應始終放置在單行中。在你的 `@task` 聲明中，你應該放置任務被呼叫執行時你期望在伺服器上運行的 Shell 命令。

60.3.1.1 本地任務

你可以通過將伺服器的 IP 地址指定為 `127.0.0.1` 來強制指令碼在本地運行：

```
@servers(['localhost' => '127.0.0.1'])
```

60.3.1.2 匯入 Envoy 任務

使用 `@import` 指令，你可以從其他的 Envoy 檔案匯入它們的故事與任務並新增到你的檔案中。匯入檔案後，你可以像定義在自己的 Envoy 檔案中一樣執行其中包含的任務：

```
@import('vendor/package/Envoy.blade.php')
```

60.3.2 多伺服器

Envoy 允許你輕鬆地在多台伺服器上運行任務。 首先，在 `@servers` 聲明中新增其他伺服器。每台伺服器都應分配一個唯一的名稱。一旦定義了其他伺服器，你可以在任務的 `on` 陣列中列出每個伺服器：

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2']])
     cd /home/user/example.com
     git pull origin {{ $branch }}
     php artisan migrate --force
@endtask
```

60.3.2.1 平行執行

默認情況下，任務將在每個伺服器上序列執行。 換句話說，任務將在第一台伺服器上完成運行後，再繼續在第二台伺服器上執行。如果你想在多個伺服器上平行運行任務，請在任務聲明中新增 `parallel` 選項：

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2'], 'parallel' => true])
     cd /home/user/example.com
     git pull origin {{ $branch }}
     php artisan migrate --force
@endtask
```

如你所見，檔案頂部定義了一個 `@server` 陣列，允許你在任務聲明的 `on` 選項中引用這些伺服器。`@server` 聲明應始終放在一行中。在你的 `@task` 聲明中，你應該放置任務被呼叫執行時你期望在伺服器上運行的 Shell 命令。

60.3.2.2 本地任務

你可以通過將伺服器的 IP 地址指定為 `127.0.0.1` 來強制指令碼在本地運行：

```
@servers(['localhost' => '127.0.0.1'])
```

60.3.2.3 匯入 Envoy 任務

使用 `@import` 指令，你可以從其他的 Envoy 檔案匯入它們的故事與任務並新增到你的檔案中。檔案匯入後，你可以執行他們所定義的任務，就像這些任務是在你的 Envoy 檔案中被定義的一樣：

```
@import('vendor/package/Envoy.blade.php')
```

60.3.3 組態

有時，你可能需要在執行 Envoy 任務之前執行一些 PHP 程式碼。你可以使用 `@setup` 指令來定義一段 PHP 程式碼塊，在任務執行之前執行這段程式碼：

```
@setup
     $now = new DateTime;
@endsetup
```

如果你需要在任務執行之前引入其他的 PHP 檔案，你可以在 `Envoy.blade.php` 檔案的頂部使用 `@include` 指令：

```
@include('vendor/autoload.php')

@task('restart-queues')
     # ...
@endtask
```

60.3.4 變數

如果需要的話，你可以在呼叫 Envoy 任務時通過在命令列中指定參數，將參數傳遞給 Envoy 任務：

```
php vendor/bin/envoy run deploy --branch=master
```

你可以使用 Blade 的「echo」語法訪問傳入任務中的命令列參數。你也可以在任務中使用 `if` 語句和循環。 例如，在執行 `git pull` 命令之前，我們先驗證 `$branch` 變數是否存在：

```
@servers(['web' => ['user@192.168.1.1']])

@task('deploy', ['on' => 'web'])
    cd /home/user/example.com

    @if ($branch)
        git pull origin {{ $branch }}
    @endif

    php artisan migrate --force
@endtask
```

60.3.5 故事

通過「故事」功能，你可以給一組任務起個名字，以便後續呼叫。例如，一個 `deploy` 故事可以運行 `update-code` 和 `install-dependencies` 等定義好的任務：

```
@servers(['web' => ['user@192.168.1.1']])

@story('deploy')
    update-code
    install-dependencies
@endstory

@task('update-code')
    cd /home/user/example.com
    git pull origin master
@endtask

@task('install-dependencies')
    cd /home/user/example.com
    composer install
@endtask
```

一旦編寫好了故事，你可以像呼叫任務一樣呼叫它：

```
php vendor/bin/envoy run deploy
```

60.3.6 任務鉤子

當任務和故事指令碼執行階段，會執行很多鉤子。Envoy 支援的鉤子類型有`@before`, `@after`, `@error`, `@success`, and `@finished`。 這些鉤子中的所有程式碼都被解釋為 PHP 並在本地執行，而不是在你的任務與之互動的遠端伺服器上執行。

你可以根據需要定義任意數量的這些。這些鉤子將按照它們在您的 Envoy 指令碼中出現的順序執行。

60.3.6.1 `@before`

在每個任務執行之前，Envoy 指令碼中註冊的所有 `@before` 鉤子都會執行。 `@before` 鉤子負責接收將要執行的任務的名稱：

```
@before
    if ($task === 'deploy') {
```

```
        // ...
    }
@endbefore
```

60.3.6.2 @after

每次任務執行後，Envoy 指令碼中註冊的所有 `@after` 鉤子都會執行。 `@after` 鉤子負責接收已執行任務的名稱：

```
@after
    if ($task === 'deploy') {
        // ...
    }
@endafter
```

60.3.6.3 @error

在每次任務失敗後（以大於 `0` 的狀態碼退出執行），Envoy 指令碼中註冊的所有 `@error` 鉤子都將執行。 `@error` 鉤子負責接收已執行任務的名稱：

```
@error
    if ($task === 'deploy') {
        // ...
    }
@enderror
```

60.3.6.4 @success

如果所有任務都已正確執行，則 Envoy 指令碼中註冊的所有 `@success` 鉤子都將執行：

```
@success
    // ...
@endsuccess
```

60.3.6.5 @finished

在所有任務都執行完畢後（不管退出狀態如何），所有的 `@finished` 鉤子都會被執行。 `@finished` 鉤子負責接收已完成任務的狀態碼，它可能是 `null` 或大於或等於 `0` 的 `integer`：

```
@finished
    if ($exitCode > 0) {
        // There were errors in one of the tasks...
    }
@endfinished
```

60.3.7 鉤子

當任務和故事執行階段，會執行許多鉤子。 Envoy 支援的鉤子類型有 `@before`、`@after`、`@error`、`@success` 和 `@finished`。這些鉤子中的所有程式碼都被解釋為 PHP 並在本地執行，而不是在與你的任務互動的遠端伺服器上執行。

你可以根據需要定義任意數量的鉤子。 它們將按照它們在你的 Envoy 指令碼中出現的順序執行。

60.3.7.1 @before

在每個任務執行之前，將執行在你的 Envoy 指令碼中註冊的所有 `@before` 鉤子。 `@before` 鉤子接收將要執行的任務的名稱：

```
@before
    if ($task === 'deploy') {
```

```
        // ...
    }
@endbefore
```

60.3.7.2 `@after`

每次任務執行後，將執行在你的 Envoy 指令碼中註冊的所有 `@after` 鉤子。 `@after` 鉤子接收已執行任務的名稱：

```
@after
    if ($task === 'deploy') {
        // ...
    }
@endafter
```

60.3.7.3 `@error`

在每個任務失敗後（退出時的狀態大於 `0`），在你的 Envoy 指令碼中註冊的所有 `@error` 鉤子都將執行。 `@error` 鉤子接收已執行任務的名稱：

```
@error
    if ($task === 'deploy') {
        // ...
    }
@enderror
```

60.3.7.4 `@success`

如果所有任務都沒有出現錯誤，那麼在你的 Envoy 指令碼中註冊的所有 `@success` 鉤子都將執行：

```
@success
    // ...
@endsuccess
```

60.3.7.5 `@finished`

在執行完所有任務後（無論退出狀態如何），所有的 `@finished` 鉤子都將被執行。 `@finished` 鉤子接收已完成任務的狀態程式碼，它可以是 `null` 或大於或等於 `0` 的 `integer`：

```
@finished
    if ($exitCode > 0) {
        // There were errors in one of the tasks...
    }
@endfinished
```

60.4 運行任務

要運行在應用程式的 `Envoy.blade.php` 檔案中定義的任務或 story，請執行 Envoy 的 `run` 命令，傳遞你想要執行的任務或 story 的名稱。Envoy 將執行該任務，並在任務執行階段顯示來自遠端伺服器的輸出：

```
php vendor/bin/envoy run deploy
```

60.4.1 確認任務執行

如果你想在在伺服器上運行給定任務之前進行確認，請將 `confirm` 指令新增到您的任務聲明中。此選項特別適用於破壞性操作：

```
@task('deploy', ['on' => 'web', 'confirm' => true])
    cd /home/user/example.com
    git pull origin {{ $branch }}
```

```
    php artisan migrate
@endtask
```

60.5 通知

60.5.1 Slack

Envoy 支援在每次任務執行後向 Slack 傳送通知。`@slack` 指令接受一個 Slack 鉤子 URL 和一個頻道/使用者名稱稱。您可以通過在 Slack 控制面板中建立 「Incoming WebHooks」 整合來檢索您的 webhook URL。

你應該將整個 webhook URL 作為傳遞給 `@slack` 指令的第一個參數。`@slack` 指令給出的第二個參數應該是頻道名稱 (`#channel`) 或使用者名稱稱 (`@user`)：

```
@finished
    @slack('webhook-url', '#bots')
@endfinished
```

默認情況下，Envoy 通知將向通知頻道傳送一條消息，描述已執行的任務。但是，你可以通過將第三個參數傳遞給 `@slack` 指令來覆蓋此消息，以自訂自己的消息：

```
@finished
    @slack('webhook-url', '#bots', 'Hello, Slack.')
@endfinished
```

60.5.2 Discord

Envoy 還支援在每個任務執行後向 Discord 傳送通知。`@discord` 指令接受一個 Discord Webhook URL 和一條消息。您可以在伺服器設定中建立「Webhook」，並選擇 Webhook 應該發佈到哪個頻道來檢索 Webhook URL。您應該將整個 Webhook URL 傳遞到 `@discord` 指令中：

```
@finished
    @discord('discord-webhook-url')
@endfinished
```

60.5.3 Telegram

Envoy 還支援在每個任務執行後向 Telegram 傳送通知。`@telegram` 指令接受一個 Telegram Bot ID 和一個 Chat ID。你可以使用 BotFather 建立一個新的機器人來檢索 Bot ID。你可以使用 @username_to_id_bot 檢索有效的 Chat ID。你應該將整個 Bot ID 和 Chat ID 傳遞到 `@telegram` 指令中：

```
@finished
    @telegram('bot-id','chat-id')
@endfinished
```

60.5.4 Microsoft Teams

Envoy 還支援在每個任務執行後向 Microsoft Teams 傳送通知。`@microsoftTeams` 指令接受 Teams Webhook（必填）、消息、主題顏色（成功、資訊、警告、錯誤）和一組選項。你可以通過建立新的 incoming webhook 來檢索 Teams Webhook。Teams API 有許多其他屬性可以自訂消息框，例如標題、摘要和部分。你可以在 Microsoft Teams 文件 中找到更多資訊。你應該將整個 Webhook URL 傳遞到 `@microsoftTeams` 指令中：

```
@finished
    @microsoftTeams('webhook-url')
@endfinished
```

61 Fortify 與前端無關的身份認證後端實現

61.1 介紹

Laravel Fortify 是一個在 Laravel 中與前端無關的身份認證後端實現。 Fortify 註冊了所有實現 Laravel 身份驗證功能所需的路由和 controller , 包括登錄、註冊、重設密碼、郵件驗證等。安裝 Fortify 後，你可以運行 Artisan 命令 `route:list` 來查看 Fortify 已註冊的路由。

由於 Fortify 不提供其自己的使用者介面，因此它應該與你自己的使用者介面配對，該使用者介面向其註冊的路由發出請求。在本文件的其餘部分中，我們將進一步討論如何向這些路由發出請求。

> **提示**
> 請記住， Fortify 是一個軟體包，旨在使你能夠快速開始實現 Laravel 的身份驗證功能。 **你並非一定要使用它**。 你始終可以按照以下說明中提供的文件，自由地與 Laravel 的身份認證服務進行互動， 使用者認證， 重設密碼 和 信箱認證 文件。

61.1.1 Fortify 是什麼？

如上所述，Laravel Fortify 是一個與前端無關的身份認證後端實現，Fortify 註冊了所有實現 Laravel 身份驗證功能所需的路由和 controller ，包括登錄，註冊，重設密碼，郵件認證等。

你不必使用 Fortify，也可以使用 Laravel 的身份認證功能。 你始終可以按照 使用者認證，重設密碼 和 信箱認證 文件中提供的文件來手動與 Laravel 的身份驗證服務進行互動。

如果你是一名新手，在使用 Laravel Fortify 之前不妨嘗試使用 Laravel Breeze 應用入門套件。Laravel Breeze 為你的應用提供身份認證支架，其中包括使用 Tailwind CSS。與 Fortify 不同，Breeze 將其路由和 controller 直接發佈到你的應用程式中。這使你可以學習並熟悉 Laravel 的身份認證功能，然後再允許 Laravel Fortify 為你實現這些功能。

Laravel Fortify 本質上是採用了 Laravel Breeze 的路由和 controller ，且提供了不包含使用者介面的擴展。這樣，你可以快速搭建應用程式身份認證層的後端實現，而不必依賴於任何特定的前端實現。

61.1.2 何時使用 Fortify？

你可能想知道何時使用 Laravel Fortify。首先，如果你正在使用 Laravel 的 應用入門套件，你不需要安裝 Laravel Fortify，因為它已經提供了完整的身份認證實現。

如果你不使用應用入門套件，並且你的應用需要身份認證功能，則有兩個選擇：手動實現應用的身份認證功能或使用由 Laravel Fortify 提供這些功能的後端實現。

如果你選擇安裝 Fortify，你的使用者介面將向 Fortify 的身份驗證路由發出請求，本文件中對此進行了詳細介紹，以便對使用者進行身份認證和註冊。

如果你選擇手動與 Laravel 的身份認證服務進行互動而不是使用 Fortify，可以按照 使用者認證，重設密碼 和 信箱認證 文件中提供的說明進行操作。

61.1.2.1 Laravel Fortify & Laravel Sanctum

一些開發人員對 Laravel Sanctum 和 Laravel Fortify 兩者之間的區別感到困惑。由於這兩個軟體包解決了兩個不同但相關的問題，因此 Laravel Fortify 和 Laravel Sanctum 並非互斥或競爭的軟體包。

Laravel Sanctum 只關心管理 API 令牌和使用 session cookie 或令牌來認證現有使用者。Sanctum 不提供任何處理使用者註冊，重設密碼等相關的路由。

如果你嘗試為提供 API 或用作單頁應用的後端的應用手動建構身份認證層，那麼完全有可能同時使用 Laravel Fortify（用於使用者註冊，重設密碼等）和 Laravel Sanctum（API 令牌管理， session 身份認證）。

61.2 安裝

首先，使用 Composer 軟體包管理器安裝 Fortify：

```
composer require laravel/fortify
```

下一步，使用 `vendor:publish` 命令來發佈 Fortify 的資源：

```
php artisan vendor:publish --provider="Laravel\Fortify\FortifyServiceProvider"
```

該命令會將 Fortify 的行為類發佈到你的 `app/Actions` 目錄，如果該目錄不存在，則會建立該目錄。此外，還將發佈 Fortify 的組態（`FortifyServiceProvider`）和遷移檔案。

下一步，你應該遷移資料庫：

```
php artisan migrate
```

61.2.1 Fortify 服務提供商

上面討論的 `vendor:publish` 命令還將發佈 `App\Providers\FortifyServiceProvider` 類。你應該確保該類已在應用程式的 `config/app.php` 組態檔案的 `providers` 陣列中註冊。

Fortify 服務提供商註冊了 Fortify 所發佈的行為類，並指導 Fortify 在執行各自的任務時使用它們。

61.2.2 Fortify 包含的功能

該 `fortify` 組態檔案包含一個 `features` 組態陣列。該陣列默路定義了 Fortify 的路由和功能。如果你不打算將 Fortify 與 Laravel Jetstream 配合使用，我們建議你僅啟用以下功能，這是大多數 Laravel 應用提供的基本身份認證功能：

```
'features' => [
    Features::registration(),
    Features::resetPasswords(),
    Features::emailVerification(),
],
```

61.2.3 停用檢視

默認情況下，Fortify 定義用於返回檢視的路由，例如登錄或註冊。但是，如果要建構 JavaScript 驅動的單頁應用，那麼可能不需要這些路由。因此，你可以通過將 `config/fortify.php` 組態檔案中的 `views` 組態值設為 `false` 來停用這些路由：

```
'views' => false,
```

61.2.3.1 停用檢視 & 重設密碼

如果你選擇停用 Fortify 的檢視，並且將為你的應用實現重設密碼功能，這時你仍然需要定義一個名為 `password.reset` 的路由，該路由負責顯示應用的「重設密碼」檢視。這是必要的，因為 Laravel 的 `Illuminate\Auth\Notifications\ResetPassword` 通知將通過名為 `password.reset` 的路由生成重設密碼 URL。

61.3 身份認證

首先，我們需要指導 Fortify 如何返回「登錄」檢視。記住，Fortify 是一個無頭認證擴展。如果你想要一個已經為你完成的 Laravel 身份認證功能的前端實現，你應該使用 應用入門套件。

所有的身份認證檢視邏輯，都可以使用 `Laravel\Fortify\Fortify` 類提供的方法來自訂。通常，你應該從應用的 `App\Providers\FortifyServiceProvider` 的 `boot` 方法中呼叫此方法。Fortify 將負責定義返回此檢視的 `/login` 路由：

```
use Laravel\Fortify\Fortify;

/**
 * 引導任何應用服務。
 */
public function boot(): void
{
    Fortify::loginView(function () {
        return view('auth.login');
    });

    // ...
}
```

你的登錄範本應包括一個向 `/login` 發出 POST 請求的表單。 `/login` 表單需要一個 `email` / `username` 和 `password`。 `email` / `username` 欄位與 `config/fortify.php` 組態檔案中的 `username` 值相匹配。另外，可以提供布林值 `remember` 欄位來指導使用者想要使用 Laravel 提供的「記住我」功能。

如果登錄嘗試成功，Fortify 會將你重新導向到通過應用程式 `fortify` 組態檔案中的 `home` 組態選項組態的 URI。如果登錄請求是 XHR 請求，將返回 200 HTTP 響應。

如果請求不成功，使用者將被重新導向回登錄頁，驗證錯誤將通過共享的 `$errors` Blade 範本變數 提供給你。或者，在 XHR 請求的情況下，驗證錯誤將與 422 HTTP 響應一起返回。

61.3.1 自訂使用者認證

Fortify 將根據提供的憑據和為你的應用程式組態的身份驗證保護自動檢索和驗證使用者。但是，你有時可能希望對登錄憑據的身份驗證和使用者的檢索方式進行完全自訂。幸運的是，Fortify 允許你使用 `Fortify::authenticateUsing` 方法輕鬆完成此操作。

此方法接受接收傳入 HTTP 請求的閉包。閉包負責驗證附加到請求的登錄憑據並返回關聯的使用者實例。如果憑據無效或找不到使用者，則閉包應返回 `null` 或 `false` 。通常，這個方法應該從你的 `FortifyServiceProvider` 的 `boot` 方法中呼叫：

```
use App\Models\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Laravel\Fortify\Fortify;

/**
 * 引導應用服務
```

```
 */
public function boot(): void
{
    Fortify::authenticateUsing(function (Request $request) {
        $user = User::where('email', $request->email)->first();

        if ($user &&
            Hash::check($request->password, $user->password)) {
            return $user;
        }
    });

    // ...
}
```

61.3.1.1 身份驗證看守器

你可以在應用程式的 `fortify` 檔案中自訂 Fortify 使用的身份驗證看守器。但是，你應該確保組態的看守器是 `Illuminate\Contracts\Auth\StatefulGuard` 的實現。如果你嘗試使用 Laravel Fortify 對 SPA 進行身份驗證，你應該將 Laravel 的默認 `web` 防護與 Laravel Sanctum 結合使用。

61.3.2 自訂身份驗證管道

Laravel Fortify 通過可呼叫類的管道對登錄請求進行身份驗證。如果你願意，你可以定義一個自訂的類管道，登錄請求應該通過管道傳輸。每個類都應該有一個 `__invoke` 方法，該方法接收傳入 `Illuminate\Http\Request` 實例的方法，並且像 中介軟體 一樣，呼叫一個 `$next` 變數，以便將請求傳遞給管道中的下一個類。

要定義自訂管道，可以使用 `Fortify::authenticateThrough` 方法。此方法接受一個閉包，該閉包應返回類陣列，以通過管道傳遞登錄請求。通常，應該從 `App\Providers\FortifyServiceProvider` 的 `boot` 方法呼叫此方法。

下面的示例包含默認管道定義，你可以在自己進行修改時將其用作開始：

```
use Laravel\Fortify\Actions\AttemptToAuthenticate;
use Laravel\Fortify\Actions\EnsureLoginIsNotThrottled;
use Laravel\Fortify\Actions\PrepareAuthenticatedSession;
use Laravel\Fortify\Actions\RedirectIfTwoFactorAuthenticatable;
use Laravel\Fortify\Fortify;
use Illuminate\Http\Request;

Fortify::authenticateThrough(function (Request $request) {
    return array_filter([
            config('fortify.limiters.login') ? null : EnsureLoginIsNotThrottled::class,
            Features::enabled(Features::twoFactorAuthentication()) ?
RedirectIfTwoFactorAuthenticatable::class : null,
            AttemptToAuthenticate::class,
            PrepareAuthenticatedSession::class,
    ]);
});
```

61.3.3 自訂跳轉

如果登錄嘗試成功，Fortify 會將你重新導向到你應用程式 `Fortify` 的組態檔案中的 `home` 組態選項的 URI 值。如果登錄請求是 XHR 請求，將返回 200 HTTP 響應。使用者註銷應用程式後，該使用者將被重新導向到 `/` 地址。

如果需要對這種行為進行高級定製，可以將 `LoginResponse` 和 `LogoutResponse` 契約的實現繫結到 Laravel 服務容器 。通常，這應該在你應用程式的 `App\Providers\FortifyServiceProvider` 類的 `register`

方法中完成：

```
use Laravel\Fortify\Contracts\LogoutResponse;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

/**
 * 註冊任何應用程式服務。
 */
public function register(): void
{
    $this->app->instance(LogoutResponse::class, new class implements LogoutResponse {
        public function toResponse(Request $request): RedirectResponse
        {
            return redirect('/');
        }
    });
}
```

61.4 雙因素認證

當 Fortify 的雙因素身份驗證功能啟用時，使用者需要在身份驗證過程中輸入一個六位數的數字令牌。該令牌使用基於時間的一次性密碼（TOTP）生成，該密碼可以從任何與 TOTP 相容的移動認證應用程式（如 Google Authenticator）中檢索。

在開始之前，你應該首先確保應用程式的 `App\Models\User` 模型使用 `Laravel\Fortify\TwoFactorAuthenticatable` trait：

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Laravel\Fortify\TwoFactorAuthenticatable;

class User extends Authenticatable
{
    use Notifiable, TwoFactorAuthenticatable;
}
```

接下來，你應該在應用程式中建構一個頁面，使用者可以在其中管理他們的雙因素身份驗證設定。該頁面應允許使用者啟用和停用雙因素身份驗證，以及重新生成雙因素身份驗證恢復的程式碼。

> 默認情況下， `fortify` 組態檔案的 `features` 陣列管理著 Fortify 的雙因素身份驗證設定在修改前需要密碼確認。因此，在使用之前，你的應用程式應該實現 Fortify 的 密碼確認 功能。

61.4.1 啟用雙因素身份驗證

要啟用雙重身份驗證，你的應用程式應向 Fortify 定義的 `/user/two-factor-authentication` 發出 POST 請求。如果請求成功，使用者將被重新導向回之前的 URL，並且 `status` session 變數將設定為 `two-factor-authentication-enabled`。你可以在範本中檢測這個 `status` session 變數以顯示適當的成功消息。如果請求是 XHR 請求，將返回 `200` HTTP 響應：

在選擇啟用雙因素認證後，使用者仍然必須通過提供一個有效的雙因素認證程式碼來「確認」他們的雙因素認證組態。因此，你的「成功」消息應該指示使用者，雙因素認證的確認仍然是必需的。

```
@if (session('status') == 'two-factor-authentication-enabled')
    <div class="mb-4 font-medium text-sm">
        Please finish configuring two factor authentication below.
```

```
    </div>
@endif
```

接下來，你應該顯示雙重身份驗證二維碼，供使用者掃描到他們的身份驗證器應用程式中。如果你使用 Blade 呈現應用程式的前端，則可以使用使用者實例上可用的 `twoFactorQrCodeSvg` 方法檢索二維碼 SVG：

```
$request->user()->twoFactorQrCodeSvg();
```

如果你正在建構由 JavaScript 驅動的前端，你可以向 `/user/two-factor-qr-code` 發出 XHR GET 請求以檢索使用者的雙重身份驗證二維碼。將返回一個包含 `svg` 鍵的 JSON 對象。

61.4.1.1 確認雙因素認證

除了顯示使用者的雙因素認證 QR 碼，你應該提供一個文字輸入，使用者可以提供一個有效的認證碼來「確認」他們的雙因素認證組態。這個程式碼應該通過 POST 請求提供到 `/user/confirmed-two-factor-authentication`，由 Fortify 來進行確認。

If the request is successful, the user will be redirected back to the previous URL and the `status` session variable will be set to `two-factor-authentication-confirmed`:

如果請求成功，使用者將被重新導向到之前的 URL，`status` session 變數將被設定為 `two-factor-authentication-confirmed'。

```
@if (session('status') == 'two-factor-authentication-confirmed')
    <div class="mb-4 font-medium text-sm">
        Two factor authentication confirmed and enabled successfully.
    </div>
@endif
```

如果對雙因素認證確認端點的請求是通過 XHR 請求進行的，將返回一個 `200` HTTP 響應。

61.4.1.2 顯示恢復程式碼

你還應該顯示使用者的兩個因素恢復程式碼。這些恢復程式碼允許使用者在無法訪問其移動裝置時進行身份驗證。如果你使用 Blade 來渲染應用程式的前端，你可以通過經過身份驗證的使用者實例訪問恢復程式碼：

```
(array) $request->user()->recoveryCodes()
```

如果你正在建構一個 JavaScript 驅動的前端，你可以向 `/user/two-factor-recovery-codes` 端點發出 XHR GET 請求。此端點將返回一個包含使用者恢復程式碼的 JSON 陣列。

要重新生成使用者的恢復程式碼，你的應用程式應向 `/user/two-factor-recovery-codes` 端點發出 POST 請求。

61.4.2 使用雙因素身份驗證進行身份驗證

在身份驗證過程中，Fortify 將自動將使用者重新導向到你的應用程式的雙因素身份驗證檢查頁面。但是，如果你的應用程式正在發出 XHR 登錄請求，則在成功進行身份驗證嘗試後返回的 JSON 響應將包含一個具有 `two_factor` 布林值屬性的 JSON 對象。你應該檢查此值以瞭解是否應該重新導向到應用程式的雙因素身份驗證檢查頁面。

要開始實現兩因素身份驗證功能，我們需要指示 Fortify 如何返回我們的雙因素身份驗證檢查頁面。Fortify 的所有身份驗證檢視渲染邏輯都可以使用通過 `Laravel\Fortify\Fortify` 類提供的適當方法進行自訂。通常，你應該從應用程式的 `App\Providers\FortifyServiceProvider` 類的 `boot` 方法呼叫此方法：

```
use Laravel\Fortify\Fortify;

/**
 * 引導任何應用程式服務。
 */
```

```
public function boot(): void
{
    Fortify::twoFactorChallengeView(function () {
        return view('auth.two-factor-challenge');
    });

    // ...
}
```

Fortify 將負責定義返回此檢視的 `/two-factor-challenge` 路由。你的 `two-factor-challenge` 範本應包含一個向 `/two-factor-challenge` 端點發出 POST 請求的表單。 `/two-factor-challenge` 操作需要包含有效 TOTP 令牌的 `code` 欄位或包含使用者恢復程式碼之一的 `recovery_code` 欄位。

如果登錄嘗試成功，Fortify 會將使用者重新導向到通過應用程式的 `fortify` 組態檔案中的 `home` 組態選項組態的 URI。如果登錄請求是 XHR 請求，將返回 204 HTTP 響應。

如果請求不成功，使用者將被重新導向回兩因素挑戰螢幕，驗證錯誤將通過共享的 `$errors` Blade 範本變數。或者，在 XHR 請求的情況下，驗證錯誤將返回 422 HTTP 響應。

61.4.3 停用兩因素身份驗證

要停用雙因素身份驗證，你的應用程式應向 `/user/two-factor-authentication` 端點發出 DELETE 請求。請記住，Fortify 的兩個因素身份驗證端點在被呼叫之前需要 密碼確認。

61.5 註冊

要開始實現我們應用程式的註冊功能，我們需要指示 Fortify 如何返回我們的“註冊”檢視。請記住，Fortify 是一個無頭身份驗證庫。如果你想要一個已經為你完成的 Laravel 身份驗證功能的前端實現，你應該使用 application starter kit。

Fortify 的所有檢視渲染邏輯都可以使用通過 `Laravel\Fortify\Fortify` 類提供的適當方法進行自訂。通常，你應該從 `App\Providers\FortifyServiceProvider` 類的 `boot` 方法呼叫此方法：

```
use Laravel\Fortify\Fortify;

/**
 * 引導任何應用程式服務。
 */
public function boot(): void
{
    Fortify::registerView(function () {
        return view('auth.register');
    });

    // ...
}
```

Fortify 將負責定義返回此檢視的 `/register` 路由。你的 `register` 範本應包含一個向 Fortify 定義的 `/register` 端點發出 POST 請求的表單。

`/register` 端點需要一個字串 `name`、字串電子郵件地址/使用者名稱、`password` 和 `password_confirmation` 欄位。電子郵件/使用者名稱欄位的名稱應與應用程式的 `fortify` 組態檔案中定義的 `username` 組態值匹配。

如果註冊嘗試成功，Fortify 會將使用者重新導向到通過應用程式的 `fortify` 組態檔案中的 `home` 組態選項組態的 URI。如果登錄請求是 XHR 請求，將返回 201 HTTP 響應。

如果請求不成功，使用者將被重新導向回註冊螢幕，驗證錯誤將通過共享的 `$errors` Blade 範本變數。或者，

在 XHR 請求的情況下，驗證錯誤將返回 422 HTTP 響應。

61.5.1 定製註冊

可以通過修改安裝 Laravel Fortify 時生成的 `App\Actions\Fortify\CreateNewUser` 操作來自訂使用者驗證和建立過程。

61.6 重設密碼

61.6.1 請求密碼重設連結

要開始實現我們應用程式的密碼重設功能，我們需要指示 Fortify 如何返回我們的“忘記密碼”檢視。請記住，Fortify 是一個無頭身份驗證庫。如果你想要一個已經為你完成的 Laravel 身份驗證功能的前端實現，你應該使用 application starter kit。

Fortify 的所有檢視渲染邏輯都可以使用通過 `Laravel\Fortify\Fortify` 類提供的適當方法進行自訂。通常，你應該從應用程式的 `App\Providers\FortifyServiceProvider` 類的 `boot` 方法呼叫此方法：

```
use Laravel\Fortify\Fortify;

/**
 * 引導任何應用程式服務。
 */
public function boot(): void
{
    Fortify::requestPasswordResetLinkView(function () {
        return view('auth.forgot-password');
    });

    // ...
}
```

Fortify 將負責定義返回此檢視的 `/forgot-password` 端點。你的 `forgot-password` 範本應該包含一個向 `/forgot-password` 端點發出 POST 請求的表單。

`/forgot-password` 端點需要一個字串 `email` 欄位。此欄位/資料庫列的名稱應與應用程式的 `fortify` 組態檔案中的 `email` 組態值匹配。

61.6.1.1 處理密碼重設連結請求響應

如果密碼重設連結請求成功，Fortify 會將使用者重新導向回 `/forgot-password` 端點，並向使用者傳送一封電子郵件，其中包含可用於重設密碼的安全連結。如果請求是 XHR 請求，將返回 200 HTTP 響應。

在請求成功後被重新導向到 `/forgot-password` 端點，`status` session 變數可用於顯示密碼重設連結請求的狀態。

在成功請求後被重新導向回 `/forgot-password` 端點後，`status` session 變數可用於顯示密碼重設連結請求嘗試的狀態。此 session 變數的值將匹配應用程式的 `password` 語言檔案 中定義的翻譯字串之一：

```
@if (session('status'))
    <div class="mb-4 font-medium text-sm text-green-600">
        {{ session('status') }}
    </div>
@endif
```

如果請求不成功，使用者將被重新導向回請求密碼重設連結螢幕，驗證錯誤將通過共享的 `$errors` Blade 範本

變數 提供給你。或者，在 XHR 請求的情況下，驗證錯誤將返回 422 HTTP 響應。

61.6.2 重設密碼

為了完成應用程式的密碼重設功能，我們需要指示 Fortify 如何返回我們的「重設密碼」檢視。

Fortify 的所有檢視渲染邏輯都可以使用通過 `Laravel\Fortify\Fortify` 類提供的適當方法進行自訂。通常，你應該從應用程式的 `App\Providers\FortifyServiceProvider` 類的 `boot` 方法呼叫此方法：

```
use Laravel\Fortify\Fortify;
use Illuminate\Http\Request;

/**
 * 引導任何應用程式服務。
 */
public function boot(): void
{
    Fortify::resetPasswordView(function (Request $request) {
        return view('auth.reset-password', ['request' => $request]);
    });

    // ...
}
```

Fortify 將負責定義顯示此檢視的路線。你的 `reset-password` 範本應該包含一個向 `/reset-password` 發出 POST 請求的表單。

`/reset-password` 端點需要一個字串 `email` 欄位、一個 `password` 欄位、一個 `password_confirmation` 欄位和一個名為 `token` 的隱藏欄位，其中包含 `request()->route('token')`。 `email` 欄位/資料庫列的名稱應與應用程式的 `fortify` 組態檔案中定義的 `email` 組態值匹配。

61.6.2.1 處理密碼重設響應

如果密碼重設請求成功，Fortify 將重新導向回 `/login` 路由，以便使用者可以使用新密碼登錄。此外，還將設定一個 `status` session 變數，以便你可以在登錄螢幕上顯示重設的成功狀態：

```
@if (session('status'))
    <div class="mb-4 font-medium text-sm text-green-600">
        {{ session('status') }}
    </div>
@endif
```

如果請求是 XHR 請求，將返回 200 HTTP 響應。

如果請求不成功，使用者將被重新導向回重設密碼螢幕，驗證錯誤將通過共享的 `$errors` Blade 範本變數。或者，在 XHR 請求的情況下，驗證錯誤將返回 422 HTTP 響應。

61.6.3 自訂密碼重設

可以通過修改安裝 Laravel Fortify 時生成的 `App\Actions\ResetUserPassword` 操作來自訂密碼重設過程。

61.7 電子郵件驗證

註冊後，你可能希望使用者在繼續訪問你的應用程式之前驗證他們的電子郵件地址。要開始使用，請確保在 `fortify` 組態檔案的 `features` 陣列中啟用了 `emailVerification` 功能。接下來，你應該確保你的 `App\`

`Models\User` 類實現了 `Illuminate\Contracts\Auth\MustVerifyEmail` 介面。

完成這兩個設定步驟後，新註冊的使用者將收到一封電子郵件，提示他們驗證其電子郵件地址的所有權。但是，我們需要通知 Fortify 如何顯示電子郵件驗證螢幕，通知使用者他們需要點選電子郵件中的驗證連結。

Fortify 的所有檢視的渲染邏輯都可以使用通過 `Laravel\Fortify\Fortify` 類提供的適當方法進行自訂。通常，你應該從應用程式的 `App\Providers\FortifyServiceProvider` 類的 `boot` 方法呼叫此方法：

```
use Laravel\Fortify\Fortify;

/**
 * 引導所有應用程式服務。
 */
public function boot(): void
{
    Fortify::verifyEmailView(function () {
        return view('auth.verify-email');
    });

    // ...
}
```

當使用者被 Laravel 內建的 `verified` 中介軟體重新導向到 `/email/verify` 端點時，Fortify 將負責定義顯示此檢視的路由。

你的 `verify-email` 範本應包含一條資訊性消息，指示使用者點選傳送到其電子郵件地址的電子郵件驗證連結。

61.7.1.1 重新傳送電子郵件驗證連結

如果你願意，你可以在應用程式的 `verify-email` 範本中新增一個按鈕，該按鈕會觸發對 `/email/verification-notification` 端點的 POST 請求。當此端點收到請求時，將通過電子郵件將新的驗證電子郵件連結傳送給使用者，如果先前的驗證連結被意外刪除或丟失，則允許使用者獲取新的驗證連結。

如果重新傳送驗證連結電子郵件的請求成功，Fortify 將使用 `status` session 變數將使用者重新導向回 `/email/verify` 端點，允許你向使用者顯示資訊性消息，通知他們操作已完成成功的。如果請求是 XHR 請求，將返回 202 HTTP 響應：

```
@if (session('status') == 'verification-link-sent')
    <div class="mb-4 font-medium text-sm text-green-600">
        A new email verification link has been emailed to you!
    </div>
@endif
```

61.7.2 保護路由

要指定一個路由或一組路由要求使用者驗證他們的電子郵件地址，你應該將 Laravel 的內建 `verified` 中介軟體附加到該路由。該中介軟體在你的應用程式的 `App\Http\Kernel` 類中註冊：

```
Route::get('/dashboard', function () {
    // ...
})->middleware(['verified']);
```

61.8 確認密碼

在建構應用程式時，你可能偶爾會有一些操作需要使用者在執行操作之前確認其密碼。通常，這些路由受到 Laravel 內建的 `password.confirm` 中介軟體的保護。

要開始實現密碼確認功能，我們需要指示 Fortify 如何返回應用程式的「密碼確認」檢視。請記住，Fortify 是一

個無頭身份驗證庫。如果你想要一個已經為你完成的 Laravel 身份驗證功能的前端實現，你應該使用 application starter kit。

Fortify 的所有檢視渲染邏輯都可以使用通過 `Laravel\Fortify\Fortify` 類提供的適當方法進行自訂。通常，你應該從應用程式的 `App\Providers\FortifyServiceProvider` 類的 `boot` 方法呼叫此方法：

```
use Laravel\Fortify\Fortify;

/**
 * 引導所有應用程式服務。
 */
public function boot(): void
{
    Fortify::confirmPasswordView(function () {
        return view('auth.confirm-password');
    });

    // ...
}
```

Fortify 將負責定義返回此檢視的 `/user/confirm-password` 端點。你的 `confirm-password` 範本應包含一個表單，該表單向 `/user/confirm-password` 端點發出 POST 請求。 `/user/confirm-password` 端點需要一個包含使用者當前密碼的 `password` 欄位。

如果密碼與使用者的當前密碼匹配，Fortify 會將使用者重新導向到他們嘗試訪問的路由。如果請求是 XHR 請求，將返回 201 HTTP 響應。

如果請求不成功，使用者將被重新導向回確認密碼螢幕，驗證錯誤將通過共享的 `$errors` Blade 範本變數提供給你。或者，在 XHR 請求的情況下，驗證錯誤將返回 422 HTTP 響應。

62 Horizon 佇列管理工具

62.1 介紹

> **提示**
> 在深入瞭解 Laravel Horizon 之前，您應該熟悉 Laravel 的基礎 佇列服務。 Horizon 為 Laravel 的佇列增加了額外的功能，如果你還不熟悉 Laravel 提供的基本佇列功能，這些功能可能會讓人感到困惑。

Laravel Horizon 為你的 Laravel Redis queues.提供了一個美觀的儀表盤和程式碼驅動的組態。它可以方便的監控佇列系統的關鍵指標：任務吞吐量、執行階段間、作業失敗情況。

在使用 Horizon 時，所有佇列的 worker 組態都儲存在一個簡單的組態檔案中。通過在受版本控制的檔案中定義應用程式的 worker 組態，你可以在部署應用程式時輕鬆擴展或修改應用程式的佇列 worker。

62.2 安裝

> **注意** Laravel Horizon 要求你使用 Redis 來為你的佇列服務。因此，你應該確保在應用程式的 `config/queue.php` 組態檔案中將佇列連接設定為 `redis`。

你可以使用 Composer 將 Horizon 安裝到你的 Laravel 項目裡：

```
composer require laravel/horizon
```

Horizon 安裝之後，使用 `horizon:install` Artisan 命令發佈資源：

```
php artisan horizon:install
```

62.2.1 組態

Horizon 資源發佈之後，其主要組態檔案會被分配到 `config/horizon.php` 檔案。可以用這個組態檔案組態工作選項，每個組態選項包含一個用途描述，請務必仔細研究這個檔案。

> **注意：** Horizon 在內部使用名為 `horizon` 的 Redis 連接。此 Redis 連接名稱是保留的，不應分配給 `database.php` 組態檔案中的另一個 Redis 連接或作為 `horizon.php` 組態檔案中的 `use` 選項的值。

62.2.1.1 環境組態

安裝後，你需要熟悉的重點 Horizon 組態選項是 `environments` 組態選項。此組態選項定義了你的應用程式運行的一系列環境，並為每個環境定義了工作處理程序選項。默認情況下，此條目包含 **生產 (`production`)** 和 **本地 (`local`)**環境。簡而言之，你可以根據自己的需要自由新增更多環境：

```
'environments' => [
    'production' => [
        'supervisor-1' => [
            'maxProcesses' => 10,
            'balanceMaxShift' => 1,
            'balanceCooldown' => 3,
        ],
    ],

    'local' => [
```

```
        'supervisor-1' => [
            'maxProcesses' => 3,
        ],
    ],
],
```

當你啟動 Horizon 時，它將使用指定應用程式運行環境所組態的 worker 處理程序選項。通常，環境組態由 APP_ENV 環境變數 的值確定。例如，默認的 local Horizon 環境組態為啟動三個工作處理程序，並自動平衡分配給每個佇列的工作處理程序數量。默認的「生產」環境組態為最多啟動 10 個 worker 處理程序，並自動平衡分配給每個佇列的 worker 處理程序數量。

> **注意：**你應該確保你的 horizon 組態檔案的 environments 部分包含計畫在其上運行 Horizon 的每個 環境 的組態。

62.2.1.2 Supervisors

正如你在 Horizon 的默認組態檔案中看到的那樣。每個環境可以包含一個或多個 Supervisor 組態。默認情況下，組態檔案將這個 Supervisor 定義為 supervisor-1；但是，你可以隨意命名你的 Supervisor。每個 Supervisor 負責監督一組 worker，並負責平衡佇列之間的 worker。

如果你想定義一組在指定環境中運行的新 worker，可以向相應的環境新增額外的 Supervisor。如果你想為應用程式使用的特定佇列定義不同的平衡策略或 worker 數量，也可以選擇這樣做。

62.2.1.3 預設值

在 Horizon 的默認組態檔案中，你會注意到一個 defaults 組態選項。這個組態選項指定應用程式的 supervisors 的預設值。Supervisor 的默認組態值將合併到每個環境的 Supervisor 組態中，讓你在定義 Supervisor 時避免不必要的重複工作。

62.2.2 均衡策略

與 Laravel 的默認佇列系統不同，Horizon 允許你從三個平衡策略中進行選擇：simple， auto， 和 false。simple 策略是組態檔案的默認選項，它會在處理程序之間平均分配進入的任務：

```
'balance' => 'simple',
```

auto 策略根據佇列的當前工作負載來調整每個佇列的工作處理程序數量。舉個例子，如果你的 notifications 佇列有 1000 個等待的任務，而你的 render 佇列是空的，那麼 Horizon 將為 notifications 佇列分配更多的工作執行緒，直到佇列為空。

當使用 auto 策略時，你可以定義 minProcesses 和 maxProcesses 的組態選項來控制 Horizon 擴展處理程序的最小和最大數量：

```
'environments' => [
    'production' => [
        'supervisor-1' => [
            'connection' => 'redis',
            'queue' => ['default'],
            'balance' => 'auto',
            'autoScalingStrategy' => 'time',
            'minProcesses' => 1,
            'maxProcesses' => 10,
            'balanceMaxShift' => 1,
            'balanceCooldown' => 3,
            'tries' => 3,
        ],
    ],
```

```
],
```

`autoScalingStrategy` 組態值決定了 Horizon 是根據清除佇列所需的總時間（`time` 策略）還是根據佇列上的作業總數（`size` 策略）來為佇列分配更多的 Worker 處理程序。

`balanceMaxShift` 和 `balanceCooldown` 組態項可以確定 Horizon 將以多快的速度擴展處理程序，在上面的示例中，每 3 秒鐘最多建立或銷毀一個新處理程序，你可以根據應用程式的需要隨意調整這些值。

當 `balance` 選項設定為 `false` 時，將使用默認的 Laravel 行為，它按照佇列在組態中列出的順序處理佇列。

62.2.3 控制面板授權

Horizon 在 `/horizon` 上顯示了一個控制面板。默認情況下，你只能在 `local` 環境中訪問這個面板。在你的 `app/Providers/HorizonServiceProvider.php` 檔案中，有一個 授權攔截器（Gates） 的方法定義，該攔截器用於控制在**非本地**環境中對 Horizon 的訪問。末可以根據需要修改此方法，來限制對 Horizon 的訪問：

```
/**
 * 註冊 Horizon 授權
 *
 * 此方法決定了誰可以在非本地環境中訪問 Horizon
 */
protected function gate(): void
{
    Gate::define('viewHorizon', function (User $user) {
        return in_array($user->email, [
            'taylor@laravel.com',
        ]);
    });
}
```

62.2.3.1 可替代的身份驗證策略

需要留意的是，Laravel 會自動將經過認證的使用者注入到攔截器（Gate）閉包中。如果你的應用程式通過其他方法（例如 IP 限制）提供 Horizon 安全性保障，那麼你訪問 Horizon 使用者可能不需要實現這個「登錄」動作。因此，你需要將上面的 `function (User $user)` 更改為 `function (User $user = null)` 以強制 Laravel 跳過身份驗證。

62.2.4 靜默作業

有時，你可能對查看某些由你的應用程式或第三方軟體包發出的工作不感興趣。與其讓這些作業在你的「已完成作業」列表中佔用空間，你可以讓它們靜默。要開始的話，在你的應用程式的 `horizon` 組態檔案中的 `silenced` 組態選項中新增作業的類名。

```
'silenced' => [
    App\Jobs\ProcessPodcast::class,
],
```

或者，你希望靜默的作業可以實現 `Laravel\Horizon\Contracts\Silenced` 介面。如果一個作業實現了這個介面，它將自動被靜默，即使它不在 `silenced` 組態陣列中。

```
use Laravel\Horizon\Contracts\Silenced;

class ProcessPodcast implements ShouldQueue, Silenced
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    // ...
}
```

62.3 升級 Horizon

當你升級到 Horizon 的一個新的主要版本時，你需要仔細閱讀 升級指南。

此外，升級到新的 Horizon 版本時，你應該重新發佈 Horizon 資源：

```
php artisan horizon:publish
```

為了使資原始檔保持最新並避免以後的更新中出現問題，你可以將以下 `horizon:publish` 命令新增到 `composer.json` 檔案中的 `post-update-cmd` 指令碼中：

```
{
    "scripts": {
        "post-update-cmd": [
            "@php artisan vendor:publish --tag=laravel-assets --ansi --force"
        ]
    }
}
```

62.4 運行 Horizon

在 `config/horizon.php` 中組態了你的 workers 之後，你可以使用 `horizon` Artisan 命令啟動 Horizon。只需這一個命令你就可以啟動你的所有已組態的 workers：

```
php artisan horizon
```

你可以暫停 Horizon 處理程序，並使用 `horizon:pause` 和 `horizon:continue` Artisan 命令指示它繼續處理任務：

```
php artisan horizon:pause

php artisan horizon:continue
```

你還可以使用 `horizon:pause-supervisor` 和 `horizon:continue-supervisor` Artisan 命令暫停和繼續指定的 Horizon supervisors：

```
php artisan horizon:pause-supervisor supervisor-1

php artisan horizon:continue-supervisor supervisor-1
```

你可以使用 `horizon:status` Artisan 命令檢查 Horizon 處理程序的當前狀態：

```
php artisan horizon:status
```

你可以使用 `horizon:terminate` Artisan 命令優雅地終止機器上的主 Horizon 處理程序。Horizon 會等當前正在處理的所有任務都完成後退出：

```
php artisan horizon:terminate
```

62.4.1 部署 Horizon

如果要將 Horizon 部署到一個正在運行的伺服器上，應該組態一個處理程序監視器來監視 `php artisan horizon` 命令，並在它意外退出時重新啟動它。

在將新程式碼部署到伺服器時，你需要終止 Horizon 主處理程序，以便處理程序監視器重新啟動它並接收程式碼的更改。

```
php artisan horizon:terminate
```

62.4.1.1 安裝 Supervisor

Supervisor 是一個用於 Linux 作業系統的處理程序監視器。如果 `Horizon` 處理程序被退出或終止，Supervisor 將

自動重啟你的 `Horizon` 處理程序。如果要在 Ubuntu 上安裝 Supervisor，你可以使用以下命令。如果你不使用 Ubuntu，也可以使用作業系統的包管理器安裝 Supervisor：

```
sudo apt-get install supervisor
```

> **技巧：**如果你覺得自己組態 Supervisor 難如登天，可以考慮使用 Laravel Forge，它將自動為你的 Laravel 項目安裝和組態 Supervisor。

62.4.1.2 Supervisor 組態

Supervisor 組態檔案通常儲存在 `/etc/supervisor/conf.d` 目錄下。在此目錄中，你可以建立任意數量的組態檔案，這些組態檔案會告訴 supervisor 如何監視你的處理程序。例如，讓我們建立一個 `horizon.conf` 檔案，它啟動並監視一個 `horizon` 處理程序：

```
[program:horizon]
process_name=%(program_name)s
command=php /home/forge/example.com/artisan horizon
autostart=true
autorestart=true
user=forge
redirect_stderr=true
stdout_logfile=/home/forge/example.com/horizon.log
stopwaitsecs=3600
```

在定義 Supervisor 組態時，你應該確保 `stopwaitsecs` 的值大於最長運行作業所消耗的秒數。否則，Supervisor 可能會在作業處理完之前就將其殺死。

> **注意：**雖然上面的例子對基於 Ubuntu 的伺服器有效，但其他伺服器作業系統對監督員組態檔案的位置和副檔名可能有所不同。請查閱你的伺服器的文件以瞭解更多資訊。

62.4.1.3 啟動 Supervisor

建立了組態檔案後，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 horizon
```

> **技巧：**關於 Supervisor 的更多資訊，可以查閱 Supervisor 文件。

62.5 標記 (Tags)

Horizon 允許你將 `tags` 分配給任務，包括郵件、事件廣播、通知和排隊的事件監聽器。實際上，Horizon 會根據附加到作業上的有 Eloquent 模型，智能地、自動地標記大多數任務。例如，看看下面的任務：

```
<?php

namespace App\Jobs;

use App\Models\Video;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;

class RenderVideo implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;
```

```
    /**
     * 建立一個新的任務實例
     */
    public function __construct(
        public Video $video,
    ) {}

    /**
     * 執行任務
     */
    public function handle(): void
    {
        // ...
    }
}
```

如果此任務與 `App\Models\Video` 實例一起排隊，且該實例的 `id` 為 `1`，則該作業將自動接收 `App\Models\Video:1` 標記。這是因為 Horizon 將為任何有 Eloquent 的模型檢查任務的屬性。如果找到了有 Eloquent 的模型，Horizon 將智能地使用模型的類名和主鍵標記任務：

```
use App\Jobs\RenderVideo;
use App\Models\Video;

$video = Video::find(1);

RenderVideo::dispatch($video);
```

62.5.1.1 手動標記作業

如果你想手動定義你的一個佇列對象的標籤，你可以在類上定義一個 `tags` 方法：

```
class RenderVideo implements ShouldQueue
{
    /**
     * 獲取應該分配給任務的標記
     *
     * @return array<int, string>
     */
    public function tags(): array
    {
        return ['render', 'video:'.$this->video->id];
    }
}
```

62.6 通知

注意： 當組態 Horizon 傳送 Slack 或 SMS 通知時，你應該查看 相關通知驅動程式的先決條件。

如果你希望在一個佇列有較長的等待時間時得到通知，你可以使用 `Horizon::routeMailNotificationsTo`, `Horizon::routeSlackNotificationsTo`, 和 `Horizon::routeSmsNotificationsTo` 方法。你可以通過應用程式的 `App\Providers\HorizonServiceProvider` 中的 `boot` 方法來呼叫這些方法：

```
/**
 * 服務引導
 */
public function boot(): void
{
    parent::boot();

    Horizon::routeSmsNotificationsTo('15556667777');
```

```
    Horizon::routeMailNotificationsTo('example@example.com');
    Horizon::routeSlackNotificationsTo('slack-webhook-url', '#channel');
}
```

62.6.1.1 組態通知等待時間閾值

你可以在 `config/horizon.php` 的組態檔案中組態多少秒算是「長等待」。你可以用該檔案中的 `waits` 組態選項控制每個 連接 / 佇列 組合的長等待閾值：

```
'waits' => [
    'redis:default' => 60,
    'redis:critical,high' => 90,
],
```

62.7 指標

Horizon 有一個指標控制面板，它提供了任務和佇列的等待時間和吞吐量等資訊。要讓這些資訊顯示在這個控制面板上，你應該組態 Horizon 的 `snapshot` Artisan 命令，通過你的應用程式的 調度器 每五分鐘運行一次：

```
/**
 * 定義應用程式的命令調度
 */
protected function schedule(Schedule $schedule): void
{
    $schedule->command('horizon:snapshot')->everyFiveMinutes();
}
```

62.8 刪除失敗的作業

如果你想刪除失敗的作業，可以使用 `horizon:forget` 命令。 `horizon:forget` 命令接受失敗作業的 ID 或 UUID 作為其唯一參數：

```
php artisan horizon:forget 5
```

62.9 從佇列中清除作業

如果你想從應用程式的默認佇列中刪除所有作業，你可以使用 `horizon:clear` Artisan 命令執行此操作：

```
php artisan horizon:clear
```

你可以設定 `queue` 選項來從特定佇列中刪除作業：

```
php artisan horizon:clear --queue=emails
```

63 Octane 加速引擎

63.1 簡介

Laravel Octane 通過使用高性能應用程式伺服器為您的應用程式提供服務來增強您的應用程式的性能，包括 Open Swoole，Swoole，和 RoadRunner。Octane 啟動您的應用程式一次，將其保存在記憶體中，然後以極快的速度向它提供請求。

63.2 安裝

Octane 可以通過 Composer 包管理器安裝：

```
composer require laravel/octane
```

安裝 Octane 後，您可以執行 `octane:install` 命令，該命令會將 Octane 的組態檔案安裝到您的應用程式中：

```
php artisan octane:install
```

63.3 伺服器先決條件

> **注意** Laravel Octane 需要 PHP 8.1+.

63.3.1 RoadRunner

RoadRunner 由使用 Go 建構的 RoadRunner 二進制檔案提供支援。當您第一次啟動基於 RoadRunner 的 Octane 伺服器時，Octane 將為您提供下載和安裝 RoadRunner 二進制檔案。

63.3.1.1 通過 Laravel Sail 使用 RoadRunner

如果你打算使用 Laravel Sail 開發應用，你應該運行如下命令安裝 Octane 和 RoadRunner:

```
./vendor/bin/sail up

./vendor/bin/sail composer require laravel/octane spiral/roadrunner
```

接下來，你應該啟動一個 Sail Shell，並運行 `rr` 可執行檔案檢索基於 Linux 的最新版 RoadRunner 二進制檔案：

```
./vendor/bin/sail shell

# Within the Sail shell...
./vendor/bin/rr get-binary
```

安裝完 RoadRunner 二進制檔案後，你可以退出 Sail Shell session 。然後，需要調整 Sail 用來保持應用運行的 `supervisor.conf` 檔案。首先，請執行 `sail:publish` Artisan 命令：

```
./vendor/bin/sail artisan sail:publish
```

接著，更新應用 `docker/supervisord.confd` 檔案中的 `command` 指令，這樣 Sail 就可以使用 Octane 作為伺服器，而非 PHP 開發伺服器，運行服務了：

```
command=/usr/bin/php -d variables_order=EGPCS /var/www/html/artisan octane:start --
server=roadrunner --host=0.0.0.0 --rpc-port=6001 --port=80
```

最後，請確保 `rr` 二進制檔案是可執行的並重新建構 Sail 鏡像：

```
chmod +x ./rr

./vendor/bin/sail build --no-cache
```

63.3.2 Swoole

如果你打算使用 Swoole 伺服器來運行 Laravel Octane 應用，你必須安裝 Swoole PHP 元件。通常可以通過 PECL 安裝：

```
pecl install swoole
```

63.3.2.1 Open Swoole

如果你想要使用 Open Swoole 伺服器運行 Laravel Octane 應用，你必須安裝 Open Swoole PHP 擴展。通常可以通過 PECL 完成安裝：

```
pecl install openswoole
```

通過 Open Swoole 使用 Laravel Octane，可以獲得 Swoole 提供的相同功能，如並行任務，計時和間隔。

63.3.2.2 通過 Laravel Sail 使用 Swoole

> **注意** 在通過 Sail 提供 Octane 應用程式之前，請確保你使用的是最新版本的 Laravel Sail 並在應用程式的根目錄中執行 `./vendor/bin/sail build --no-cache`。

你可以使用 Laravel 的官方 Docker 開發環境 Laravel Sail 開發基於 Swoole 的 Octane 應用程式。 Laravel Sail 默認包含 Swoole 擴展。但是，你仍然需要調整 Sail 使用的 `supervisor.conf` 件以保持應用運行。首先，執行 `sail:publish` Artisan 命令：

```
./vendor/bin/sail artisan sail:publish
```

接下來，更新應用程式的 `docker/supervisord.conf` 檔案的 `command` 指令，使得 Sail 使用 Octane 替代 PHP 開發伺服器：

```
command=/usr/bin/php -d variables_order=EGPCS /var/www/html/artisan octane:start --
server=swoole --host=0.0.0.0 --port=80
```

最後，建構你的 Sail 鏡像：

```
./vendor/bin/sail build --no-cache
```

63.3.2.3 Swoole 組態

Swoole 支援一些額外的組態選項，如果需要，你可以將它們新增到你的 `octane` 組態檔案中。因為它們很少需要修改，所以這些選項不包含在默認組態檔案中：

```
'swoole' => [
    'options' => [
        'log_file' => storage_path('logs/swoole_http.log'),
        'package_max_length' => 10 * 1024 * 1024,
    ],
],
```

63.4 為應用程式提供服務

Octane 伺服器可以通過 `octane:start` Artisan 命令啟動。此命令將使用由應用程式的 `octane` 組態檔案的 `server` 組態選項指定的伺服器：

```
php artisan octane:start
```

默認情況下，Octane 將在 8000 連接埠上啟動伺服器（可組態），因此你可以在 Web 瀏覽器中通過

`http://localhost:8000` 訪問你的應用程式。

63.4.1 通過 HTTPS 為應用程式提供服務

默認情況下，通過 Octane 運行的應用程式會生成以 `http://` 為前綴的連結。當使用 HTTPS 時，可將在應用的 `config/octane.php` 組態檔案中使用的 `OCTANE_HTTPS` 環境變數設定為 `true`。當此組態值設定為 `true` 時，Octane 將指示 Laravel 在所有生成的連結前加上 `https://`：

```
'https' => env('OCTANE_HTTPS', false),
```

63.4.2 通過 Nginx 為應用提供服務

> **提示** 如果你還沒有準備好管理自己的伺服器組態，或者不習慣組態運行健壯的 Laravel Octane 應用所需的所有各種服務，請查看 Laravel Forge。

在生產環境中，你應該在傳統 Web 伺服器（例如 Nginx 或 Apache）之後為 Octane 應用提供服務。 這樣做將允許 Web 伺服器為你的靜態資源（例如圖片和樣式表）提供服務，並管理 SSL 證書。

在下面的 Nginx 組態示例檔案中，Nginx 將向在連接埠 8000 上運行的 Octane 伺服器提供站點的靜態資源和代理請求：

```
map $http_upgrade $connection_upgrade {
    default upgrade;
    ''      close;
}

server {
    listen 80;
    listen [::]:80;
    server_name domain.com;
    server_tokens off;
    root /home/forge/domain.com/public;

    index index.php;

    charset utf-8;

    location /index.php {
        try_files /not_exists @octane;
    }

    location / {
        try_files $uri $uri/ @octane;
    }

    location = /favicon.ico { access_log off; log_not_found off; }
    location = /robots.txt  { access_log off; log_not_found off; }

    access_log off;
    error_log  /var/log/nginx/domain.com-error.log error;

    error_page 404 /index.php;

    location @octane {
        set $suffix "";

        if ($uri = /index.php) {
            set $suffix ?$query_string;
        }

        proxy_http_version 1.1;
```

```
        proxy_set_header Host $http_host;
        proxy_set_header Scheme $scheme;
        proxy_set_header SERVER_PORT $server_port;
        proxy_set_header REMOTE_ADDR $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection $connection_upgrade;

        proxy_pass http://127.0.0.1:8000$suffix;
    }
}
```

63.4.3 監視檔案更改

由於 Octane 伺服器啟動時應用程式被載入到記憶體中一次，因此對應用程式檔案的任何更改都不會在您刷新瀏覽器時反映出來。例如，新增到 `routes/web.php` 檔案的路由定義在伺服器重新啟動之前不會反映出來。為了方便起見，你可以使用 `--watch` 標誌指示 Octane 在應用程式中的任何檔案更改時自動重新啟動伺服器：

```
php artisan octane:start --watch
```

在使用此功能之前，您應該確保在本地開發環境中安裝了 Node。此外，你還應該在項目中安裝 Chokidar 檔案監視庫：

```
npm install --save-dev chokidar
```

你可以使用應用程式的 `config/octane.php` 組態檔案中的 `watch` 組態選項來組態應該被監視的目錄和檔案。

63.4.4 指定工作處理程序數

默認情況下，Octane 會為機器提供的每個 CPU 核心啟動一個應用程式請求工作處理程序。這些工作處理程序將用於在進入應用程式時服務傳入的 HTTP 請求。你可以使用 `--workers` 選項手動指定要啟動的工作處理程序數量，當呼叫 `octane:start` 命令時：

```
php artisan octane:start --workers=4
```

如果你使用 Swoole 應用程式伺服器，則還可以指定要啟動的任務工作處理程序數量：

```
php artisan octane:start --workers=4 --task-workers=6
```

63.4.5 指定最大請求數量

為了防止記憶體洩漏，Octane 在處理完 500 個請求後會優雅地重新啟動任何 worker。要調整這個數字，你可以使用 `--max-requests` 選項：

```
php artisan octane:start --max-requests=250
```

63.4.6 多載 Workers

你可以使用 `octane:reload` 命令優雅地重新啟動 Octane 伺服器的應用 workers。通常，這應該在部署後完成，以便將新部署的程式碼載入到記憶體中並用於為後續請求提供服務：

```
php artisan octane:reload
```

63.4.7 停止伺服器

你可以使用 `octane:stop` Artisan 命令停止 Octane 伺服器：

```
php artisan octane:stop
```

63.4.7.1 檢查伺服器狀態

你可以使用 `octane:status` Artisan 命令檢查 Octane 伺服器的當前狀態：

```
php artisan octane:status
```

63.5 依賴注入和 Octane

由於 Octane 只啟動你的應用程式一次，並在服務請求時將其保留在記憶體中，所以在建構你的應用程式時，你應該考慮一些注意事項。例如，你的應用程式的服務提供者的 `register` 和 `boot` 方法將只在 request worker 最初啟動時執行一次。在隨後的請求中，將重用相同的應用程式實例。

鑑於這個機制，在將應用服務容器或請求注入任何對象的建構函式時應特別小心。這樣一來，該對像在隨後的請求中就可能有一個穩定版本的容器或請求。

Octane 會在兩次請求之間自動處理重設任何第一方框架的狀態。然而，Octane 並不總是知道如何重設由你的應用程式建立的全域狀態。因此，你應該知道如何以一種對 Octane 友好的方式來建構你的應用程式。下面，我們將討論在使用 Octane 時可能引起問題的最常見情況。

63.5.1 容器注入

通常來說，你應該避免將應用服務容器或 HTTP 請求實例注入到其他對象的建構函式中。例如，下面的繫結將整個應用服務容器注入到繫結為單例的對象中：

```
use App\Service;
use Illuminate\Contracts\Foundation\Application;

/**
 * Register any application services.
 */
public function register(): void
{
    $this->app->singleton(Service::class, function (Application $app) {
        return new Service($app);
    });
}
```

在這個例子中，如果在應用程式引導過程中解析 `Service` 實例，容器將被注入到該服務中，並且該容器將在後續的請求中保留。這對於你的特定應用程式**可能**不是一個問題，但是它可能會導致容器意外地缺少後來在引導過程中新增的繫結或後續請求中新增的繫結。

為瞭解決這個問題，你可以停止將繫結註冊為單例，或者你可以將一個容器解析器閉包注入到服務中，該閉包總是解析當前的容器實例：

```
use App\Service;
use Illuminate\Container\Container;
use Illuminate\Contracts\Foundation\Application;

$this->app->bind(Service::class, function (Application $app) {
    return new Service($app);
});

$this->app->singleton(Service::class, function () {
    return new Service(fn () => Container::getInstance());
});
```

全域的 `app` 輔助函數和 `Container::getInstance()` 方法將始終返回應用程式容器的最新版本。

63.5.2 請求注入

通常來說，你應該避免將應用服務容器或 HTTP 請求實例注入到其他對象的建構函式中。例如，下面的繫結將整個請求實例注入到繫結為單例的對象中：

```
use App\Service;
use Illuminate\Contracts\Foundation\Application;

/**
 * Register any application services.
 */
public function register(): void
{
    $this->app->singleton(Service::class, function (Application $app) {
        return new Service($app['request']);
    });
}
```

在這個例子中，如果在應用程式啟動過程中解析 `Service` 實例，則會將 HTTP 請求注入到服務中，並且相同的請求將由 `Service` 實例保持在後續請求中。因此，所有標頭、輸入和查詢字串資料以及所有其他請求資料都將不正確。

為瞭解決這個問題，你可以停止將繫結註冊為單例，或者你可以將請求解析器閉包注入到服務中，該閉包始終解析當前請求實例。或者，最推薦的方法是在執行階段將對象所需的特定請求資訊傳遞給對象的方法之一：

```
use App\Service;
use Illuminate\Contracts\Foundation\Application;

$this->app->bind(Service::class, function (Application $app) {
    return new Service($app['request']);
});

$this->app->singleton(Service::class, function (Application $app) {
    return new Service(fn () => $app['request']);
});

// Or...

$service->method($request->input('name'));
```

全域的 `request` 幫助函數將始終返回應用程式當前處理的請求，因此可以在應用程式中安全使用它。

> **警告** 在 controller 方法和路由閉包中類型提示 Illuminate 實例是可以接受的。

63.5.3 組態庫注入

一般來說，你應該避免將組態庫實例注入到其他對象的建構函式中。例如，以下繫結將組態庫注入到繫結為單例的對象中：

```
use App\Service;
use Illuminate\Contracts\Foundation\Application;

/**
 * Register any application services.
 */
public function register(): void
{
    $this->app->singleton(Service::class, function (Application $app) {
        return new Service($app->make('config'));
    });
}
```

在這個示例中，如果在請求之間的組態值更改了，那麼這個服務將無法訪問新的值，因為它依賴於原始儲存庫

實例。

作為解決方法，你可以停止將繫結註冊為單例，或者將組態儲存庫解析器閉包注入到類中：

```
use App\Service;
use Illuminate\Container\Container;
use Illuminate\Contracts\Foundation\Application;

$this->app->bind(Service::class, function (Application $app) {
    return new Service($app->make('config'));
});

$this->app->singleton(Service::class, function () {
    return new Service(fn () => Container::getInstance()->make('config'));
});
```

全域 `config` 將始終返回組態儲存庫的最新版本，因此在應用程式中使用是安全的。

63.5.4 管理記憶體洩漏

請記住，Octane 在請求之間保留應用程式，因此將資料新增到靜態維護的陣列中將導致記憶體洩漏。例如，以下 controller 具有記憶體洩漏，因為對應用程式的每個請求將繼續向靜態的 `$data` 陣列新增資料：

```
use App\Service;
use Illuminate\Http\Request;
use Illuminate\Support\Str;

/**
 * 處理傳入的請求。
 */
public function index(Request $request): array
{
    Service::$data[] = Str::random(10);

    return [
        // ...
    ];
}
```

在建構應用程式時，你應特別注意避免建立此類記憶體洩漏。建議在本地開發期間監視應用程式的記憶體使用情況，以確保您不會在應用程式中引入新的記憶體洩漏。

63.6 並行任務

> **警告** 此功能需要 Swoole。

當使用 Swoole 時，你可以通過輕量級的後台任務並行執行操作。你可以使用 Octane 的 `concurrently` 方法實現此目的。你可以將此方法與 PHP 陣列解構結合使用，以檢索每個操作的結果：

```
use App\User;
use App\Server;
use Laravel\Octane\Facades\Octane;

[$users, $servers] = Octane::concurrently([
    fn () => User::all(),
    fn () => Server::all(),
]);
```

由 Octane 處理的並行任務利用 Swoole 的 「task workers」 並在與傳入請求完全不同的處理程序中執行。可用於處理並行任務的工作程序的數量由 `octane:start` 命令的 `--task-workers` 指令確定：

```
php artisan octane:start --workers=4 --task-workers=6
```

在呼叫 `concurrently` 方法時，你應該不要提供超過 1024 個任務，因為 Swoole 任務系統強制執行此限制。

63.7 刻度和間隔

警告 此功能需要 Swoole.

當使用 Swoole 時，你可以註冊定期執行的 「tick」 操作。你可以通過 `tick` 方法註冊 「tick」 回呼函數。提供給 `tick` 方法的第一個參數應該是一個字串，表示定時器的名稱。第二個參數應該是在指定間隔內呼叫的可呼叫對象。

在此示例中，我們將註冊一個閉包，每 10 秒呼叫一次。通常，`tick` 方法應該在你應用程式的任何服務提供程序的 `boot` 方法中呼叫：

```
Octane::tick('simple-ticker', fn () => ray('Ticking...'))
         ->seconds(10);
```

使用 `immediate` 方法，你可以指示 Octane 在 Octane 伺服器初始啟動時立即呼叫 tick 回呼，並在 N 秒後每次呼叫：

```
Octane::tick('simple-ticker', fn () => ray('Ticking...'))
         ->seconds(10)
         ->immediate();
```

63.8 Octane 快取

警告 此功能需要 Swoole.

使用 Swoole 時，你可以利用 Octane 快取驅動程式，該驅動程式提供每秒高達 200 萬次的讀寫速度。因此，這個快取驅動程式是需要從快取層中獲得極高讀寫速度的應用程式的絕佳選擇。

該快取驅動程式由 Swoole tables 驅動。快取中的所有資料可供伺服器上的所有工作處理程序訪問。但是，當伺服器重新啟動時，快取資料將被清除：

```
Cache::store('octane')->put('framework', 'Laravel', 30);
```

注意 Octane 快取中允許的最大條目數可以在您的應用程式的 octane 組態檔案中定義。

63.8.1 快取間隔

除了 Laravel 快取系統提供的典型方法外，Octane 快取驅動程式還提供了基於間隔的快取。這些快取會在指定的間隔自動刷新，並應在一個應用程式服務提供程序的 `boot` 方法中註冊。例如，以下快取將每五秒刷新一次：

```
use Illuminate\Support\Str;

Cache::store('octane')->interval('random', function () {
    return Str::random(10);
}, seconds: 5);
```

63.9 表格

警告 此功能需要 Swoole.

使用 Swoole 時，你可以定義和與自己的任意 Swoole tables 進行互動。Swoole tables 提供極高的性能吞吐量，並且可以通過伺服器上的所有工作處理程序訪問其中的資料。但是，當它們內部的資料在伺服器重新啟動時將被丟

失。

表在應用 octane 組態檔案 tables 陣列組態中設定。最大運行 1000 行的示例表已經組態。像下面這樣，字串行支援的最大長度在列類型後面設定：

```
'tables' => [
    'example:1000' => [
        'name' => 'string:1000',
        'votes' => 'int',
    ],
],
```

通過 Octane::table 方法訪問表：

```
use Laravel\Octane\Facades\Octane;

Octane::table('example')->set('uuid', [
    'name' => 'Nuno Maduro',
    'votes' => 1000,
]);

return Octane::table('example')->get('uuid');
```

注意 Swoole table 支援的列類型有： string ，int 和 float 。

64 Passport OAuth 認證

64.1 簡介

Laravel Passport 可以在幾分鐘之內為你的應用程式提供完整的 OAuth2 伺服器端實現。Passport 是基於由 Andy Millington 和 Simon Hamp 維護的 League OAuth2 server 建立的。

> **注意**
> 本文件假定你已熟悉 OAuth2 。如果你並不瞭解 OAuth2 ，閱讀之前請先熟悉下 OAuth2 的 常用術語 和特性。

64.1.1 Passport 還是 Sanctum?

在開始之前，我們希望你先確認下是 Laravel Passport 還是 Laravel Sanctum 能為你的應用提供更好的服務。如果你的應用確確實實需要支援 OAuth2，那沒疑問，你需要選用 Laravel Passport。

然而，如果你只是試圖要去認證一個單頁應用，或者手機應用，或者發佈 API 令牌，你應該選用 Laravel Sanctum。 Laravel Sanctum 不支援 OAuth2，它提供了更為簡單的 API 授權開發體驗。

64.2 安裝

在開始使用之前，使用 Composer 包管理器安裝 Passport：

```
composer require laravel/passport
```

Passport 的 服務提供器 註冊了自己的資料庫遷移指令碼目錄， 所以你應該在安裝軟體包完成後遷移你自己的資料庫。 Passport 的遷移指令碼將為你的應用建立用於儲存 OAuth2 客戶端和訪問令牌的資料表：

```
php artisan migrate
```

接下來，你需要執行 Artisan 命令 `passport:install`。這個命令將會建立一個用於生成安全訪問令牌的加密秘鑰。另外，這個命令也將建立用於生成訪問令牌的 「個人訪問」 客戶端和 「密碼授權」 客戶端 ：

```
php artisan passport:install
```

> **技巧**
> 如果你想用使用 UUID 作為 Passport `Client` 模型的主鍵，代替默認的自動增長整形欄位，請在安裝 Passport 時使用 uuids 參數 。

在執行 `passport:install` 命令後， 新增 `Laravel\Passport\HasApiTokens` trait 到你的 `App\Models\User` 模型中。 這個 trait 會提供一些幫助方法用於檢查已認證使用者的令牌和權限範圍。如果你的模型已經在使用 `Laravel\Sanctum\HasApiTokens` trait，你可以刪除該 trait：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Laravel\Passport\HasApiTokens;

class User extends Authenticatable
{
```

```
    use HasApiTokens, HasFactory, Notifiable;
}
```

最後，在您的應用的 config/auth.php 組態檔案中，您應當定義一個 api 的授權看守器，並且將其 driver 選項設定為 passport 。這個調整將會讓您的應用程式使用 Passport 的 TokenGuard 來鑑權 API 介面請求：

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],

    'api' => [
        'driver' => 'passport',
        'provider' => 'users',
    ],
],
```

64.2.1.1 客戶端 UUID

您也可以在運行 passport:install 命令的時候使用 --uuids 選項。這個參數將會讓 Passport 使用 UUID 來替代默認的自增長形式的 Passport Client 模型主鍵。在您運行帶有 --uuids 參數的 passport:install 命令後，您將得到關於停用 Passport 默認遷移的相關指令說明：

```
php artisan passport:install --uuids
```

64.2.2 部署 Passport

在您第一次部署 Passport 到您的應用伺服器時，您需要執行 passport:keys 命令。該命令用於生成 Passport 用於生成 access token 的一個加密金鑰。生成的加密金鑰不應到新增到原始碼控制系統中：

```
php artisan passport:keys
```

如有必要，您可以定義 Passport 的金鑰應當載入的位置。您可以使用 Passport:loadKeysFrom 方法來實現。通常，這個方法應當在您的 App\Providers\AuthServiceProvider 類的 boot 方法中呼叫：

```
/**
 * Register any authentication / authorization services.
 */
public function boot(): void
{
    Passport::loadKeysFrom(__DIR__.'/../secrets/oauth');
}
```

64.2.2.1 從環境中載入金鑰

此外，您可以使用 vendor:publish Artisan 命令來發佈您的 Passport 組態檔案：

```
php artisan vendor:publish --tag=passport-config
```

在發佈組態檔案之後，您可以將加密金鑰組態為環境變數，再載入它們：

```
PASSPORT_PRIVATE_KEY="-----BEGIN RSA PRIVATE KEY-----
<private key here>
-----END RSA PRIVATE KEY-----"

PASSPORT_PUBLIC_KEY="-----BEGIN PUBLIC KEY-----
<public key here>
-----END PUBLIC KEY-----"
```

64.2.3 自訂遷移

如果您不打算使用 Passport 的默認遷移，您應當在 `App\Providers\AppServiceProvider` 類的 `register` 方法中呼叫 `Passport::ignoreMigrations` 方法。您可以 使用 `vendor:publish` Artisan 命令來匯出默認的遷移檔案：

```
php artisan vendor:publish --tag=passport-migrations
```

64.2.4 Passport 的升級

當升級到 Passport 的主要版本時，請務必查閱 升級指南.

64.3 組態

64.3.1 客戶端金鑰的 Hash 加密

如果您希望客戶端金鑰在儲存到資料庫時使用 Hash 對其進行加密，您應當在 `App\Provider\AuthServiceProvider` 類的 `boot` 方法中呼叫 `Passport:hashClientSecrets` ：

```
use Laravel\Passport\Passport;

Passport::hashClientSecrets();
```

一旦啟用後，所有的客戶端金鑰都將只在建立的時候顯示。由於明文的客戶端金鑰沒有儲存到資料庫中，因此一旦其丟失後便無法恢復。

64.3.2 Token 生命週期

默認情況下，Passport 會頒髮長達一年的長期 token 。如果您想要組態一個更長或更短的 token 生命週期，您可以在 `App\Provider\AuthServiceProvider` 類的 `boot` 方法中呼叫 `tokensExpiresIn` 、`refresgTokensExpireIn` 和 `personalAccessTokensExpireIn` 方法：

```
/**
 * 註冊身份驗證/授權服務。
 */
public function boot(): void
{
    Passport::tokensExpireIn(now()->addDays(15));
    Passport::refreshTokensExpireIn(now()->addDays(30));
    Passport::personalAccessTokensExpireIn(now()->addMonths(6));
}
```

> **注意**
> Passport 資料庫表中的 `expires_at` 列是唯讀的，僅僅用於顯示。在頒發 token 的時候，Passport 將過期資訊儲存在已簽名和加密的 token 中。如果你想讓 token 失效，你應當 撤銷它 。

64.3.3 重寫 Passport 的默認模型

您可以通過定義自己的模型並繼承相應的 Passport 模型來實現自由自由擴展 Passport 內部使用的模型：

```
use Laravel\Passport\Client as PassportClient;

class Client extends PassportClient
{
```

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

在定義您的模型之後，您可以在 `Laravel\Passport\Passport` 類中指定 Passport 使用您自訂的模型。一樣的，您應該在應用程式的 `App\Providers\AuthServiceProvider` 類中的 `boot` 方法中指定 Passport 使用您自訂的模型：

```
use App\Models\Passport\AuthCode;
use App\Models\Passport\Client;
use App\Models\Passport\PersonalAccessClient;
use App\Models\Passport\RefreshToken;
use App\Models\Passport\Token;

/**
 * 註冊任意認證/授權服務。
 */
public function boot(): void
{
    Passport::useTokenModel(Token::class);
    Passport::useRefreshTokenModel(RefreshToken::class);
    Passport::useAuthCodeModel(AuthCode::class);
    Passport::useClientModel(Client::class);
    Passport::usePersonalAccessClientModel(PersonalAccessClient::class);
}
```

64.3.4 重寫路由

您可能希望自訂 Passport 定義的路由。要實現這個功能，第一步，您需要在應用程式的 `AppServiceProvider` 中的 `register` 方法中新增 `Passport:ignoreRoutes` 語句，以忽略由 Passport 註冊的路由：

```
use Laravel\Passport\Passport;

/**
 * 註冊任意的應用程式服務。
 */
public function register(): void
{
    Passport::ignoreRoutes();
}
```

然後，您可以複製 Passport 在自己的檔案中 定義的路由到應用程式的 `routes/web.php` 檔案中，並且將其修改為您喜歡的任何形式：

```
Route::group([
    'as' => 'passport.',
    'prefix' => config('passport.path', 'oauth'),
    'namespace' => 'Laravel\Passport\Http\Controllers',
], function () {
    // Passport 路由……
});
```

64.4 發佈訪問令牌

通過授權碼使用 OAuth2 是大多數開發人員熟悉的方式。使用授權碼方式時，客戶端應用程式會將使用者重新導向到你的伺服器，在那裡他們會批准或拒絕向客戶端發出訪問令牌的請求。

64.4.1 客戶端管理

首先，開發者如果想要搭建一個與你的伺服器端介面互動的應用端，需要在伺服器端這邊註冊一個「客戶

端」。通常，這需要開發者提供應用程式的名稱和一個 URL，在應用軟體的使用者授權請求後，應用程式會被重新導向到該 URL。

64.4.1.1 `passport:client` 命令

使用 Artisan 命令 `passport:client` 是一種最簡單的建立客戶端的方式。 這個命令可以建立你自己私有的客戶端，用於 Oauth2 功能測試。 當你執行 `client` 命令後， Passport 將會給你更多關於客戶端的提示，以及生成的客戶端 ID

```
php artisan passport:client
```

多重新導向 URL 地址的設定

如果你想為你的客戶端提供多個重新導向 URL ，你可以在執行 `Passport:client` 命令出現提示輸入 URL 地址的時候，輸入用逗號分割的多個 URL 。任何包含逗號的 URL 都需要先執行 URL 轉碼：

```
http://example.com/callback,http://examplefoo.com/callback
```

64.4.1.2 JSON API

因為應用程式的開發者是無法使用 `client` 命令的，所以 Passport 提供了 JSON 格式的 API ，用於建立客戶端。這解決了你還要去手動建立 controller 程式碼（程式碼用於新增，更新，刪除客戶端）的麻煩。

但是，你需要結合 Passport 的 JSON API 介面和你的前端面板管理頁面， 為你的使用者提供客戶端管理功能。接下里，我們會回顧所有用於管理客戶端的的 API 介面。方便起見，我們使用 Axios 模擬對端點的 HTTP 請求。

這些 JSON API 介面被 `web` 和 `auth` 兩個中介軟體保護著，因此，你只能從你的應用中呼叫。 外部來源的呼叫是被禁止的。

64.4.1.3 `GET /oauth/clients`

下面的路由將為授權使用者返回所有的客戶端。最主要的作用是列出所有的使用者客戶端，接下來就可以編輯或刪除它們了：

```
axios.get('/oauth/clients')
    .then(response => {
        console.log(response.data);
    });
```

64.4.1.4 `POST /oauth/clients`

下面的路由用於建立新的客戶端。 它需要兩個參數： **客戶端名稱**和**重新導向 URL** 地址。 **重新導向 URL** 地址是使用者在授權或者拒絕授權後被重新導向到的地方。

客戶端被建立後，將會生成客戶端 ID 和客戶端秘鑰。 這對值用於從你的應用獲取訪問令牌。 呼叫下面的客戶端建立路由將建立新的客戶端實例：

```
const data = {
    name: 'Client Name',
    redirect: 'http://example.com/callback'
};

axios.post('/oauth/clients', data)
    .then(response => {
        console.log(response.data);
    })
    .catch (response => {
        // 列出響應的錯誤...
    });
```

64.4.1.5 PUT /oauth/clients/{client-id}

下面的路由用來更新客戶端。它需要兩個參數： 客戶端名稱和重新導向 URL 地址。 重新導向 URL 地址是使用者在授權或者拒絕授權後被重新導向到的地方。路由將返回更新後的客戶端實例：

```
const data = {
    name: 'New Client Name',
    redirect: 'http://example.com/callback'
};

axios.put('/oauth/clients/' + clientId, data)
    .then(response => {
        console.log(response.data);
    })
    .catch (response => {
        // 列出響應的錯誤...
    });
```

64.4.1.6 DELETE /oauth/clients/{client-id}

下面的路由用於刪除客戶端：

```
axios.delete('/oauth/clients/' + clientId)
    .then(response => {
        // ...
    });
```

64.4.2 請求令牌

64.4.2.1 授權重新導向

客戶端建立好後，開發者使用 client ID 和秘鑰向你的應用伺服器傳送請求，以便獲取授權碼和訪問令牌。 首先，接收到請求的業務端伺服器會重新導向到你應用的 `/oauth/authorize` 路由上，如下所示：

```
use Illuminate\Http\Request;
use Illuminate\Support\Str;

Route::get('/redirect', function (Request $request) {
    $request->session()->put('state', $state = Str::random(40));

    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://third-party-app.com/callback',
        'response_type' => 'code',
        'scope' => '',
        'state' => $state,
        // 'prompt' => '', // "none", "consent", or "login"
    ]);

    return redirect('http://passport-app.test/oauth/authorize?'.$query);
});
```

`prompt` 參數可用於指定 Passport 應用程式的認證行為。

如果 `prompt` 值為 `none`，如果使用者還沒有通過 Passport 應用程式的認證，Passport 將總是拋出一個認證錯誤。如果值是 `同意`，Passport 將總是顯示授權批准螢幕，即使所有的範疇以前都被授予消費應用程式。如果值是 `login`，Passport 應用程式將總是提示使用者重新登錄到應用程式，即使他們已經有一個現有的 session 。

如果沒有提供 `prompt` 值，只有當使用者以前沒有授權訪問所請求範圍的消費應用程式時，才會提示使用者進行授權。

技巧：請記住，/oauth/authorize 路由默認已經在 Passport::route 方法中定義，你無需手動定義它。

64.4.2.2 請求認證

當接收到一個請求後， Passport 會自動展示一個範本頁面給使用者，使用者可以選擇授權或者拒絕授權。如果請求被認證，使用者將被重新導向到之前業務伺服器設定的 redirect_uri 上去。 這個 redirect_uri 就是客戶端在建立時提供的重新導向地址參數。

如果你想自訂授權頁面，你可以先使用 Artisan 命令 vendor:publish 發佈 Passport 的檢視頁面。 被發佈的檢視頁面位於 resources/views/vendor/passport 路徑下：

```
php artisan vendor:publish --tag=passport-views
```

有時，你可能希望跳過授權提示，比如在授權第一梯隊客戶端的時候。你可以通過 繼承 Client 模型並實現 skipsAuthorization 方法。如果 skipsAuthorization 方法返回 true， 客戶端就會直接被認證並立即重新導向到設定的重新導向地址：

```
<?php

namespace App\Models\Passport;

use Laravel\Passport\Client as BaseClient;

class Client extends BaseClient
{
    /**
     * 確定客戶端是否應跳過授權提示。
     */
    public function skipsAuthorization(): bool
    {
        return $this->firstParty();
    }
}
```

64.4.2.3 授權碼到授權令牌的轉化

如果使用者授權了訪問，他們會被重新導向到業務伺服器端。首先，業務端服務需要檢查 state 參數是否和重新導向之前儲存的值一致。 如果 state 參數的值正確，業務端伺服器需要對你的應用發起獲取 access token 的 POST 請求。 請求需要攜帶有授權碼，授權碼就是之前使用者授權後由你的應用伺服器生成的碼：

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Http;

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

    throw_unless(
        strlen($state) > 0 && $state === $request->state,
        InvalidArgumentException::class
    );

    $response = Http::asForm()->post('http://passport-app.test/oauth/token', [
        'grant_type' => 'authorization_code',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'redirect_uri' => 'http://third-party-app.com/callback',
        'code' => $request->code,
    ]);

    return $response->json();
});
```

呼叫路由 `/oauth/token` 將返回一串 json 字串，包含了 `access_token`, `refresh_token` 和 `expires_in` 屬性。`expires_in` 屬性的值是 access_token 剩餘的有效時間。

> **技巧：**就和 `/oauth/authorize` 路由一樣， `/oauth/token` 路由已經在 `Passport::routes` 方法中定義，你無需再自訂這個路由。

64.4.2.4 JSON API

Passport 同樣包含了一個 JSON API 介面用來管理授權訪問令牌。你可以使用該介面為使用者搭建一個管理訪問令牌的控制面板。方便來著，我們將使用 Axios 模擬 HTTP 對端點發起請求。由於 JSON API 被中介軟體 `web` 和 `auth` 保護著，我們只能在應用內部呼叫。

64.4.2.5 `GET /oauth/tokens`

下面的路由包含了授權使用者建立的所有授權訪問令牌。介面的主要作用是列出使用者所有可撤銷的令牌：

```
axios.get('/oauth/tokens')
     .then(response => {
         console.log(response.data);
     });
```

64.4.2.6 `DELETE /oauth/tokens/{token-id}`

下面的路由用於撤銷授權訪問令牌以及相關的刷新令牌：

```
axios.delete('/oauth/tokens/' + tokenId);
```

64.4.3 刷新令牌

如果你的應用發佈的是短生命週期訪問令牌，使用者需要使用刷新令牌來延長訪問令牌的生命週期，刷新令牌是在生成訪問令牌時同時生成的：

```
use Illuminate\Support\Facades\Http;

$response = Http::asForm()->post('http://passport-app.test/oauth/token', [
     'grant_type' => 'refresh_token',
     'refresh_token' => 'the-refresh-token',
     'client_id' => 'client-id',
     'client_secret' => 'client-secret',
     'scope' => '',
]);

return $response->json();
```

呼叫路由 `/oauth/token` 將返回一串 json 字串，包含了 `access_token`, `refresh_token` 和 `expires_in` 屬性。`expires_in` 屬性的值是 access_token 剩餘的有效時間。

64.4.4 撤銷令牌

你可以使用 `Laravel\Passport\TokenRepository` 類的 `revokeAccessToken` 方法撤銷令牌。你可以使用 `Laravel\Passport\RefreshTokenRepository` 類的 `revokeRefreshTokensByAccessTokenId` 方法撤銷刷新令牌。這兩個類可以通過 Laravel 的服務容器得到：

```
use Laravel\Passport\TokenRepository;
use Laravel\Passport\RefreshTokenRepository;

$tokenRepository = app(TokenRepository::class);
```

```
$refreshTokenRepository = app(RefreshTokenRepository::class);

// 撤銷一個訪問令牌...
$tokenRepository->revokeAccessToken($tokenId);

// 撤銷該令牌的所有刷新令牌...
$refreshTokenRepository->revokeRefreshTokensByAccessTokenId($tokenId);
```

64.4.5 清除令牌

如果令牌已經被撤銷或者已經過期了，你可能希望把它們從資料庫中清理掉。Passport 提供了 Artisan 命令 `passport:purge` 幫助你實現這個操作:

```
# 清除已經撤銷或者過期的令牌以及授權碼...
php artisan passport:purge

# 只清理過期 6 小時的令牌以及授權碼...
php artisan passport:purge --hours=6

# 只清理撤銷的令牌以及授權碼...
php artisan passport:purge --revoked

# 只清理過期的令牌以及授權碼...
php artisan passport:purge --expired
```

你可以在應用的 `App\Console\Kernel` 類中組態一個定時任務，每天自動的清理令牌：

```
/**
 * Define the application's command schedule.
 */
protected function schedule(Schedule $schedule): void
{
    $schedule->command('passport:purge')->hourly();
}
```

64.5 通過 PKCE 發佈授權碼

通過 PKCE 「 Proof Key for Code Exchange, 中文譯為 程式碼交換的證明金鑰」 發放授權碼是對單頁面應用或原生應用進行認證以便訪問 API 介面的安全方式。這種發放授權碼是用於不能保證客戶端密碼被安全儲存，或為降低攻擊者攔截授權碼的威脅。在這種模式下，當授權碼獲取令牌時，用 「驗證碼」(code verifier) 和 「質疑碼」（ code challenge, challenge, 名詞可譯為：挑戰；異議；質疑等）的組合來交換客戶端訪問金鑰。

64.5.1 建立客戶端

在使用 PKCE 方式發佈令牌之前，你需要先建立一個啟用了 PKCE 的客戶端。你可以使用 Artisan 命令 `passport:client` 並帶上 `--public` 參數來完成該操作：

```
php artisan passport:client --public
```

64.5.2 請求令牌

64.5.2.1 驗證碼（Code Verifier ）和質疑碼（Code Challenge）

這種授權方式不提供授權秘鑰，開發者需要建立一個驗證碼和質疑碼的組合來請求得到一個令牌。

驗證碼是一串包含 43 位到 128 位字元的隨機字串。可用字元包括字母，數字以及下面這些字元：`"-"`, `"."`, `"_"`, `"~"`，可參考 RFC 7636 specification 定義。

質疑碼是一串 Base64 編碼包含 URL 和檔案名稱安全字元的字串，字串結尾的 '=' 號需要刪除，並且不能包含分行符號，空白符或其他附加字元。

```
$encoded = base64_encode(hash('sha256', $code_verifier, true));

$codeChallenge = strtr(rtrim($encoded, '='), '+/', '-_');
```

64.5.2.2 授權重新導向

客戶端建立完後，你可以使用客戶端 ID 以及生成的驗證碼，質疑碼從你的應用請求獲取授權碼和訪問令牌。首先，業務端應用需要向伺服器端路由 /oauth/authorize 發起重新導向請求：

```
use Illuminate\Http\Request;
use Illuminate\Support\Str;

Route::get('/redirect', function (Request $request) {
    $request->session()->put('state', $state = Str::random(40));

    $request->session()->put(
        'code_verifier', $code_verifier = Str::random(128)
    );

    $codeChallenge = strtr(rtrim(
        base64_encode(hash('sha256', $code_verifier, true))
    , '='), '+/', '-_');

    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://third-party-app.com/callback',
        'response_type' => 'code',
        'scope' => '',
        'state' => $state,
        'code_challenge' => $codeChallenge,
        'code_challenge_method' => 'S256',
        // 'prompt' => '', // "none", "consent", or "login"
    ]);

    return redirect('http://passport-app.test/oauth/authorize?'.$query);
});
```

64.5.2.3 驗證碼到訪問令牌的轉換

使用者授權訪問後，將重新導向到業務端服務。正如標準授權定義那樣，業務端需要驗證回傳的 state 參數的值和在重新導向之前設定的值是否一致。

如果 state 的值驗證通過，業務接入端需要嚮應用端發起一個獲取訪問令牌的 POST 請求。請求的參數需要包括之前使用者授權通過後你的應用生成的授權碼，以及之前生成的驗證碼：

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Http;

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

    $codeVerifier = $request->session()->pull('code_verifier');

    throw_unless(
        strlen($state) > 0 && $state === $request->state,
        InvalidArgumentException::class
    );

    $response = Http::asForm()->post('http://passport-app.test/oauth/token', [
        'grant_type' => 'authorization_code',
        'client_id' => 'client-id',
```

```
        'redirect_uri' => 'http://third-party-app.com/callback',
        'code_verifier' => $codeVerifier,
        'code' => $request->code,
    ]);

    return $response->json();
});
```

64.6 密碼授權方式的令牌

> **注意**
> 我們不再建議使用密碼授予令牌。相反，你應該選擇 OAuth2 伺服器當前推薦的授權類型 。

OAuth2 的密碼授權方式允許你自己的客戶端（比如手機端應用），通過使用信箱 / 使用者名稱和密碼獲取訪問秘鑰。這樣你就可以安全的為自己發放令牌，而不需要完整地走 OAuth2 的重新導向授權訪問流程。

64.6.1 建立密碼授權方式客戶端

在你使用密碼授權方式發佈令牌前，你需要先建立密碼授權方式的客戶端。你可以通過 Artisan 命令 `passport:client` ， 並加上 `--password` 參數來建立這樣的客戶端。 **如果你已經運行過 `passport:install` 命令，則不需要再運行下面的命令:**

```
php artisan passport:client --password
```

64.6.2 請求令牌

密碼授權方式的客戶端建立好後，你就可以使用使用者信箱和密碼向 `/oauth/token` 路由發起 `POST` 請求，以獲取訪問令牌。請記住，該路由已經在 `Passport::routes` 方法中定義，你無需再手動實現它。如果請求成功，你將在返回 JSON 串中獲取到 `access_token` 和 `refresh_token` :

```
use Illuminate\Support\Facades\Http;

$response = Http::asForm()->post('http://passport-app.test/oauth/token', [
    'grant_type' => 'password',
    'client_id' => 'client-id',
    'client_secret' => 'client-secret',
    'username' => 'taylor@laravel.com',
    'password' => 'my-password',
    'scope' => '',
]);

return $response->json();
```

> **技巧**
> 請記住，默認情況下 access token 都是長生命週期的，但是如果有需要的話，你可以主動去 設定 access token 的過期時間 。

64.6.3 請求所有的範疇

當使用密碼授權（password grant）或者客戶端認證授權（client credentials grant）方式時，你可能希望將應用所有的範疇範圍都授權給令牌。你可以通過設定 scope 參數為 `*` 來實現。 一旦你這樣設定了，所有的 `can` 方法都將返回 `true` 值。 此範圍只能在密碼授權 `password` 或客戶端認證授權 `client_credentials` 下使用：

```
use Illuminate\Support\Facades\Http;
```

```
$response = Http::asForm()->post('http://passport-app.test/oauth/token', [
    'grant_type' => 'password',
    'client_id' => 'client-id',
    'client_secret' => 'client-secret',
    'username' => 'taylor@laravel.com',
    'password' => 'my-password',
    'scope' => '*',
]);
```

64.6.4 自訂使用者提供者

如果你的應用程式使用多個 使用者認證提供器，你可以在建立客戶端通過 `artisan passport:client --password` 命令時使用 `--provider` 選項來指定提供器。 給定的提供器名稱應與應用程式的 `config/auth.php` 組態檔案中定義的有效提供器匹配。 然後，你可以 使用中介軟體保護你的路由 以確保只有來自守衛指定提供器的使用者才被授權。

64.6.5 自訂使用者名稱欄位

當使用密碼授權進行身份驗證時，Passport 將使用可驗證模型的 `email` 屬性作為 「使用者名稱」 。 但是，你可以通過在模型上定義 `findForPassport` 方法來自訂此行為：

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Laravel\Passport\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;

    /**
     * 尋找給定使用者名稱的使用者實例。
     */
    public function findForPassport(string $username): User
    {
        return $this->where('username', $username)->first();
    }
}
```

64.6.6 自訂密碼驗證

當使用密碼授權進行身份驗證時，Passport 將使用模型的 `password` 屬性來驗證給定的密碼。 如果你的模型沒有 `password` 屬性或者你希望自訂密碼驗證邏輯，你可以在模型上定義 `validateForPassportPasswordGrant` 方法：

```
<?php

namespace App\Models;

use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Illuminate\Support\Facades\Hash;
use Laravel\Passport\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;
```

```
    /**
     * 驗證使用者的密碼以獲得 Passport 密碼授權。
     */
    public function validateForPassportPasswordGrant(string $password): bool
    {
        return Hash::check($password, $this->password);
    }
}
```

64.7 隱式授權令牌

> **注意**
> 我們不再推薦使用隱式授權令牌。 相反，你應該選擇 OAuth2 伺服器當前推薦的授權類型 。

隱式授權類似於授權碼授權； 但是，令牌會在不交換授權碼的情況下返回給客戶端。 此授權最常用於無法安全儲存客戶端憑據的 JavaScript 或移動應用程式。 要啟用授權，請在應用程式的 `App\Providers\AuthServiceProvider` 類的 `boot` 方法中呼叫 `enableImplicitGrant` 方法：

```
/**
 * 註冊任何身份驗證/授權服務。
 */
public function boot(): void
{
    Passport::enableImplicitGrant();
}
```

啟用授權後，開發人員可以使用他們的客戶端 ID 從你的應用程式請求訪問令牌。 消費應用程式應該嚮應用程序的 `/oauth/authorize` 路由發出重新導向請求，如下所示：

```
use Illuminate\Http\Request;

Route::get('/redirect', function (Request $request) {
    $request->session()->put('state', $state = Str::random(40));

    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://third-party-app.com/callback',
        'response_type' => 'token',
        'scope' => '',
        'state' => $state,
        // 'prompt' => '', // "none", "consent", or "login"
    ]);

    return redirect('http://passport-app.test/oauth/authorize?'.$query);
});
```

> **技巧**
> 請記住， `/oauth/authorize` 路由已經由 `Passport::routes` 方法定義。 你無需手動定義此路由。

64.8 客戶憑證授予令牌

客戶端憑據授予適用於機器對機器身份驗證。 例如，你可以在通過 API 執行維護任務的計畫作業中使用此授權。

要想讓應用程式可以通過客戶端憑據授權發佈令牌，首先，你需要建立一個客戶端憑據授權客戶端。你可以使用 `passport:client` Artisan 命令的 `--client` 選項來執行此操作：

```
php artisan passport:client --client
```

接下來，要使用這種授權，你首先需要在 app/Http/Kernel.php 的 $routeMiddleware 屬性中新增 CheckClientCredentials 中介軟體：

```
use Laravel\Passport\Http\Middleware\CheckClientCredentials;

protected $middlewareAliases = [
    'client' => CheckClientCredentials::class,
];
```

之後，在路由上附加中介軟體：

```
Route::get('/orders', function (Request $request) {
    ...
})->middleware('client');
```

要將對路由的訪問限制為特定範圍，你可以在將 client 中介軟體附加到路由時提供所需範圍的逗號分隔列表：

```
Route::get('/orders', function (Request $request) {
    ...
})->middleware('client:check-status,your-scope');
```

64.8.1 檢索令牌

要使用此授權類型檢索令牌，請向 oauth/token 端點發出請求：

```
use Illuminate\Support\Facades\Http;

$response = Http::asForm()->post('http://passport-app.test/oauth/token', [
    'grant_type' => 'client_credentials',
    'client_id' => 'client-id',
    'client_secret' => 'client-secret',
    'scope' => 'your-scope',
]);

return $response->json()['access_token'];
```

64.9 個人訪問令牌

有時，你的使用者要在不經過傳統的授權碼重新導向流程的情況下向自己頒發訪問令牌。允許使用者通過應用程式使用者介面對自己發佈令牌，有助於使用者體驗你的 API，或者也可以將其作為一種更簡單的發佈訪問令牌的方式。

> **技巧**
> 如果你的應用程式主要使用 Passport 來發佈個人訪問令牌，請考慮使用 Laravel 的輕量級第一方庫 Laravel Sanctum 來發佈 API 訪問令牌。

64.9.1 建立個人訪問客戶端

在應用程式發出個人訪問令牌前，你需要在 passport:client 命令後帶上 --personal 參數來建立對應的客戶端。如果你已經運行了 passport:install 命令，則無需再運行此命令:

```
php artisan passport:client --personal
```

建立個人訪問客戶端後，將客戶端的 ID 和純文字金鑰放在應用程式的 .env 檔案中:

```
PASSPORT_PERSONAL_ACCESS_CLIENT_ID="client-id-value"
PASSPORT_PERSONAL_ACCESS_CLIENT_SECRET="unhashed-client-secret-value"
```

64.9.2 管理個人令牌

建立個人訪問客戶端後，你可以使用 App\Models\User 模型實例的 createToken 方法來為給定使用者發佈令牌。 createToken 方法接受令牌的名稱作為其第一個參數和可選的 範疇 陣列作為其第二個參數:

```
use App\Models\User;

$user = User::find(1);

// 建立沒有範疇的令牌...
$token = $user->createToken('Token Name')->accessToken;

// 建立具有範疇的令牌...
$token = $user->createToken('My Token', ['place-orders'])->accessToken;
```

64.9.2.1 JSON API

Passport 中還有一個用於管理個人訪問令牌的 JSON API。你可以將其與你自己的前端配對，為你的使用者提供一個用於管理個人訪問令牌的儀表板。下面，我們將回顧所有用於管理個人訪問令牌的 API 。為了方便起見，我們將使用 Axios 來演示向 API 發出 HTTP 請求。

JSON API 由 web 和 auth 這兩個中介軟體保護；因此，只能從你自己的應用程式中呼叫它。無法從外部源呼叫它。

64.9.2.2 GET /oauth/scopes

此路由會返回應用中定義的所有 範疇 。你可以使用此路由列出使用者可以分配給個人訪問令牌的範圍:

```
axios.get('/oauth/scopes')
    .then(response => {
        console.log(response.data);
    });
```

64.9.2.3 GET /oauth/personal-access-tokens

此路由返回認證使用者建立的所有個人訪問令牌。這主要用於列出使用者的所有令牌，以便他們可以編輯和撤銷它們:

```
axios.get('/oauth/personal-access-tokens')
    .then(response => {
        console.log(response.data);
    });
```

64.9.2.4 POST /oauth/personal-access-tokens

此路由建立新的個人訪問令牌。它需要兩個資料：令牌的 name 和 scopes 。

```
const data = {
    name: 'Token Name',
    scopes: []
};

axios.post('/oauth/personal-access-tokens', data)
    .then(response => {
        console.log(response.data.accessToken);
    })
    .catch (response => {
        // 列出響應的錯誤...
    });
```

64.9.2.5 `DELETE /oauth/personal-access-tokens/{token-id}`

此路由可用於撤銷個人訪問令牌：

```
axios.delete('/oauth/personal-access-tokens/' + tokenId);
```

64.10 路由保護

64.10.1 通過中介軟體

Passport 包含一個 驗證保護機制 驗證請求中傳入的訪問令牌。 若組態 `api` 的看守器使用 `passport` 驅動，你只要在需要有效訪問令牌的路由上指定 `auth:api` 中介軟體即可：

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

> **注意**
> 如果你正在使用 客戶端授權令牌 ，你應該使用 `client` 中介軟體 來保護你的路由，而不是使用 `auth:api` 中介軟體。

64.10.1.1 多個身份驗證看守器

如果你的應用程式可能使用完全不同的 `Eloquent` 模型、不同類型的使用者進行身份驗證，則可能需要為應用程式中的每種使用者設定看守器。這使你可以保護特定看守器的請求。例如，在組態檔案 `config/auth.php` 中設定以下看守器：

```
'api' => [
    'driver' => 'passport',
    'provider' => 'users',
],

'api-customers' => [
    'driver' => 'passport',
    'provider' => 'customers',
],
```

以下路由將使用 `customers` 使用者提供者的 `api-customers` 看守器來驗證傳入的請求：

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

> **技巧**
> 關於使用 Passport 的多個使用者提供器的更多資訊，請參考 密碼認證文件 。

64.10.2 傳遞訪問令牌

當呼叫 Passport 保護下的路由時，接入的 API 應用需要將訪問令牌作為 `Bearer` 令牌放在要求標頭 `Authorization` 中。例如，使用 Guzzle HTTP 庫時：

```
use Illuminate\Support\Facades\Http;

$response = Http::withHeaders([
    'Accept' => 'application/json',
    'Authorization' => 'Bearer '.$accessToken,
])->get('https://passport-app.test/api/user');
```

```
return $response->json();
```

64.11 令牌範疇

範疇可以讓 API 客戶端在請求帳戶授權時請求特定的權限。例如，如果你正在建構電子商務應用程式，並不是所有接入的 API 應用都需要下訂單的功能。你可以讓接入的 API 應用只被允許授權訪問訂單發貨狀態。換句話說，範疇允許應用程式的使用者限制第三方應用程式執行的操作。

64.11.1 定義範疇

你可以在 `App\Providers\AuthServiceProvider` 的 `boot` 方法中使用 `Passport::tokensCan` 方法來定義 API 的範疇。`tokensCan` 方法接受一個包含範疇名稱和描述的陣列作為參數。範疇描述將會在授權確認頁中直接展示給使用者，你可以將其定義為任何你需要的內容：

```
/**
 * 註冊身份驗證/授權服務。
 */
public function boot(): void
{
    Passport::tokensCan([
        'place-orders' => 'Place orders',
        'check-status' => 'Check order status',
    ]);
}
```

64.11.2 默認範疇

如果客戶端沒有請求任何特定的範圍，你可以在 `App\Providers\AuthServiceProvider` 類的 `boot` 方法中使用 `setDefaultScope` 方法來定義默認的範疇。

```
use Laravel\Passport\Passport;

Passport::tokensCan([
    'place-orders' => 'Place orders',
    'check-status' => 'Check order status',
]);

Passport::setDefaultScope([
    'check-status',
    'place-orders',
]);
```

> **技巧** Passport 的默認範疇不適用於由使用者生成的個人訪問令牌。

64.11.3 給令牌分配範疇

64.11.3.1 請求授權碼

使用授權碼請求訪問令牌時，接入的應用需為 `scope` 參數指定所需範疇。 `scope` 參數包含多個範疇時，名稱之間使用空格分割：

```
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'code',
```

```
        'scope' => 'place-orders check-status',
    ]);

    return redirect('http://passport-app.test/oauth/authorize?'.$query);
});
```

64.11.3.2 分發個人訪問令牌

使用 App\Models\User 模型的 createToken 方法發放個人訪問令牌時，可以將所需範疇的陣列作為第二個參數傳給此方法：

```
$token = $user->createToken('My Token', ['place-orders'])->accessToken;
```

64.11.4 檢查範疇

Passport 包含兩個中介軟體，可用於驗證傳入的請求是否包含訪問指定範疇的令牌。使用之前，需要將下面的中介軟體新增到 app/Http/Kernel.php 檔案的 $middlewareAliases 屬性中：

```
'scopes' => \Laravel\Passport\Http\Middleware\CheckScopes::class,
'scope' => \Laravel\Passport\Http\Middleware\CheckForAnyScope::class,
```

64.11.4.1 檢查所有範疇

路由可以使用 scopes 中介軟體來檢查當前請求是否擁有指定的所有範疇：

```
Route::get('/orders', function () {
    // 訪問令牌具有 "check-status" 和 "place-orders" 範疇...
})->middleware(['auth:api', 'scopes:check-status,place-orders']);
```

64.11.4.2 檢查任意範疇

路由可以使用 scope 中介軟體來檢查當前請求是否擁有指定的 *任意* 範疇：

```
Route::get('/orders', function () {
    // 訪問令牌具有 "check-status" 或 "place-orders" 範疇...
})->middleware(['auth:api', 'scope:check-status,place-orders']);
```

64.11.4.3 檢查令牌實例上的範疇

即使含有訪問令牌驗證的請求已經通過應用程式的驗證，你仍然可以使用當前授權 App\Models\User 實例上的 tokenCan 方法來驗證令牌是否擁有指定的範疇：

```
use Illuminate\Http\Request;

Route::get('/orders', function (Request $request) {
    if ($request->user()->tokenCan('place-orders')) {
        // ...
    }
});
```

64.11.4.4 附加範疇方法

scopeIds 方法將返回所有已定義 ID / 名稱的陣列：

```
use Laravel\Passport\Passport;

Passport::scopeIds();
```

scopes 方法將返回一個包含所有已定義範疇陣列的 Laravel\Passport\Scope 實例：

```
Passport::scopes();
```

scopesFor 方法將返回與給定 ID / 名稱匹配的 Laravel\Passport\Scope 實例陣列：

```
Passport::scopesFor(['place-orders', 'check-status']);
```

你可以使用 hasScope 方法確定是否已定義給定範疇：

```
Passport::hasScope('place-orders');
```

64.12 使用 JavaScript 接入 API

在建構 API 時， 如果能通過 JavaScript 應用接入自己的 API 將會給開發過程帶來極大的便利。這種 API 開發方法允許你使用自己的應用程式的 API 和別人共享的 API 。你的 Web 應用程式、移動應用程式、第三方應用程式以及可能在各種軟體包管理器上發佈的任何 SDK 都可能會使用相同的 API 。

通常，如果要在 JavaScript 應用程式中使用 API ，需要手動嚮應用程序傳送訪問令牌，並將其傳遞給應用程式。但是， Passport 有一個可以處理這個問題的中介軟體。將 CreateFreshApiToken 中介軟體新增到 app/Http/Kernel.php 檔案中的 web 中介軟體組就可以了：

```
'web' => [
    // 其他中介軟體...
    \Laravel\Passport\Http\Middleware\CreateFreshApiToken::class,
],
```

> **注意**
> 你需要確保 CreateFreshApiToken 中介軟體是你的中介軟體堆疊中的最後一個中介軟體。

該中介軟體會將 laravel_token cookie 附加到你的響應中。該 cookie 將包含一個加密後的 JWT ， Passport 將用來驗證來自 JavaScript 應用程式的 API 請求。JWT 的生命週期等於你的 session.lifetime 組態值。至此，你可以在不明確傳遞訪問令牌的情況下嚮應用程序的 API 發出請求：

```
axios.get('/api/user')
    .then(response => {
        console.log(response.data);
    });
```

64.12.1.1 自訂 Cookie 名稱

如果需要，你可以在 App\Providers\AuthServiceProvider 類的 boot 方法中使用 Passport::cookie 方法來自訂 laravel_token cookie 的名稱：

```
/**
 * 註冊認證 / 授權服務.
 */
public function boot(): void
{
    Passport::cookie('custom_name');
}
```

64.12.1.2 CSRF 保護

當使用這種授權方法時，你需要確認請求中包含有效的 CSRF 令牌。默認的 Laravel JavaScript 腳手架會包含一個 Axios 實例，該實例是自動使用加密的 XSRF-TOKEN cookie 值在同源請求上傳送 X-XSRF-TOKEN 要求標頭。

> **技巧**
> 如果你選擇傳送 X-CSRF-TOKEN 要求標頭而不是 X-XSRF-TOKEN ，則需要使用 csrf_token() 提供的未加密令牌。

64.13 事件

Passport 在發出訪問令牌和刷新令牌時引發事件。 你可以使用這些事件來修改或撤消資料庫中的其他訪問令牌。如果你願意，可以在應用程式的 `App\Providers\EventServiceProvider` 類中將監聽器註冊到這些事件：

```
/**
 * 應用程式的事件監聽器對應
 *
 * @var array
 */
protected $listen = [
    'Laravel\Passport\Events\AccessTokenCreated' => [
        'App\Listeners\RevokeOldTokens',
    ],

    'Laravel\Passport\Events\RefreshTokenCreated' => [
        'App\Listeners\PruneOldTokens',
    ],
];
```

64.14 測試

Passport 的 `actingAs` 方法可以指定當前已認證使用者及其範疇。`actingAs` 方法的第一個參數是使用者實例，第二個參數是使用者令牌範疇陣列：

```
use App\Models\User;
use Laravel\Passport\Passport;

public function test_servers_can_be_created(): void
{
    Passport::actingAs(
        User::factory()->create(),
        ['create-servers']
    );

    $response = $this->post('/api/create-server');

    $response->assertStatus(201);
}
```

Passport 的 `actingAsClient` 方法可以指定當前已認證使用者及其範疇。 `actingAsClient` 方法的第一個參數是使用者實例，第二個參數是使用者令牌範疇陣列：

```
use Laravel\Passport\Client;
use Laravel\Passport\Passport;

public function test_orders_can_be_retrieved(): void
{
    Passport::actingAsClient(
        Client::factory()->create(),
        ['check-status']
    );

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

    $response->assertStatus(200);
}
```

65 Pennant 測試新功能

65.1 介紹

Laravel Pennant 是一個簡單輕量的特性標誌包，沒有臃腫。特性標誌使你可以有信心地逐步推出新的應用程式功能，測試新的介面設計，支援基幹開發策略等等。

65.2 安裝

首先，使用 Composer 包管理器將 Pennant 安裝到你的項目中：

```
composer require laravel/pennant
```

接下來，你應該使用 `vendor:publish` Artisan 命令發佈 Pennant 組態和遷移檔案： `vendor:publish` Artisan command:

```
php artisan vendor:publish --provider="Laravel\Pennant\PennantServiceProvider"
```

最後，你應該運行應用程式的資料庫遷移。這將建立一個 `features` 表，`Pennant` 使用它來驅動其 `database` 驅動程式：

```
php artisan migrate
```

65.3 組態

在發佈 Pennant 資源之後，組態檔案將位於 `config/pennant.php`。此組態檔案允許你指定 Pennant 用於儲存已解析的特性標誌值的默認儲存機制。

Pennant 支援使用 `array` 驅動程式在記憶體陣列中儲存已解析的特性標誌值。或者，Pennant 可以使用 `database` 驅動程式在關聯式資料庫中持久儲存已解析的特性標誌值，這是 Pennant 使用的默認儲存機制。

65.4 定義特性

要定義特性，你可以使用 `Feature` 門面提供的 `define` 方法。你需要為該特性提供一個名稱以及一個閉包，用於解析該特性的初始值。

通常，特性是在服務提供程序中使用 `Feature` 門面定義的。閉包將接收特性檢查的“範疇”。最常見的是，範疇是當前已認證的使用者。在此示例中，我們將定義一個功能，用於逐步嚮應用程序使用者推出新的 API：

```
<?php

namespace App\Providers;

use App\Models\User;
use Illuminate\Support\Lottery;
use Illuminate\Support\ServiceProvider;
use Laravel\Pennant\Feature;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     */
```

```
    public function boot(): void
    {
        Feature::define('new-api', fn (User $user) => match (true) {
            $user->isInternalTeamMember() => true,
            $user->isHighTrafficCustomer() => false,
            default => Lottery::odds(1 / 100),
        });
    }
}
```

正如你所看到的，我們對我們的特性有以下規則：

- 所有內部團隊成員應使用新 API。
- 任何高流量客戶不應使用新 API。
- 否則，該特性應在具有 1/100 機率啟動的使用者中隨機分配。

首次檢查給定使用者的 `new-api` 特性時，儲存驅動程式將儲存閉包的結果。下一次針對相同使用者檢查特性時，將從儲存中檢索該值，不會呼叫閉包。

為方便起見，如果特性定義僅返回一個 Lottery，你可以完全省略閉包：

```
Feature::define('site-redesign', Lottery::odds(1, 1000));
```

65.4.1 基於類的特性

Pennant 還允許你定義基於類的特性。不像基於閉包的特性定義，不需要在服務提供者中註冊基於類的特性。為了建立一個基於類的特性，你可以呼叫 `pennant:feature` Artisan 命令。默認情況下，特性類將被放置在你的應用程式的 `app/Features` 目錄中：

```
php artisan pennant:feature NewApi
```

在編寫特性類時，你只需要定義一個 `resolve` 方法，用於為給定的範圍解析特性的初始值。同樣，範圍通常是當前經過身份驗證的使用者：

```
<?php

namespace App\Features;

use Illuminate\Support\Lottery;

class NewApi
{
    /**
     * 解析特性的初始值.
     */
    public function resolve(User $user): mixed
    {
        return match (true) {
            $user->isInternalTeamMember() => true,
            $user->isHighTrafficCustomer() => false,
            default => Lottery::odds(1 / 100),
        };
    }
}
```

注 特性類是通過容器,解析的，因此在需要時可以在特性類的建構函式中注入依賴項。

65.5 檢查特性

要確定一個特性是否處於活動狀態，你可以在 `Feature` 門面上使用 `active` 方法。默認情況下，特性針對當前已認證的使用者進行檢查：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Http\Response;
use Laravel\Pennant\Feature;

class PodcastController
{
    /**
     * 顯示資源的列表.
     */
    public function index(Request $request): Response
    {
        return Feature::active('new-api')
                ? $this->resolveNewApiResponse($request)
                : $this->resolveLegacyApiResponse($request);
    }
    // ...
}
```

為了方便起見，如果一個特徵定義只返回一個抽獎結果，你可以完全省略閉包:

```
Feature::define('site-redesign', Lottery::odds(1, 1000));
```

65.5.1 基於類的特徵

Pennant 還允許你定義基於類的特徵。與基於閉包的特徵定義不同，無需在服務提供者中註冊基於類的特徵。要建立基於類的特徵，你可以呼叫 pennant:feature Artisan 命令。默認情況下，特徵類將被放置在你的應用程式的 app/Features 目錄中。:

```
php artisan pennant:feature NewApi
```

編寫特徵類時，你只需要定義一個 `resolve` 方法，該方法將被呼叫以解析給定範疇的特徵的初始值。同樣，該範疇通常是當前已驗證的使用者。:

```
<?php

namespace App\Features;

use Illuminate\Support\Lottery;

class NewApi
{
    /**
     * 解析特徵的初始值.
     */
    public function resolve(User $user): mixed
    {
        return match (true) {
            $user->isInternalTeamMember() => true,
            $user->isHighTrafficCustomer() => false,
            default => Lottery::odds(1 / 100),
        };
    }
}
```

注意 特徵類通過 容器, 解析，因此在需要時，你可以將依賴項注入到特徵類的建構函式中.

65.6 Checking Features

要確定特徵是否處於活動狀態，你可以在 Feature 門面上使用 `active` 方法。默認情況下，特徵將針對當前已

驗證的使用者進行檢查。:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Http\Response;
use Laravel\Pennant\Feature;

class PodcastController
{
    /**
     *顯示資源的列表.
     */
    public function index(Request $request): Response
    {
        return Feature::active('new-api')
                ? $this->resolveNewApiResponse($request)
                : $this->resolveLegacyApiResponse($request);
    }
    // ...
}
```

雖然默認情況下特性針對當前已認證的使用者進行檢查，但你可以輕鬆地針對其他使用者或範圍檢查特性。為此，使用 `Feature` 門面提供的 `for` 方法：

```
return Feature::for($user)->active('new-api')
        ? $this->resolveNewApiResponse($request)
        : $this->resolveLegacyApiResponse($request);
```

Pennant 還提供了一些額外的方便方法，在確定特性是否活動或不活動時可能非常有用：

```
//確定所有給定的特性是否都活動...
Feature::allAreActive(['new-api', 'site-redesign']);

// 確定任何給定的特性是否都活動...
Feature::someAreActive(['new-api', 'site-redesign']);

// 確定特性是否處於非活動狀態...
Feature::inactive('new-api');

// 確定所有給定的特性是否都處於非活動狀態...
Feature::allAreInactive(['new-api', 'site-redesign']);

// 確定任何給定的特性是否都處於非活動狀態...
Feature::someAreInactive(['new-api', 'site-redesign']);
```

> **注**
>
> 當在 HTTP 上下文之外使用 Pennant（例如在 Artisan 命令或排隊作業中）時，你通常應明確指定特性的範疇。或者，你可以定義一個默認範疇，該範疇考慮到已認證的 HTTP 上下文和未經身份驗證的上下文。

65.6.1.1 檢查基於類的特性

對於基於類的特性，應該在檢查特性時提供類名：

```
<?php

namespace App\Http\Controllers;

use App\Features\NewApi;
use Illuminate\Http\Request;
use Illuminate\Http\Response;
```

```
use Laravel\Pennant\Feature;

class PodcastController
{

    /**
     * 顯示資源的列表.
     */
    public function index(Request $request): Response
    {
        return Feature::active(NewApi::class)
                ? $this->resolveNewApiResponse($request)
                : $this->resolveLegacyApiResponse($request);
    }
    // ...
}
```

65.6.2 條件執行

when 方法可用於在特性啟動時流暢地執行給定的閉包。此外，可以提供第二個閉包，如果特性未啟動，則將執行它：

```
<?php

namespace App\Http\Controllers;

use App\Features\NewApi;
use Illuminate\Http\Request;
use Illuminate\Http\Response;
use Laravel\Pennant\Feature;

class PodcastController
{
    /**
     * 顯示資源的列表.
     */
    public function index(Request $request): Response
    {
        return Feature::when(NewApi::class,
            fn () => $this->resolveNewApiResponse($request),
            fn () => $this->resolveLegacyApiResponse($request),
        );
    }
    // ...
}
```

unless 方法是 when 方法的相反，如果特性未啟動，則執行第一個閉包：

```
return Feature::unless(NewApi::class,

    fn () => $this->resolveLegacyApiResponse($request),

    fn () => $this->resolveNewApiResponse($request),

);
```

65.6.3 HasFeatures Trait

Pennant 的 HasFeatures Trait 可以新增到你的應用的 User 模型（或其他具有特性的模型）中，以提供一種流暢、方便的方式從模型直接檢查特性：

```
<?php

namespace App\Models;
```

```
use Illuminate\Foundation\Auth\User as Authenticatable;
use Laravel\Pennant\Concerns\HasFeatures;

class User extends Authenticatable
{
    use HasFeatures;
    // ...
}
```

一旦將 HasFeatures Trait 新增到你的模型中，你可以通過呼叫 `features` 方法輕鬆檢查特性：

```
if ($user->features()->active('new-api')) {
    // ...
}
```

當然，`features` 方法提供了許多其他方便的方法來與特性互動：

```
// 值...
$value = $user->features()->value('purchase-button')
$values = $user->features()->values(['new-api', 'purchase-button']);

// 狀態...
$user->features()->active('new-api');
$user->features()->allAreActive(['new-api', 'server-api']);
$user->features()->someAreActive(['new-api', 'server-api']);
$user->features()->inactive('new-api');
$user->features()->allAreInactive(['new-api', 'server-api']);
$user->features()->someAreInactive(['new-api', 'server-api']);

// 條件執行...
$user->features()->when('new-api',
    fn () => /* ... */,
    fn () => /* ... */,
);

$user->features()->unless('new-api',
    fn () => /* ... */,
    fn () => /* ... */,
);
```

65.6.4 Blade 指令

為了使在 Blade 中檢查特性的體驗更加流暢，Pennant 提供了一個 `@feature` 指令：

```
@feature('site-redesign')
<!-- 'site-redesign' 活躍中 -->
@else
<!-- 'site-redesign' 不活躍-->
@endfeature
```

65.6.5 中介軟體

Pennant 還包括一個中介軟體，它可以在路由呼叫之前驗證當前認證使用者是否有訪問功能的權限。首先，你應該將 `EnsureFeaturesAreActive` 中介軟體的別名新增到你的應用程式的 `app/Http/Kernel.php` 檔案中：

```
use Laravel\Pennant\Middleware\EnsureFeaturesAreActive;

protected $middlewareAliases = [
    // ...
 'features' => EnsureFeaturesAreActive::class,
];
```

接下來，你可以將中介軟體分配給一個路由並指定需要訪問該路由的功能。如果當前認證使用者的任何指定功

能未啟動，則路由將返回 `400 Bad Request` HTTP 響應。可以使用逗號分隔的列表指定多個功能：

```
Route::get('/api/servers', function () {
    // ...
})->middleware(['features:new-api,servers-api']);
```

65.6.5.1 自訂響應

如果你希望在未啟動列表中的任何一個功能時自訂中介軟體返回的響應，可以使用 `EnsureFeaturesAreActive` 中介軟體提供的 `whenInactive` 方法。通常，這個方法應該在應用程式的服務提供者的 boot 方法中呼叫：

```
use Illuminate\Http\Request;
use Illuminate\Http\Response;
use Laravel\Pennant\Middleware\EnsureFeaturesAreActive;

/**
 * 載入服務.
 */
public function boot(): void
{
    EnsureFeaturesAreActive::whenInactive(
        function (Request $request, array $features) {
            return new Response(status: 403);
        }
    );
    // ...
}
```

65.6.6 記憶體快取

當檢查特性時，Pennant 將建立一個記憶體快取以儲存結果。如果你使用的是 `database` 驅動程式，則在單個請求中重新檢查相同的功能標誌將不會觸發額外的資料庫查詢。這也確保了該功能在請求的持續時間內具有一致的結果。

如果你需要手動刷新記憶體快取，可以使用 `Feature` 門面提供的 `flushCache` 方法：

```
Feature::flushCache();
```

65.7 範疇

65.7.1 指定範疇

如前所述，特性通常會針對當前已驗證的使用者進行檢查。但這可能並不總是適合你的需求。因此，你可以通過 `Feature` 門面的 `for` 方法來指定要針對哪個範疇檢查給定的特性：

```
return Feature::for($user)->active('new-api')
        ? $this->resolveNewApiResponse($request)
        : $this->resolveLegacyApiResponse($request);
```

當然，特性範疇不限於“使用者”。假設你建構了一個新的結算體驗，你要將其推出給整個團隊而不是單個使用者。也許你希望年齡最大的團隊的推出速度比年輕的團隊慢。你的特性解析閉包可能如下所示：

```
use App\Models\Team;
use Carbon\Carbon;
use Illuminate\Support\Lottery;
use Laravel\Pennant\Feature;
```

```
Feature::define('billing-v2', function (Team $team) {
    if ($team->created_at->isAfter(new Carbon('1st Jan, 2023'))) {
        return true;
    }

    if ($team->created_at->isAfter(new Carbon('1st Jan, 2019'))) {
        return Lottery::odds(1 / 100);
    }

    return Lottery::odds(1 / 1000);
});
```

你會注意到，我們定義的閉包不需要 `User`，而是需要一個 `Team` 模型。要確定該特性是否對使用者的團隊可用，你應該將團隊傳遞給 `Feature` 門面提供的 `for` 方法：

```
if (Feature::for($user->team)->active('billing-v2')) {
    return redirect()->to('/billing/v2');
}
// ...
```

65.7.2 默認範疇

還可以自訂 Pennant 用於檢查特性的默認範疇。例如，你可能希望所有特性都針對當前認證使用者的團隊進行檢查，而不是針對使用者。你可以在應用程式的服務提供程序中指定此範疇。通常，應該在一個應用程式的服務提供程序中完成這個過程:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Auth;
use Illuminate\Support\ServiceProvider;
use Laravel\Pennant\Feature;

class AppServiceProvider extends ServiceProvider
{
    /**
     * 載入程序服務.
     */
    public function boot(): void
    {
        Feature::resolveScopeUsing(fn ($driver) => Auth::user()?->team);
        // ...
    }
}
```

如果沒有通過 `for` 方法顯式提供範疇，則特性檢查將使用當前認證使用者的團隊作為默認範疇：

```
Feature::active('billing-v2');

// 目前等價於...
Feature::for($user->team)->active('billing-v2');
```

65.7.3 空範疇

如果你檢查特性時提供的範疇範圍為 `null`，且特性定義中不支援 `null`（即不是 nullable type 或者沒有在 union type 中包含 `null`），那麼 Pennant 將自動返回 `false` 作為特性的結果值。

因此，如果你傳遞給特性的範疇可能為 null 並且你想要特性值的解析器被呼叫，你應該在特性定義邏輯中處理 `null` 範圍值。在一個 Artisan 命令、排隊作業或未經身份驗證的路由中檢查特性可能會出現 `null` 範疇。因為在這些情況下通常沒有經過身份驗證的使用者，所以默認的範疇將為 `null`。

如果你不總是明確指定特性範疇，則應確保範圍類型為”nullable”，並在特性定義邏輯中處理 `null` 範圍值:

```
use App\Models\User;
use Illuminate\Support\Lottery;
use Laravel\Pennant\Feature;

Feature::define('new-api', fn (User $user) => match (true) {// [tl! remove]
Feature::define('new-api', fn (User|null $user) => match (true) {// [tl! add]
    $user === null => true,// [tl! add]
    $user->isInternalTeamMember() => true,
    $user->isHighTrafficCustomer() => false,
    default => Lottery::odds(1 / 100),
});
```

65.7.4 標識範疇

Pennant 的內建 `array` 和 `database` 儲存驅動程式可以正確地儲存所有 PHP 資料類型以及 Eloquent 模型的範疇識別碼。但是，如果你的應用程式使用第三方的 Pennant 驅動程式，該驅動程式可能不知道如何正確地儲存 Eloquent 模型或應用程式中其他自訂類型的識別碼。

因此，Pennant 允許你通過在應用程式中用作 Pennant 範疇的對象上實現 `FeatureScopeable` 協議來格式化儲存範圍值。

例如，假設你在單個應用程式中使用了兩個不同的特性驅動程式：內建 `database` 驅動程式和第三方的“Flag Rocket”驅動程式。“Flag Rocket”驅動程式不知道如何正確地儲存 Eloquent 模型。相反，它需要一個 `FlagRocketUser` 實例。通過實現 `FeatureScopeable` 協議中的 `toFeatureIdentifier` 方法，我們可以自訂提供給應用程式中每個驅動程式的可儲存範圍值：

```
<?php

namespace App\Models;

use FlagRocket\FlagRocketUser;
use Illuminate\Database\Eloquent\Model;
use Laravel\Pennant\Contracts\FeatureScopeable;

class User extends Model implements FeatureScopeable
{
    /**
     * 將對象強制轉換為給定驅動程式的功能範圍識別碼.
     */
    public function toFeatureIdentifier(string $driver): mixed
    {
        return match($driver) {
        'database' => $this,
        'flag-rocket' => FlagRocketUser::fromId($this->flag_rocket_id),
        };
    }
}
```

65.8 豐富的特徵值

到目前為止，我們主要展示了特性的二進制狀態，即它們是「活動的」還是「非活動的」，但是 Pennant 也允許你儲存豐富的值。

例如，假設你正在測試應用程式的「立即購買」按鈕的三種新顏色。你可以從特性定義中返回一個字串，而不是 `true` 或 `false`：

```
use Illuminate\Support\Arr;
use Laravel\Pennant\Feature;

Feature::define('purchase-button', fn (User $user) => Arr::random([
 'blue-sapphire',
```

```
 'seafoam-green',
 'tart-orange',
]));
```

你可以使用 value 方法檢索 purchase-button 特性的值：

```
$color = Feature::value('purchase-button');
```

Pennant 提供的 Blade 指令也使得根據特性的當前值條件性地呈現內容變得容易：

```
@feature('purchase-button', 'blue-sapphire')
<!-- 'blue-sapphire' is active -->
@elsefeature('purchase-button', 'seafoam-green')
<!-- 'seafoam-green' is active -->
@elsefeature('purchase-button', 'tart-orange')
<!-- 'tart-orange' is active -->
@endfeature
```

使用豐富值時，重要的是要知道，只要特性具有除 false 以外的任何值，它就被視為「活動」。

在呼叫條件 when 方法時，特性的豐富值將提供給第一個閉包：

```
Feature::when('purchase-button',
    fn ($color) => /* ... */,
    fn () => /* ... */,
);
```

同樣，當呼叫條件 unless 方法時，特性的豐富值將提供給可選的第二個閉包：

```
Feature::unless('purchase-button',
    fn () => /* ... */,
    fn ($color) => /* ... */,
);
```

65.9 獲取多個特性

values 方法允許檢索給定範疇的多個特徵：

```
Feature::values(['billing-v2', 'purchase-button']);

// [
// 'billing-v2' => false,
// 'purchase-button' => 'blue-sapphire',
// ]
```

或者，你可以使用 all 方法檢索給定範圍內所有已定義功能的值：

```
Feature::all();

// [
// 'billing-v2' => false,
// 'purchase-button' => 'blue-sapphire',
// 'site-redesign' => true,
// ]
```

但是，基於類的功能是動態註冊的，直到它們被顯式檢查之前，Pennant 並不知道它們的存在。這意味著，如果在當前請求期間尚未檢查過應用程式的基於類的功能，則這些功能可能不會出現在 all 方法返回的結果中。

如果你想確保使用 all 方法時始終包括功能類，你可以使用 Pennant 的功能發現功能。要開始使用，請在你的應用程式的任何服務提供程序之一中呼叫 discover 方法：

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Laravel\Pennant\Feature;
```

```
class AppServiceProvider extends ServiceProvider
{
     /**
      * Bootstrap any application services.
      */
     public function boot(): void
     {
          Feature::discover();
          // ...
     }
}
```

`discover` 方法將註冊應用程式 `app/Features` 目錄中的所有功能類。`all` 方法現在將在其結果中包括這些類，無論它們是否在當前請求期間進行了檢查：

```
Feature::all();

// [
// 'App\Features\NewApi' => true,
// 'billing-v2' => false,
// 'purchase-button' => 'blue-sapphire',
// 'site-redesign' => true,
// ]
```

65.10 預載入

儘管 Pennant 在單個請求中保留了所有已解析功能的記憶體快取，但仍可能遇到性能問題。為了緩解這種情況，Pennant 提供了預載入功能。

為了說明這一點，想像一下我們正在循環中檢查功能是否處於活動狀態：

```
use Laravel\Pennant\Feature;

foreach ($users as $user) {
     if (Feature::for($user)->active('notifications-beta')) {
          $user->notify(new RegistrationSuccess);
     }
}
```

假設我們正在使用資料庫驅動程式，此程式碼將為循環中的每個使用者執行資料庫查詢-執行潛在的數百個查詢。但是，使用 Pennant 的 `load` 方法，我們可以通過預載入一組使用者或範疇的功能值來消除這種潛在的性能瓶頸：

```
Feature::for($users)->load(['notifications-beta']);

foreach ($users as $user) {
     if (Feature::for($user)->active('notifications-beta')) {
          $user->notify(new RegistrationSuccess);
     }
}
```

為了僅在尚未載入功能值時載入它們，你可以使用 `loadMissing` 方法：

```
Feature::for($users)->loadMissing([
  'new-api',
  'purchase-button',
  'notifications-beta',
]);
```

65.11 更新值

當首次解析功能的值時，底層驅動程式將把結果儲存在儲存中。這通常是為了確保在請求之間為你的使用者提供一致的體驗。但是，有時你可能想手動更新功能的儲存值。

為了實現這一點，你可以使用 activate 和 deactivate 方法來切換功能的 「打開」或「關閉」狀態：

```
use Laravel\Pennant\Feature;

// 啟動默認範疇的功能...
Feature::activate('new-api');

// 在給定的範圍中停用功能...
Feature::for($user->team)->deactivate('billing-v2');
```

還可以通過向 activate 方法提供第二個參數來手動設定功能的豐富值：

```
Feature::activate('purchase-button', 'seafoam-green');
```

要指示 Pennant 忘記功能的儲存值，你可以使用 forget 方法。當再次檢查功能時，Pennant 將從其功能定義中解析功能的值：

```
Feature::forget('purchase-button');
```

65.11.1 批次更新

要批次更新儲存的功能值，你可以使用 activateForEveryone 和 deactivateForEveryone 方法。

例如，假設你現在對 new-api 功能的穩定性有信心，並為結帳流程找到了最佳的「purchase-button」顏色-你可以相應地更新所有使用者的儲存值：

```
use Laravel\Pennant\Feature;

Feature::activateForEveryone('new-api');
Feature::activateForEveryone('purchase-button', 'seafoam-green');
```

或者，你可以停用所有使用者的該功能：

```
Feature::deactivateForEveryone('new-api');
```

> **注意：**這將僅更新已由 Pennant 儲存驅動程式儲存的已解析功能值。你還需要更新應用程式中的功能定義。

65.11.2 清除功能

有時，清除儲存中的整個功能可以非常有用。如果你已從應用程式中刪除了功能或已對功能的定義進行了調整，並希望將其部署到所有使用者，則通常需要這樣做。

你可以使用 purge 方法刪除功能的所有儲存值：

```
// 清除單個功能...
Feature::purge('new-api');

// 清除多個功能...
Feature::purge(['new-api', 'purchase-button']);
```

如果你想從儲存中清除所有功能，則可以呼叫 purge 方法而不帶任何參數：

```
Feature::purge();
```

由於在應用程式的部署流程中清除功能可能非常有用，因此 Pennant 包括一個 pennant:purge Artisan 命令：

```
php artisan pennant:purge new-api
php artisan pennant:purge new-api purchase-button
```

65.12 測試

當測試與功能標誌互動的程式碼時，控制測試中返回的功能標誌的最簡單方法是簡單地重新定義該功能。例

如，假設你在應用程式的一個服務提供程序中定義了以下功能：

```
use Illuminate\Support\Arr;
use Laravel\Pennant\Feature;

Feature::define('purchase-button', fn () => Arr::random([
 'blue-sapphire',
 'seafoam-green',
 'tart-orange',
]));
```

要在測試中修改功能的返回值，你可以在測試開始時重新定義該功能。以下測試將始終通過，即使 `Arr::random()` 實現仍然存在於服務提供程序中：

```
use Laravel\Pennant\Feature;

public function test_it_can_control_feature_values()
{
    Feature::define('purchase-button', 'seafoam-green');
    $this->assertSame('seafoam-green', Feature::value('purchase-button'));
}
```

相同的方法也可以用於基於類的功能：

```
use App\Features\NewApi;
use Laravel\Pennant\Feature;

public function test_it_can_control_feature_values()
{
    Feature::define(NewApi::class, true);
    $this->assertTrue(Feature::value(NewApi::class));
}
```

如果你的功能返回一個 `Lottery` 實例，那麼有一些有用的測試輔助函數可用。

65.12.1.1 儲存組態

你可以通過在應用程式的 `phpunit.xml` 檔案中定義 `PENNANT_STORE` 環境變數來組態 Pennant 在測試期間使用的儲存：

```
<?xml version="1.0"  encoding="UTF-8"?>
<phpunit colors="true">
    <!-- ... -->
    <php>
<env name="PENNANT_STORE"  value="array"/>
        <!-- ... -->
    </php>
</phpunit>
```

65.13 新增自訂 Pennant 驅動程式

65.13.1.1 實現驅動程式

如果 Pennant 現有的儲存驅動程式都不符合你的應用程式需求，則可以編寫自己的儲存驅動程式。你的自訂驅動程式應實現 `Laravel\Pennant\Contracts\Driver` 介面：

```
<?php

namespace App\Extensions;

use Laravel\Pennant\Contracts\Driver;

class RedisFeatureDriver implements Driver
{
```

```
    public function define(string $feature, callable $resolver): void {}
    public function defined(): array {}
    public function getAll(array $features): array {}
    public function get(string $feature, mixed $scope): mixed {}
    public function set(string $feature, mixed $scope, mixed $value): void {}
    public function setForAllScopes(string $feature, mixed $value): void {}
    public function delete(string $feature, mixed $scope): void {}
    public function purge(array|null $features): void {}
}
```

現在，我們只需要使用 Redis 連接實現這些方法。可以在 Pennant 原始碼中查看如何實現這些方法的示例。

> **注意**
>
> Laravel 不附帶包含擴展的目錄。你可以自由地將它們放在任何你喜歡的位置。在這個示例中，我們建立了一個 `Extensions` 目錄來存放 `RedisFeatureDriver`。

65.13.1.2 註冊驅動

一旦你的驅動程式被實現，就可以將其註冊到 Laravel 中。要向 Pennant 新增其他驅動程式，可以使用 `Feature` 門面提供的 `extend` 方法。應該在應用程式的 服務提供者 的 `boot` 方法中呼叫 `extend` 方法：

```
<?php

namespace App\Providers;

use App\Extensions\RedisFeatureDriver;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Support\ServiceProvider;
use Laravel\Pennant\Feature;

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
    {
        Feature::extend('redis', function (Application $app) {
            return new RedisFeatureDriver($app->make('redis'), $app->make('events'),
[]);
        });
    }
}
```

一旦驅動程式被註冊，就可以在應用程式的 `config/pennant.php` 組態檔案中使用 `redis` 驅動程式：

```
'stores' => [
    'redis' => [
        'driver' => 'redis',
        'connection' => null,
    ],
    // ...
],
```

65.14 事件

Pennant 分發了各種事件，這些事件在跟蹤應用程式中的特性標誌時非常有用。

65.14.1 `Laravel\Pennant\Events\RetrievingKnownFeature`

該事件在請求特定範疇的已知特徵值第一次被檢索時被觸發。此事件可用於建立和跟蹤應用程式中使用的特徵標記的度量標準。

65.14.2 `Laravel\Pennant\Events\RetrievingUnknownFeature`

當在請求特定範疇的情況下第一次檢索未知特性時，將分派此事件。如果你打算從應用程式中刪除功能標誌，但可能在整個應用程式中留下了某些零散的引用，此事件可能會有用。你可能會發現有用的是監聽此事件並在其發生時 `report` 或拋出異常：

例如，你可能會發現在監聽到此事件並出現此情況時，使用 `report` 或引發異常會很有用：

```
<?php

namespace App\Providers;

use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvider;
use Illuminate\Support\Facades\Event;
use Laravel\Pennant\Events\RetrievingUnknownFeature;

class EventServiceProvider extends ServiceProvider
{
    /**
     * Register any other events for your application.
     */
    public function boot(): void
    {
        Event::listen(function (RetrievingUnknownFeature $event) {
            report("Resolving unknown feature [{$event->feature}].");
        });
    }
}
```

65.14.3 `Laravel\Pennant\Events\DynamicallyDefiningFeature`

當在請求期間首次動態檢查基於類的特性時，將分派此事件。

66 Pint 程式碼風格

66.1 介紹

Laravel Pint 是一款面向極簡主義者的 PHP 程式碼風格固定工具。Pint 是建立在 PHP-CS-Fixer 基礎上，使保持程式碼風格的整潔和一致變得簡單。

Pint 會隨著所有新的 Laravel 應用程式自動安裝，所以你可以立即開始使用它。默認情況下，Pint 不需要任何組態，將通過遵循 Laravel 的觀點性編碼風格來修復你的程式碼風格問題。

66.2 安裝

Pint 已包含在 Laravel 框架的最近版本中，所以無需安裝。然而，對於舊的應用程式，你可以通過 Composer 安裝 Laravel Pint：

```
composer require laravel/pint --dev
```

66.3 運行 Pint

可以通過呼叫你項目中的 `vendor/bin` 目錄下的 `pint` 二進制檔案來指示 Pint 修復程式碼風格問題：

```
./vendor/bin/pint
```

你也可以在特定的檔案或目錄上運行 Pint：

```
./vendor/bin/pint app/Models

./vendor/bin/pint app/Models/User.php
```

Pint 將顯示它所更新的所有檔案的詳細列表。 你可以在呼叫 Pint 時提供 `-v` 選項來查看更多關於 Pint 修改的細節。：

```
./vendor/bin/pint -v
```

如果你只想 Pint 檢查程式碼中風格是否有錯誤，而不實際更改檔案，則可以使用 `--test` 選項：

```
./vendor/bin/pint --test
```

如果你希望 Pint 根據 Git 僅修改未提交更改的檔案，你可以使用 `--dirty` 選項：

```
./vendor/bin/pint --dirty
```

66.4 組態 Pint

如前面所述，Pint 不需要任何組態。但是，如果你希望自訂預設、規則或檢查的資料夾，可以在項目的根目錄中建立一個 `pint.json` 檔案：

```
{
  "preset": "laravel"
}
```

此外，如果你希望使用特定目錄中的 `pint.json`，可以在呼叫 Pint 時提供 `--config` 選項：

```
pint --config vendor/my-company/coding-style/pint.json
```

66.4.1 Presets(預設)

Presets 定義了一組規則，可以用來修復程式碼風格問題。默認情況下，Pint 使用 laravel preset，通過遵循 `Laravel` 的固定編碼風格來修復問題。但是，你可以通過向 Pint 提供 `--preset` 選項來指定一個不同的 preset 值：

```
pint --preset psr12
```

如果你願意，還可以在項目的 `pint.json` 檔案中設定 preset ：

```
{
  "preset": "psr12"
}
```

Pint 目前支援的 presets 有：`laravel`、`psr12` 和 `symfony`。

66.4.2 規則

規則是 Pint 用於修復程式碼風格問題的風格指南。如上所述，presets 是預定義的規則組，適用於大多數 PHP 項目，因此你通常不需要擔心它們所包含的單個規則。

但是，如果你願意，可以在 `pint.json` 檔案中啟用或停用特定規則：

```
{
    "preset": "laravel",
    "rules": {
        "simplified_null_return": true,
        "braces": false,
        "new_with_braces": {
            "anonymous_class": false,
            "named_class": false
        }
    }
}
```

Pint 是基於 PHP-CS-Fixer 建構的。因此，您可以使用它的任何規則來修復項目中的程式碼風格問題： PHP-CS-Fixer Configurator.

66.4.3 排除檔案/資料夾

默認情況下，Pint 將檢查項目中除 `vendor` 目錄以外的所有 `.php` 檔案。如果您希望排除更多資料夾，可以使用 `exclude` 組態選項:

```
{
    "exclude": [
        "my-specific/folder"
    ]
}
```

如果您希望排除包含給定名稱模式的所有檔案，則可以使用 `notName` 組態選項:

```
{
    "notName": [
        "*-my-file.php"
    ]
}
```

如果您想要通過提供檔案的完整路徑來排除檔案，則可以使用 `notPath` 組態選項:

```
{
    "notPath": [
        "path/to/excluded-file.php"
    ]
```

```
}
```

67 Sanctum API 授權

67.1 介紹

Laravel Sanctum 提供了一個輕量級的認證系統，可用於 SPA（單頁應用程式）、移動應用程式和基於簡單令牌的 API。Sanctum 允許的應用程式中的每個使用者為他們的帳戶生成多個 API 令牌。這些令牌可以被授予權限/範圍，以指定令牌允許執行哪些操作。

67.1.1 工作原理

Laravel Sanctum 旨在解決兩個不同的問題。在深入探討該庫之前，讓我們先討論一下每個問題。

67.1.1.1 API 令牌

首先，Sanctum 是一個簡單的包，你可以使用它向你的使用者發出 API 令牌，而無需 OAuth 的複雜性。這個功能受到 GitHub 和其他應用程式發出「訪問令牌」的啟發。例如，假如你的應用程式的「帳戶設定」有一個介面，使用者可以在其中為他們的帳戶生成 API 令牌。你可以使用 Sanctum 生成和管理這些令牌。這些令牌通常具有非常長的過期時間（以年計），但使用者可以隨時手動撤銷它們。

Laravel Sanctum 通過將使用者 API 令牌儲存在單個資料庫表中，並通過應該包含有效 API 令牌的 `Authorization` 標頭對傳入的 HTTP 請求進行身份驗證來提供此功能。

67.1.1.2 SPA 認證

第二個功能，Sanctum 存在的目的是為需要與 Laravel 支援的 API 通訊的單頁應用程式 (SPAs) 提供一種簡單的身份驗證方式。這些 SPAs 可能存在於與 Laravel 應用程式相同的儲存庫中，也可能是一個完全獨立的儲存庫，例如使用 Vue CLI 建立的 SPA 或 Next.js 應用程式。

對於此功能，Sanctum 不使用任何類型的令牌。相反，Sanctum 使用 Laravel 內建基於 cookie 的 session 身份驗證服務。通常，Sanctum 使用 Laravel 的 `web` 認證保護方式實現這一點。這提供了 CSRF 保護、 session 身份驗證以及防止通過 XSS 洩漏身份驗證憑據的好處。

只有在傳入請求來自你自己的 SPA 前端時，Sanctum 才會嘗試使用 cookies 進行身份驗證。當 Sanctum 檢查傳入的 HTTP 請求時，它首先會檢查身份驗證 cookie，如果不存在，則 Sanctum 會檢查 `Authorization` 標頭是否包含有效的 API 令牌。

> **注意** 完全可以只使用 Sanctum 進行 API 令牌身份驗證或只使用 Sanctuary 進行 SPA 身份驗證。僅因為你使用 Sanctum 並不意味著你必須使用它提供的兩個功能。

67.2 安裝

> **注意** 最近的 Laravel 版本已經包括 Laravel Sanctum。但如果你的應用程式的 `composer.json` 檔案不包括 `laravel/sanctum`，你可以遵循下面的安裝說明。

你可以通過 Composer 包管理器安裝 Laravel Sanctum：

```
composer require laravel/sanctum
```

接下來，你應該使用 `vendor:publish` Artisan 命令發佈 Sanctum 組態檔案和遷移檔案。`sanctum` 組態檔案將被放置在你的應用程式的 `config` 目錄中：

```
php artisan vendor:publish --provider="Laravel\Sanctum\SanctumServiceProvider"
```

最後，你應該運行資料庫遷移。Sanctum 會建立一個資料庫表來儲存 API 令牌：

```
php artisan migrate
```

接下來，如果你打算使用 Sanctum 來對 SPA 單頁應用程式進行認證，則應該將 Sanctum 的中介軟體新增到你的應用程式的 `app/Http/Kernel.php` 檔案中的 `api` 中介軟體組中：

```
'api' => [
\Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful::class,
    \Illuminate\Routing\Middleware\ThrottleRequests::class.':api',
    \Illuminate\Routing\Middleware\SubstituteBindings::class,
],
```

67.2.1.1 自訂遷移

如果你不打算使用 Sanctum 的默認遷移檔案，則應該在 `App\Providers\AppServiceProvider` 類的 `register` 方法中呼叫 `Sanctum::ignoreMigrations` 方法。你可以通過執行以下命令匯出默認的遷移檔案：`php artisan vendor:publish --tag=sanctum-migrations`

67.3 組態

67.3.1 覆蓋默認模型

雖然通常不需要，但你可以自由擴展 Sanctum 內部使用的 `PersonalAccessToken` 模型:

```
use Laravel\Sanctum\PersonalAccessToken as SanctumPersonalAccessToken;

class PersonalAccessToken extends SanctumPersonalAccessToken
{
    // ...
}
```

然後，你可以通過 Sanctum 提供的 `usePersonalAccessTokenModel` 方法來指示 Sanctum 使用你的自訂模型。通常，你應該在一個應用程式的服務提供者的 `boot` 方法中呼叫此方法：

```
use App\Models\Sanctum\PersonalAccessToken;
use Laravel\Sanctum\Sanctum;

/**
 * 引導任何應用程式服務。
 */
public function boot(): void
{
    Sanctum::usePersonalAccessTokenModel(PersonalAccessToken::class);
}
```

67.4 API 令牌認證

> **注意** 你不應該使用 API 令牌來認證你自己的第一方單頁應用程式。而應該使用 Sanctum 內建的 SPA 身份驗證功能。

67.4.1 發行 API 令牌

Sanctum 允許你發行 API 令牌/個人訪問令牌，可用於對你的應用程式的 API 請求進行身份驗證。使用 API 令牌發出請求時，應將令牌作為 `Bearer` 令牌包括在 `Authorization` 頭中。

要開始為使用者發行令牌，你的使用者模型應該使用 `Laravel\Sanctum\HasApiTokens` trait：

```
use Laravel\Sanctum\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, HasFactory, Notifiable;
}
```

要發行令牌，你可以使用 `createToken` 方法。`createToken` 方法會返回一個 `Laravel\Sanctum\NewAccessToken` 實例。在將 API 令牌儲存到資料庫之前，令牌將使用 SHA-256 雜湊進行雜湊處理，但是你可以通過 `NewAccessToken` 實例的 `plainTextToken` 屬性訪問令牌的明文值。你應該在令牌被建立後立即將其值顯示給使用者：

```
use Illuminate\Http\Request;

Route::post('/tokens/create', function (Request $request) {
    $token = $request->user()->createToken($request->token_name);

    return ['token' => $token->plainTextToken];
});
```

你可以使用 `HasApiTokens` trait 提供的 `tokens` Eloquent 關聯來訪問使用者的所有令牌：

```
foreach ($user->tokens as $token) {
    // ...
}
```

67.4.2 令牌能力

Sanctum 允許你為令牌分配「能力」。能力的作用類似於 OAuth 的「Scope」。你可以將一個字串能力陣列作為 `createToken` 方法的第二個參數傳遞：

```
return $user->createToken('token-name', ['server:update'])->plainTextToken;
```

當處理由 Sanctum 驗證的入站請求時，你可以使用 `tokenCan` 方法確定令牌是否具有給定的能力：

```
if ($user->tokenCan('server:update')) {
    // ...
}
```

67.4.2.1 令牌能力中介軟體

Sanctum 還包括兩個中介軟體，可用於驗證傳入的請求是否使用授予了給定能力的令牌進行了身份驗證。首先，請將以下中介軟體新增到應用程式的 `app/Http/Kernel.php` 檔案的 `$middlewareAliases` 屬性中：

```
'abilities' => \Laravel\Sanctum\Http\Middleware\CheckAbilities::class,
'ability' => \Laravel\Sanctum\Http\Middleware\CheckForAnyAbility::class,
```

可以將 `abilities` 中介軟體分配給路由，以驗證傳入請求的令牌是否具有所有列出的能力：

```
Route::get('/orders', function () {
    // 令牌具有「check-status」和「place-orders」能力...
})->middleware(['auth:sanctum', 'abilities:check-status,place-orders']);
```

可以將 `ability` 中介軟體分配給路由，以驗證傳入請求的令牌是否至少具有一個列出的能力：

```
Route::get('/orders', function () {
    // 令牌具有「check-status」或「place-orders」能力...
})->middleware(['auth:sanctum', 'ability:check-status,place-orders']);
```

67.4.2.2 第一方 UI 啟動的請求

為了方便起見，如果入站身份驗證請求來自你的第一方 SPA ，並且你正在使用 Sanctum 內建的 SPA 認證，`tokenCan` 方法將始終返回 `true`。

然而，這並不一定意味著你的應用程式必須允許使用者執行該操作。通常，你的應用程式的授權策略 將確定是否已授予令牌執行能力的權限，並檢查使用者實例本身是否允許執行該操作。

例如，如果我們想像一個管理伺服器的應用程式，這可能意味著檢查令牌是否被授權更新伺服器**並且**伺服器屬於使用者：

```
return $request->user()->id === $server->user_id &&
       $request->user()->tokenCan('server:update')
```

首先允許 `tokenCan` 方法被呼叫並始終為第一方 UI 啟動的請求返回 `true` 可能看起來很奇怪。然而，能夠始終假設 API 令牌可用並可通過 `tokenCan` 方法進行檢查非常方便。通過採用這種方法，你可以始終在應用程式的授權策略中呼叫 `tokenCan` 方法，而不用再擔心請求是從應用程式的 UI 觸發還是由 API 的第三方使用者發起的。

67.4.3 保護路由

為了保護路由，使所有入站請求必須進行身份驗證，你應該在你的 `routes/web.php` 和 `routes/api.php` 路由檔案中，將 `sanctum` 認證守衛附加到受保護的路由上。如果該請求來自第三方，該守衛將確保傳入的請求經過身份驗證，要麼是具有狀態的 Cookie 身份驗證請求，要麼是包含有效的 API 令牌標頭的請求。

你可能想知道我們為什麼建議你使用 `sanctum` 守衛在應用程式的 `routes/web.php` 檔案中對路由進行身份驗證。請記住，Sanctum 首先將嘗試使用 Laravel 的典型 session 身份驗證 cookie 對傳入請求進行身份驗證。如果該 cookie 不存在，則 Sanctum 將嘗試使用請求的 `Authorization` 標頭中的令牌來驗證請求。此外，使用 Sanctum 對所有請求進行身份驗證，確保我們可以始終在當前經過身份驗證的使用者實例上呼叫 `tokenCan` 方法：

```
use Illuminate\Http\Request;

Route::middleware('auth:sanctum')->get('/user', function (Request $request) {
    return $request->user();
});
```

67.4.4 撤銷令牌

你可以通過使用 `Laravel\Sanctum\HasApiTokens` trait 提供的 `tokens` 關係，從資料庫中刪除它們來達到「撤銷」令牌的目的：

```
// 撤銷所有令牌...
$user->tokens()->delete();

// 撤銷用於驗證當前請求的令牌...
$request->user()->currentAccessToken()->delete();

// 撤銷特定的令牌...
$user->tokens()->where('id', $tokenId)->delete();
```

67.4.5 令牌有效期

默認情況下，Sanctum 令牌永不過期，並且只能通過撤銷令牌進行無效化。但是，如果你想為你的應用程式 API 令牌組態過期時間，可以通過在應用程式的 `sanctum` 組態檔案中定義的 `expiration` 組態選項進行組態。此組態選項定義發放的令牌被視為過期之前的分鐘數：

```
// 365 天後過期
'expiration' => 525600,
```

如果你已為應用程式組態了令牌過期時間，你可能還希望任務調度來刪除應用程式過期的令牌。幸運的是，Sanctum 包括一個 `sanctum:prune-expired` Artisan 命令，你可以使用它來完成此操作。例如，你可以組態工作排程來刪除所有過期至少 24 小時的令牌資料庫記錄：

```
$schedule->command('sanctum:prune-expired --hours=24')->daily();
```

67.5 SPA 身份驗證

Sanctum 還提供一種簡單的方法來驗證需要與 Laravel API 通訊的單頁面應用程式（SPA）。這些 SPA 可能存在於與你的 Laravel 應用程式相同的儲存庫中，也可能是一個完全獨立的儲存庫。

對於此功能，Sanctum 不使用任何類型的令牌。相反，Sanctum 使用 Laravel 內建的基於 cookie 的 session 身份驗證服務。此身份驗證方法提供了 CSRF 保護、session 身份驗證以及防止身份驗證憑據通過 XSS 洩漏的好處。

> **警告** 為了進行身份驗證，你的 SPA 和 API 必須共享相同的頂級域。但是，它們可以放置在不同的子域中。此外，你應該確保你的請求中傳送 `Accept: application/json` 標頭檔。

67.5.1 組態

67.5.1.1 組態你的第一個域

首先，你應該通過 `sanctum` 組態檔案中的 `stateful` 組態選項來組態你的 SPA 將從哪些域發出請求。此組態設定確定哪些域將在向你的 API 傳送請求時使用 Laravel session cookie 維護「有狀態的」身份驗證。

> **警告** 如果你通過包含連接埠的 URL（`127.0.0.1:8000`）訪問應用程式，你應該確保在域名中包括連接埠號。

67.5.1.2 Sanctum 中介軟體

接下來，你應該將 Sanctum 中介軟體新增到你的 `app/Http/Kernel.php` 檔案中的 `api` 中介軟體組中。此中介軟體負責確保來自你的 SPA 的傳入請求可以使用 Laravel session cookie 進行身份驗證，同時仍允許來自第三方或移動應用程式使用 API 令牌進行身份驗證：

```
'api' => [ \Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful::class,
    \Illuminate\Routing\Middleware\ThrottleRequests::class.':api',
    \Illuminate\Routing\Middleware\SubstituteBindings::class,
],
```

67.5.1.3 CORS 和 Cookies

如果你無法從執行在單獨子域上的 SPA 中進行應用程式身份驗證的話，你可能已錯誤組態了 CORS（跨域資源共享）或 session cookie 設定。

你應該確保你的應用程式的 CORS 組態返回的 `Access-Control-Allow-Credentials` 要求標頭的值為 `True` 。這可以通過在應用程式的 `config/cors.php` 組態檔案中設定 `supports_credentials` 選項為 `true` 來完成。

此外，你應該在應用程式的全域 `axios` 實例中啟用 `withCredentials` 選項。通常，這應該在你的 `resources/js/bootstrap.js` 檔案中進行。如果你沒有使用 Axios 從前端進行 HTTP 請求，你應該使用自己的 HTTP 客戶端進行等效組態：

```
axios.defaults.withCredentials = true;
```

最後，你應該確保應用程式的 session cookie 域組態支援根域的任何子域。你可以通過在應用程式的 `config/session.php` 組態檔案中使用前導 `.` 作為域的前綴來實現此目的：

```
'domain' => '.domain.com',
```

67.5.2 身份驗證

67.5.2.1 CSRF 保護

要驗證你的 SPA，你的 SPA 的「登錄」頁面應首先向 `/sanctum/csrf-cookie` 發出請求以初始化應用程式的 CSRF 保護：

```
axios.get('/sanctum/csrf-cookie').then(response => {
    // Login...
});
```

在此請求期間，Laravel 將設定一個包含當前 CSRF 令牌的 `XSRF-TOKEN` cookie。然後，此令牌應在隨後的請求中通過 `X-XSRF-TOKEN` 標頭傳遞，其中某些 HTTP 客戶端庫（如 Axios 和 Angular HttpClient）將自動為你執行此操作。如果你的 JavaScript HTTP 庫沒有為你設定值，你將需要手動設定 `X-XSRF-TOKEN` 要求標頭以匹配此路由設定的 `XSRF-TOKEN` cookie 的值。

67.5.2.2 登錄

一旦已經初始化了 CSRF 保護，你應該向 Laravel 應用程式的 `/login` 路由發出 `POST` 請求。這個 `/login` 路由可以通過手動實現或使用像 Laravel Fortify 這樣的無要求標頭身份驗證包來實現。

如果登錄請求成功，你將被驗證，隨後對應用程式路由的後續請求將通過 Laravel 應用程式發出的 session cookie 自動進行身份驗證。此外，由於你的應用程式已經發出了對 `/sanctum/csrf-cookie` 路由的請求，因此只要你的 JavaScript HTTP 客戶端在 `X-XSRF-TOKEN` 標頭中傳送了 `XSRF-TOKEN` cookie 的值，後續的請求應該自動接受 CSRF 保護。

當然，如果你的使用者 session 因缺乏活動而過期，那麼對 Laravel 應用程式的後續請求可能會收到 401 或 419 HTTP 錯誤響應。在這種情況下，你應該將使用者重新導向到你 SPA 的登錄頁面。

> **警告** 你可以自己編寫 `/login` 端點；但是，你應該確保使用 Laravel 提供的標準基於 session 的身份驗證服務來驗證使用者。通常，這意味著使用 `web` 身份驗證 Guard。

67.5.3 保護路由

為了保護路由，以便所有傳入的請求必須進行身份驗證，你應該將 `sanctum` 身份驗證 guard 附加到 `routes/api.php` 檔案中的 API 路由上。這個 guard 將確保傳入的請求被驗證為來自你的 SPA 的有狀態身份驗證請求，或者如果請求來自第三方，則包含有效的 API 令牌標頭：

```
use Illuminate\Http\Request;

Route::middleware('auth:sanctum')->get('/user', function (Request $request) {
    return $request->user();
});
```

67.5.4 授權私有廣播頻道

如果你的 SPA 需要對私有/存在 broadcast 頻道進行身份驗證，你應該在 `routes/api.php` 檔案中呼叫 `Broadcast::routes` 方法：

```
Broadcast::routes(['middleware' => ['auth:sanctum']]);
```

接下來，為了讓 Pusher 的授權請求成功，你需要在初始化 Laravel Echo 時提供自訂的 Pusher `authorizer`。這允許你的應用程式組態 Pusher 以使用為跨域請求正確組態的 `axios` 實例：

```
window.Echo = new Echo({
    broadcaster: "pusher",
    cluster: import.meta.env.VITE_PUSHER_APP_CLUSTER,
    encrypted: true,
    key: import.meta.env.VITE_PUSHER_APP_KEY,
    authorizer: (channel, options) => {
        return {
            authorize: (socketId, callback) => {
                axios.post('/api/broadcasting/auth', {
                    socket_id: socketId,
                    channel_name: channel.name
                })
                .then(response => {
                    callback(false, response.data);
                })
                .catch(error => {
                    callback(true, error);
                });
            }
        };
    },
})
```

67.6 移動應用程式身份驗證

你也可以使用 Sanctum 令牌來驗證你的移動應用程式對 API 的請求。驗證移動應用程式請求的過程類似於驗證第三方 API 請求；但是，你將發佈 API 令牌的方式有所不同。

67.6.1 發佈 API 令牌

首先，請建立一個路由，該路由接受使用者的電子郵件/使用者名稱、密碼和裝置名稱，然後將這些憑據交換為新的 Sanctum 令牌。給此端點提供「裝置名稱」的目的是為了記錄資訊，僅供參考。通常來說，裝置名稱值應該是使用者能夠識別的名稱，例如「Nuno's iPhone 12」。

通常，你將從你的移動應用程式的「登錄」頁面向令牌端點發出請求。此端點將返回純文字的 API 令牌，可以儲存在移動裝置上，並用於進行額外的 API 請求：

```
use App\Models\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Illuminate\Validation\ValidationException;

Route::post('/sanctum/token', function (Request $request) {
    $request->validate([
        'email' => 'required|email',
        'password' => 'required',
        'device_name' => 'required',
    ]);

    $user = User::where('email', $request->email)->first();

    if (! $user || ! Hash::check($request->password, $user->password)) {
        throw ValidationException::withMessages([
            'email' => ['The provided credentials are incorrect.'],
        ]);
    }
```

```
    return $user->createToken($request->device_name)->plainTextToken;
});
```

當移動應用程式使用令牌向你的應用程式發出 API 請求時，它應該將令牌作為 `Bearer` 令牌放在 `Authorization` 標頭中傳遞。

> **注意** 當為移動應用程式發佈令牌時，你可以自由指定令牌權限。

67.6.2 路由保護

如之前所述，你可以通過使用 `sanctum` 認證守衛附加到路由上來保護路由，以便所有傳入請求都必須進行身份驗證：

```
Route::middleware('auth:sanctum')->get('/user', function (Request $request) {
    return $request->user();
});
```

67.6.3 撤銷令牌

為了允許使用者撤銷發放給移動裝置的 API 令牌，你可以在 Web 應用程式 UI 的 「帳戶設定」部分中按名稱列出它們，並提供一個「撤銷」按鈕。當使用者點選「撤銷」按鈕時，你可以從資料庫中刪除令牌。請記住，你可以通過 `Laravel\Sanctum\HasApiTokens` 特性提供的 `tokens` 關係訪問使用者的 API 令牌：

```
// 撤銷所有令牌...
$user->tokens()->delete();

// 撤銷特定令牌...
$user->tokens()->where('id', $tokenId)->delete();
```

67.7 測試

在測試時，`Sanctum::actingAs` 方法可用於驗證使用者並指定為其令牌授予哪些能力：

```
use App\Models\User;
use Laravel\Sanctum\Sanctum;

public function test_task_list_can_be_retrieved(): void
{
    Sanctum::actingAs(
        User::factory()->create(),
        ['view-tasks']
    );

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

    $response->assertOk();
}
```

如果你想授予令牌所有的能力，你應該在提供給 `actingAs` 方法的能力列表中包含 `*` ：

```
Sanctum::actingAs(
    User::factory()->create(),
    ['*']
);
```

68 Scout 全文搜尋

68.1 介紹

Laravel Scout 為 Eloquent models 的全文搜尋提供了一個簡單的基於驅動程式的解決方案，通過使用模型觀察者，Scout 將自動同步 Eloquent 記錄的搜尋索引。

目前，Scout 附帶 Algolia, Meilisearch, 和 MySQL / PostgreSQL (`database`) 驅動程式。此外，Scout 包括一個「collection」驅動程式，該驅動程式專為本地開發使用而設計，不需要任何外部依賴項或第三方服務。此外，編寫自訂驅動程式很簡單，你可以使用自己的搜尋實現自由擴展 Scout。

68.2 安裝

首先，通過 Composer 軟體包管理器安裝 Scout：

```
composer require laravel/scout
```

Scout 安裝完成後，使用 Artisan 命令 `vendor:publish` 生成 Scout 組態檔案。此命令將會在你的 `config` 目錄下 生成一個 `scout.php` 組態檔案:

```
php artisan vendor:publish --provider="Laravel\Scout\ScoutServiceProvider"
```

最後，在你要做搜尋的模型中新增 `Laravel\Scout\Searchable` trait 。這個 trait 會註冊一個模型觀察者來保持模型和搜尋驅動的同步:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Laravel\Scout\Searchable;

class Post extends Model
{
    use Searchable;
}
```

68.2.1 驅動的先決條件

68.2.1.1 Algolia

使用 Algolia 驅動時，需要在 `config/scout.php` 組態檔案組態你的 `Algolia` `id` 和 `secret` 憑證。組態好憑證之後，還需要使用 Composer 安裝 Algolia PHP SDK：

```
composer require algolia/algoliasearch-client-php
```

68.2.1.2 Meilisearch

Meilisearch 是一個速度極快的開源搜尋引擎。如果你不確定如何在本地機器上安裝 MeiliSearch，你可以使用 Laravel 官方支援的 Docker 開發環境 Laravel Sail。

使用 MeiliSearch 驅動程式時，你需要通過 Composer 包管理器安裝 MeiliSearch PHP SDK：

```
composer require meilisearch/meilisearch-php http-interop/http-factory-guzzle
```

然後，在應用程式的 .env 檔案中設定 SCOUT_DRIVER 環境變數以及你的 MeiliSearch host 和 key 憑據：

```
SCOUT_DRIVER=meilisearch
MEILISEARCH_HOST=http://127.0.0.1:7700
MEILISEARCH_KEY=masterKey
```

更多關於 MeiliSearch 的資訊，請參考 MeiliSearch 技術文件。

此外，你應該通過查看 MeiliSearch 關於二進制相容性的文件確保安裝與你的 MeiliSearch 二進製版本相容的 meilisearch/meilisearch-php 版本。

> Meilisearch service itself. 注意：在使用 MeiliSearch 的應用程式上升級 Scout 時，你應該始終留意查看關於 MeiliSearch 升級發佈的其他重大（破壞性）更改，以保證升級順利。

68.2.2 佇列

雖然不強制要求使用 Scout，但在使用該庫之前，強烈建議組態一個佇列驅動。運行佇列 worker 將允許 Scout 將所有同步模型資訊到搜尋索引的操作都放入佇列中，從而為你的應用程式的 Web 介面提供更快的響應時間。

一旦你組態了佇列驅動程式，請將 config/scout.php 組態檔案中的 queue 選項的值設定為 true：

```
'queue' => true,
```

即使將 queue 選項設定為 false，也要記住有些 Scout 驅動程式（如 Algolia 和 Meilisearch）始終非同步記錄索引。也就是說，即使索引操作已在 Laravel 應用程式中完成，但搜尋引擎本身可能不會立即反映新記錄和更新記錄。

要指定 Scout 使用的連接和佇列，請將 queue 組態選項定義為陣列：

```
'queue' => [
    'connection' => 'redis',
    'queue' => 'scout'
],
```

68.3 組態

68.3.1 組態模型索引

每個 Eloquent 模型都與一個給定的搜尋「索引」同步，該索引包含該模型的所有可搜尋記錄。換句話說，可以將每個索引視為 MySQL 表。默認情況下，每個模型將持久化到與模型的典型「表」名稱匹配的索引中。通常，這是模型名稱的複數形式；但是，你可以通過在模型上重寫 searchableAs 方法來自訂模型的索引：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Laravel\Scout\Searchable;

class Post extends Model
{
    use Searchable;

    /**
     * 獲取與模型關聯的索引的名稱.
     */
    public function searchableAs(): string
    {
        return 'posts_index';
```

```
    }
}
```

68.3.2 組態可搜尋資料

默認情況下，給定模型的 `toArray` 形式的整個內容將被持久化到其搜尋索引中。如果要自訂同步到搜尋索引的資料，可以重寫模型上的 `toSearchableArray` 方法：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Laravel\Scout\Searchable;

class Post extends Model
{
    use Searchable;

    /**
     * 獲取模型的可索引資料。
     *
     * @return array<string, mixed>
     */
    public function toSearchableArray(): array
    {
        $array = $this->toArray();

        // 自訂資料陣列...

        return $array;
    }
}
```

一些搜尋引擎（如 Meilisearch）只會在正確的資料類型上執行過濾操作（`>`、`<` 等）。因此，在使用這些搜尋引擎並自訂可搜尋資料時，你應該確保數值類型被轉換為正確的類型：

```
public function toSearchableArray()
{
    return [
        'id' => (int) $this->id,
        'name' => $this->name,
        'price' => (float) $this->price,
    ];
}
```

68.3.2.1 組態可過濾資料和索引設定 (Meilisearch)

與 Scout 的其他驅動程式不同，Meilisearch 要求你預定義索引搜尋設定，例如可過濾屬性、可排序屬性和其他支援的設定欄位。

可過濾屬性是你在呼叫 Scout 的 `where` 方法時想要過濾的任何屬性，而可排序屬性是你在呼叫 Scout 的 `orderBy` 方法時想要排序的任何屬性。要定義索引設定，請調整應用程式的 `scout` 組態檔案中 `meilisearch` 組態條目的 `index-settings` 部分：

```
use App\Models\User;
use App\Models\Flight;

'meilisearch' => [
    'host' => env('MEILISEARCH_HOST', 'http://localhost:7700'),
    'key' => env('MEILISEARCH_KEY', null),
    'index-settings' => [
        User::class => [
```

```
            'filterableAttributes'=> ['id', 'name', 'email'],
            'sortableAttributes' => ['created_at'],
            // 其他設定欄位...
        ],
        Flight::class => [
            'filterableAttributes'=> ['id', 'destination'],
            'sortableAttributes' => ['updated_at'],
        ],
    ],
],
```

如果給定索引下的模型可以進行軟刪除，並且已包含在 `index-settings` 陣列中，Scout 將自動支援在該索引上過濾軟刪除的模型。如果你沒有其他可過濾或可排序的屬性來定義軟刪除的模型索引，則可以簡單地向該模型的 `index-settings` 陣列新增一個空條目：

```
'index-settings' => [
    Flight::class => []
],
```

在組態應用程式的索引設定之後，你必須呼叫 `scout:sync-index-settings` Artisan 命令。此命令將向 Meilisearch 通知你當前組態的索引設定。為了方便起見，你可能希望將此命令作為部署過程的一部分：

```
php artisan scout:sync-index-settings
```

68.3.3 組態模型 ID

默認情況下，Scout 將使用模型的主鍵作為儲存在搜尋索引中的模型唯一 ID/鍵。如果你需要自訂此行為，可以重寫模型的 `getScoutKey` 和 `getScoutKeyName` 方法：

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Laravel\Scout\Searchable;

class User extends Model
{
    use Searchable;

    /**
     * 獲取這個模型用於索引的值.
     */
    public function getScoutKey(): mixed
    {
        return $this->email;
    }

    /**
     * 獲取這個模型用於索引的鍵.
     */
    public function getScoutKeyName(): mixed
    {
        return 'email';
    }
}
```

68.3.4 設定模型的搜尋引擎

當進行搜尋時，Scout 通常會使用應用程式的 `scout` 組態檔案中指定的默認搜尋引擎。但是，可以通過在模型上覆蓋 `searchableUsing` 方法來更改特定模型的搜尋引擎：

```
<?php
```

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Laravel\Scout\Engines\Engine;
use Laravel\Scout\EngineManager;
use Laravel\Scout\Searchable;

class User extends Model
{
    use Searchable;

    /**
     * 獲取這個模型用於索引的搜尋引擎.
     */
    public function searchableUsing(): Engine
    {
        return app(EngineManager::class)->engine('meilisearch');
    }
}
```

68.3.5 組態模型 ID

默認情況下，Scout 將使用模型的主鍵作為儲存在搜尋索引中的模型的唯一 ID /鍵。如果你需要自訂此行為，你可以覆蓋模型上的 `getScoutKey` 和 `getScoutKeyName` 方法:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Laravel\Scout\Searchable;

class User extends Model
{
    use Searchable;

    /**
     * 獲取用於索引模型的值.
     */
    public function getScoutKey(): mixed
    {
        return $this->email;
    }

    /**
     * 獲取用於索引模型的鍵名.
     */
    public function getScoutKeyName(): mixed
    {
        return 'email';
    }
}
```

68.3.6 按型號組態搜尋引擎

搜尋時，Scout 通常使用你應用程式的 Scout 組態檔案中指定的默認搜尋引擎。然而，可以通過覆蓋模型上的 `searchableUsing` 方法來更改特定模型的搜尋引擎:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
```

```
use Laravel\Scout\Engines\Engine;
use Laravel\Scout\EngineManager;
use Laravel\Scout\Searchable;

class User extends Model
{
    use Searchable;

    /**
     * 獲取用於索引模型的引擎.
     */
    public function searchableUsing(): Engine
    {
        return app(EngineManager::class)->engine('meilisearch');
    }
}
```

68.3.7 識別使用者

如果你在使用 Algolia 時想要自動識別使用者，Scout 可以幫助你。將已認證的使用者與搜尋操作相關聯，可以在 Algolia 的儀表板中查看搜尋分析時非常有用。你可以通過在應用程式的 `.env` 檔案中將 `SCOUT_IDENTIFY` 環境變數定義為 `true` 來啟用使用者識別：

```
SCOUT_IDENTIFY=true
```

啟用此功能還會將請求的 IP 地址和已驗證的使用者的主要識別碼傳遞給 Algolia，以便將此資料與使用者發出的任何搜尋請求相關聯。

68.4 資料庫/集合引擎

68.4.1 資料庫引擎

> 注意：目前，資料庫引擎支援 MySQL 和 PostgreSQL。

如果你的應用程式與小到中等大小的資料庫互動或工作負載較輕，你可能會發現使用 Scout 的 「database」 引擎更為方便。資料庫引擎將使用 「where like」 子句和全文索引來過濾你現有資料庫的結果，以確定適用於你查詢的搜尋結果。

要使用資料庫引擎，你可以簡單地將 `SCOUT_DRIVER` 環境變數的值設定為 `database`，或直接在你的應用程式的 `scout` 組態檔案中指定 `database` 驅動程式：

```
SCOUT_DRIVER=database
```

一旦你已將資料庫引擎指定為首選驅動程式，你必須組態你的可搜尋資料。然後，你可以開始執行搜尋查詢來查詢你的模型。使用資料庫引擎時，不需要進行搜尋引擎索引，例如用於填充 Algolia 或 Meilisearch 索引所需的索引。

68.4.1.1 自訂資料庫搜尋策略

默認情況下，資料庫引擎將對你所組態為可搜尋的每個模型屬性執行 「where like」 查詢。然而，在某些情況下，這可能會導致性能不佳。因此，資料庫引擎的搜尋策略可以組態，以便某些指定的列利用全文搜尋查詢，或者僅使用 「where like」 約束來搜尋字串的前綴(`example%`)，而不是在整個字串中搜尋(`%example%`)。

為了定義這種行為，你可以將 PHP 屬性賦值給你的模型的 toSearchableArray 方法。任何未被分配其他搜尋策略行為的列將繼續使用默認的「where like」 策略：

```
use Laravel\Scout\Attributes\SearchUsingFullText;
use Laravel\Scout\Attributes\SearchUsingPrefix;

/**
 * 獲取模型的可索引資料陣列。
 *
 * @return array<string, mixed>
 */
#[SearchUsingPrefix(['id', 'email'])]
#[SearchUsingFullText(['bio'])]
public function toSearchableArray(): array
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'bio' => $this->bio,
    ];
}
```

注意：在指定列應使用全文查詢約束之前，請確保已為該列分配全文索引。

68.4.2 集合引擎

在本地開發過程中，你可以自由地使用 Algolia 或 Meilisearch 搜尋引擎，但你可能會發現使用「集合」引擎更加方便。集合引擎將使用「where」子句和集合過濾器來從你現有的資料庫結果中確定適用於你查詢的搜尋結果。當使用此引擎時，無需對可搜尋模型進行「索引」，因為它們只需從本地資料庫中檢索即可。

要使用收集引擎，你可以簡單地將 `SCOUT_DRIVER` 環境變數的值設定為 `collection`，或者直接在你的應用的 `scout` 組態檔案中指定 `collection` 驅動程式：

```
SCOUT_DRIVER=collection
```

一旦你將收集驅動程式指定為首選驅動程式，你就可以開始針對你的模型執行搜尋查詢。使用收集引擎時，不需要進行搜尋引擎索引，如種子 Algolia 或 Meilisearch 索引所需的索引。

68.4.2.1 與資料庫引擎的區別

乍一看，「資料庫」和「收集」引擎非常相似。它們都直接與你的資料庫互動以檢索搜尋結果。然而，收集引擎不使用全文索引或 `LIKE` 子句來尋找匹配的記錄。相反，它會拉取所有可能的記錄，並使用 Laravel 的 `Str::is` 助手來確定搜尋字串是否存在於模型屬性值中。

收集引擎是最便攜的搜尋引擎，因為它適用於 Laravel 支援的所有關係型資料庫（包括 SQLite 和 SQL Server）；然而，它的效率比 Scout 的資料庫引擎低。

68.5 索引

68.5.1 批次匯入

如果你要將 Scout 安裝到現有項目中，你可能已經有需要匯入到你的索引中的資料庫記錄。Scout 提供了一個 `scout:import` Artisan 命令，你可以使用它將所有現有記錄匯入到你的搜尋索引中：

```
php artisan scout:import "App\Models\Post"
```

`flush` 命令可用於從你的搜尋索引中刪除模型的所有記錄：

```
php artisan scout:flush "App\Models\Post"
```

68.5.1.1 修改匯入查詢

如果你想修改用於獲取所有批次匯入模型的查詢，你可以在你的模型上定義一個 `makeAllSearchableUsing` 方法。這是一個很好的地方，可以在匯入模型之前新增任何必要的關係載入:

```
use Illuminate\Database\Eloquent\Builder;

/**
 * 修改用於檢索模型的查詢，使所有模型都可搜尋.
 */
protected function makeAllSearchableUsing(Builder $query): Builder
{
    return $query->with('author');
}
```

68.5.2 新增記錄

一旦你將 `Laravel\Scout\Searchable` Trait 新增到模型中，你所需要做的就是**保存**或**建立**一個模型實例，它將自動新增到你的搜尋索引中。如果你將 Scout 組態為使用佇列，則此操作將由你的佇列工作者在後台執行:

```
use App\Models\Order;

$order = new Order;

// ...

$order->save();
```

68.5.2.1 通過查詢新增記錄

如果你想通過 Eloquent 查詢將模型集合新增到你的搜尋索引中，你可以將 `searchable` 方法連結到 Eloquent 查詢中。`searchable` 方法會將查詢的結果分塊並將記錄新增到你的搜尋索引中。同樣，如果你已經組態了 Scout 來使用佇列，那麼所有的塊都將由你的佇列工作程序在後台匯入:

```
use App\Models\Order;

Order::where('price', '>', 100)->searchable();
```

你也可以在 Eloquent 關係實例上呼叫 `searchable` 方法:

```
$user->orders()->searchable();
```

如果你已經有一組 Eloquent 模型對像在記憶體中，可以在該集合實例上呼叫 `searchable` 方法，將模型實例新增到對應的索引中：

```
$orders->searchable();
```

> **注意** searchable 方法可以被視為「upsert」操作。換句話說，如果模型記錄已經存在於索引中，它將被更新。如果它在搜尋索引中不存在，則將其新增到索引中。

68.5.3 更新記錄

要更新可搜尋的模型，只需更新模型實例的屬性並將模型保存到你的資料庫中。Scout 將自動將更改持久化到你的搜尋索引中：

```
use App\Models\Order;

$order = Order::find(1);

// 更新訂單...

$order->save();
```

你還可以在 Eloquent 查詢實例上呼叫 `searchable` 方法，以更新模型的集合。如果模型不存在於搜尋索引中，則將建立它們：

```
Order::where('price', '>', 100)->searchable();
```

如果想要更新關係中所有模型的搜尋索引記錄，可以在關係實例上呼叫 `searchable` 方法：

```
$user->orders()->searchable();
```

或者，如果你已經在記憶體中有一組 Eloquent 模型，可以在該集合實例上呼叫 `searchable` 方法，以更新對應索引中的模型實例：

```
$orders->searchable();
```

68.5.4 刪除記錄

要從索引中刪除記錄，只需從資料庫中刪除模型即可。即使你正在使用軟刪除模型，也可以這樣做：

```
use App\Models\Order;

$order = Order::find(1);

$order->delete();
```

如果你不想在刪除記錄之前檢索模型，你可以在 Eloquent 查詢實例上使用 `unsearchable` 方法：

```
Order::where('price', '>', 100)->unsearchable();
```

如果你想刪除與關係中所有模型相關的搜尋索引記錄，你可以在關係實例上呼叫 `unsearchable` 方法：

```
$user->orders()->unsearchable();
```

或者，如果你已經有一組 Eloquent 模型在記憶體中，你可以在集合實例上呼叫 `unsearchable` 方法，將模型實例從相應的索引中移除：

```
$orders->unsearchable();
```

68.5.5 暫停索引

有時你可能需要在不將模型資料同步到搜尋索引的情況下對模型執行一批 Eloquent 操作。你可以使用 `withoutSyncingToSearch` 方法來實現這一點。該方法接受一個閉包，將立即執行。在閉包內發生的任何模型操作都不會同步到模型的索引：

```
use App\Models\Order;

Order::withoutSyncingToSearch(function () {
    // 執行模型動作...
});
```

68.5.6 有條件地搜尋模型實例

有時你可能需要在某些條件下使模型可搜尋。例如，假設你有一個 `App\Models\Post` 模型，它可能處於兩種狀態之一：「draft」（草稿）和 「published」（已發佈）。你可能只想讓 「published」（已發佈）的帖子可以被搜尋。為了實現這一點，你可以在你的模型中定義一個 `shouldBeSearchable` 方法：

```
/**
 * 確定模型是否應該可搜尋。
 */
public function shouldBeSearchable(): bool
{
    return $this->isPublished();
}
```

`shouldBeSearchable` 方法僅在通過 `save` 和 `create` 方法、查詢或關係操作模型時應用。直接使用

`searchable` 方法使模型或集合可搜尋將覆蓋 `shouldBeSearchable` 方法的結果。

> **警告**
> 當使用 Scout 的「database」（資料庫）引擎時，`shouldBeSearchable` 方法不適用，因為所有可搜尋的資料都儲存在資料庫中。要在使用資料庫引擎時實現類似的行為，你應該使用 where 子句代替。

68.6 搜尋

你可以使用 `search` 方法開始搜尋一個模型。搜尋方法接受一個將用於搜尋模型的字串。然後，你應該在搜尋查詢上連結 `get` 方法以檢索與給定搜尋查詢匹配的 Eloquent 模型：

```
use App\Models\Order;

$orders = Order::search('Star Trek')->get();
```

由於 Scout 搜尋返回一組 Eloquent 模型，你甚至可以直接從路由或 controller 返回結果，它們將自動轉換為 JSON：

```
use App\Models\Order;
use Illuminate\Http\Request;

Route::get('/search', function (Request $request) {
    return Order::search($request->search)->get();
});
```

如果你想在將搜尋結果轉換為 Eloquent 模型之前獲取原始搜尋結果，你可以使用 `raw` 方法：

```
$orders = Order::search('Star Trek')->raw();
```

68.6.1.1 自訂索引

搜尋查詢通常會在模型的 `searchableAs` 方法指定的索引上執行。然而，你可以使用 `within` 方法指定一個應該被搜尋的自訂索引：

```
$orders = Order::search('Star Trek')
    ->within('tv_shows_popularity_desc')
    ->get();
```

68.6.2 Where 子句

Scout 允許你在搜尋查詢中新增簡單的「where」子句。目前，這些子句只支援基本的數值相等檢查，主要用於通過所有者 ID 限定搜尋查詢：

```
use App\Models\Order;

$orders = Order::search('Star Trek')->where('user_id', 1)->get();
```

你可以使用 `whereIn` 方法將結果約束在給定的一組值中：

```
$orders = Order::search('Star Trek')->whereIn(
    'status', ['paid', 'open']
)->get();
```

由於搜尋索引不是關聯式資料庫，所以目前不支援更高級的「where」子句。

> **警告** 如果你的應用程式使用了 Meilisearch，你必須在使用 Scout 的「where」子句之前組態你的應用程式的可過濾屬性。

68.6.3 分頁

除了檢索模型集合之外，你還可以使用 `paginate` 方法對搜尋結果進行分頁。此方法將返回一個 `Illuminate\Pagination\LengthAwarePaginator` 實例，就像你對傳統的 Eloquent 查詢進行分頁一樣：

```
use App\Models\Order;

$orders = Order::search('Star Trek')->paginate();
```

你可以通過將數量作為第一個參數傳遞給 `paginate` 方法來指定每頁檢索多少個模型：

```
$orders = Order::search('Star Trek')->paginate(15);
```

一旦你檢索到了結果，你可以使用 Blade 顯示結果並渲染頁面連結，就像你對傳統的 Eloquent 查詢進行分頁一樣：

```
<div class="container">
    @foreach ($orders as $order)
        {{ $order->price }}
    @endforeach
</div>

{{ $orders->links() }}
```

當然，如果你想將分頁結果作為 JSON 檢索，可以直接從路由或 controller 返回分頁器實例：

```
use App\Models\Order;
use Illuminate\Http\Request;

Route::get('/orders', function (Request $request) {
    return Order::search($request->input('query'))->paginate(15);
});
```

> **警告**
> 由於搜尋引擎不瞭解你的 Eloquent 模型的全域範疇定義，因此在使用 Scout 分頁的應用程式中，你不應該使用全域範疇。或者，你應該在通過 Scout 搜尋時重新建立全域範疇的約束。

68.6.4 軟刪除

如果你的索引模型使用了軟刪除並且你需要搜尋已軟刪除的模型，請將 `config/scout.php` 組態檔案中的 `soft_delete` 選項設定為 `true`。

```
'soft_delete' => true,
```

當這個組態選項為 `true` 時，Scout 不會從搜尋索引中刪除已軟刪除的模型。相反，它會在索引記錄上設定一個隱藏的 `__soft_deleted` 屬性。然後，你可以使用 `withTrashed` 或 `onlyTrashed` 方法在搜尋時檢索已軟刪除的記錄：

```
use App\Models\Order;

// 在檢索結果時包含已刪除的記錄。。。
$orders = Order::search('Star Trek')->withTrashed()->get();

// 僅在檢索結果時包含已刪除的記錄。。。
$orders = Order::search('Star Trek')->onlyTrashed()->get();
```

> 技巧：當使用 `forceDelete` 永久刪除軟刪除模型時，Scout 將自動從搜尋索引中移除它。

68.6.5 自訂引擎搜尋

如果你需要對一個引擎的搜尋行為進行高級定製，你可以將一個閉包作為 `search` 方法的第二個參數傳遞進去。例如，你可以使用這個回呼在搜尋查詢傳遞給 Algolia 之前將地理位置資料新增到你的搜尋選項中：

```
use Algolia\AlgoliaSearch\SearchIndex;
use App\Models\Order;

Order::search(
    'Star Trek',
    function (SearchIndex $algolia, string $query, array $options) {
        $options['body']['query']['bool']['filter']['geo_distance'] = [
            'distance' => '1000km',
            'location' => ['lat' => 36, 'lon' => 111],
        ];

        return $algolia->search($query, $options);
    }
)->get();
```

68.6.5.1 自訂 Eloquent 結果查詢

在 Scout 從你的應用程式搜尋引擎中檢索到匹配的 Eloquent 模型列表後，Eloquent 會使用模型的主鍵檢索所有匹配的模型。你可以通過呼叫 `query` 方法來自訂此查詢。`query` 方法接受一個閉包，它將接收 Eloquent 查詢建構器實例作為參數：

```
use App\Models\Order;
use Illuminate\Database\Eloquent\Builder;

$orders = Order::search('Star Trek')
    ->query(fn (Builder $query) => $query->with('invoices'))
    ->get();
```

由於此回呼是在從應用程式的搜尋引擎中已經檢索到相關模型之後呼叫的，因此 `query` 方法不應用於「過濾」結果。相反，你應該使用 Scout where 子句。

68.7 自訂引擎

68.7.1.1 編寫引擎

如果 Scout 內建的搜尋引擎不符合你的需求，你可以編寫自己的自訂引擎並將其註冊到 Scout。你的引擎應該繼承 `Laravel\Scout\Engines\Engine` 抽象類。這個抽象類包含了你的自訂引擎必須實現的八個方法：

```
use Laravel\Scout\Builder;

abstract public function update($models);
abstract public function delete($models);
abstract public function search(Builder $builder);
abstract public function paginate(Builder $builder, $perPage, $page);
abstract public function mapIds($results);
abstract public function map(Builder $builder, $results, $model);
abstract public function getTotalCount($results);
abstract public function flush($model);
```

你可能會發現，查看 `Laravel\Scout\Engines\AlgoliaEngine` 類上這些方法的實現會很有幫助。這個類將為你提供一個良好的起點，以學習如何在自己的引擎中實現每個方法。

68.7.1.2 註冊引擎

一旦你編寫好自己的引擎，就可以使用 Scout 的引擎管理器的 `extend` 方法將其註冊到 Scout 中。Scout 的引擎管理器可以從 Laravel 服務容器中解析。你應該從 `App\Providers\AppServiceProvider` 類或應用程式使用的任何其他服務提供程序的 `boot` 方法中呼叫 `extend` 方法：

```
use App\ScoutExtensions\MySqlSearchEngine;
use Laravel\Scout\EngineManager;
```

```
/**
 * 引導任何應用程式服務。
 */
public function boot(): void
{
    resolve(EngineManager::class)->extend('mysql', function () {
        return new MySqlSearchEngine;
    });
}
```

引擎註冊後，你可以在 `config/scout.php` , 組態檔案中指定它為默認的 Scout `driver`

```
'driver' => 'mysql',
```

68.8 生成宏命令

如果你想要自訂生成器方法，你可以使用 `Laravel\Scout\Builder` 類下的 “macro” 方法。 通常，定義「macros」時，需要實現 service provider's `boot` 方法:

```
use Illuminate\Support\Facades\Response;
use Illuminate\Support\ServiceProvider;
use Laravel\Scout\Builder;

/**
 * 引導任何應用程式服務。
 */
public function boot(): void
{
    Builder::macro('count', function () {
        return $this->engine()->getTotalCount(
            $this->engine()->search($this)
        );
    });
}
```

`macro` 函數接受一個名字作為第一個參數，第二個參數為一個閉包涵數。當呼叫 `Laravel\Scout\Builder` 宏命令時，呼叫這個函數.

```
use App\Models\Order;

Order::search('Star Trek')->count();
```

69 Socialite 第三方登入

69.1 簡介

除了典型的基於表單的身份驗證之外，Laravel 還提供了一種使用 Laravel Socialite 對 OAuth providers 進行身份驗證的簡單方便的方法。 Socialite 目前支援 Facebook，Twitter，LinkedIn，Google，GitHub，GitLab 和 Bitbucket 的身份驗證。

> 技巧：其他平台的驅動器可以在 Socialite Providers 社區驅動網站尋找。

69.2 安裝

在開始使用 Socialite 之前，通過 Composer 軟體包管理器將軟體包新增到項目的依賴項中:

```
composer require laravel/socialite
```

69.3 升級

升級到 Socialite 的新主要版本時，請務必仔細查看 the upgrade guide.

69.4 組態

在使用 Socialite 之前，你需要為應用程式使用的 OAuth 提供程序新增憑據。通常，可以通過在要驗證的服務的儀表板中建立“開發人員應用程式”來檢索這些憑據。

這些憑證應該放在你的 `config/services.php` 組態檔案中， 並且應該使用 `facebook`, `twitter` (OAuth 1.0), `twitter-oauth-2` (OAuth 2.0), `linkedin`, `google`, `github`, `gitlab`, 或者 `bitbucket`, 取決於應用程式所需的提供商：

```
'github' => [
    'client_id' => env('GITHUB_CLIENT_ID'),
    'client_secret' => env('GITHUB_CLIENT_SECRET'),
    'redirect' => 'http://example.com/callback-url',
],
```

> 技巧：如果 `redirect` 項的值包含一個相對路徑，它將會自動解析為全稱 URL。

69.5 認證

69.5.1 路由

要使用 OAuth 提供程序對使用者進行身份驗證，你需要兩個路由：一個用於將使用者重新導向到 OAuth provider，另一個用於在身份驗證後接收來自 provider 的回呼。下面的示例 controller 演示了這兩個路由的實現：

```
use Laravel\Socialite\Facades\Socialite;

Route::get('/auth/redirect', function () {
```

```
    return Socialite::driver('github')->redirect();
});

Route::get('/auth/callback', function () {
    $user = Socialite::driver('github')->user();

    // $user->token
});
```

`redirect` 提供的方法 `Socialite` facade 負責將使用者重新導向到 OAuth provider，而該 user 方法將讀取傳入的請求並在身份驗證後從提供程序中檢索使用者的資訊。

69.5.2 身份驗證和儲存

從 OAuth 提供程序檢索到使用者後，你可以確定該使用者是否存在於應用程式的資料庫中並驗證使用者。如果使用者在應用程式的資料庫中不存在，通常會在資料庫中建立一條新記錄來代表該使用者：

```
use App\Models\User;
use Illuminate\Support\Facades\Auth;
use Laravel\Socialite\Facades\Socialite;

Route::get('/auth/callback', function () {
    $githubUser = Socialite::driver('github')->user();

    $user = User::updateOrCreate([
        'github_id' => $githubUser->id,
    ], [
        'name' => $githubUser->name,
        'email' => $githubUser->email,
        'github_token' => $githubUser->token,
        'github_refresh_token' => $githubUser->refreshToken,
    ]);

    Auth::login($user);

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

> 技巧：有關特定 OAuth 提供商提供哪些使用者資訊的更多資訊，請參閱有關 檢索使用者詳細資訊 的文件。

69.5.3 訪問範疇

在重新導向使用者之前，你還可以使用 `scopes` 方法在請求中新增其他「範疇」。此方法會將所有現有範疇與你提供的範疇合併：

```
use Laravel\Socialite\Facades\Socialite;

return Socialite::driver('github')
    ->scopes(['read:user', 'public_repo'])
    ->redirect();
```

你可以使用 `setScopes` 方法覆蓋所有現有範圍：

```
return Socialite::driver('github')
    ->setScopes(['read:user', 'public_repo'])
    ->redirect();
```

69.5.4 可選參數

許多 OAuth providers 支援重新導向請求中的可選參數。 要在請求中包含任何可選參數，請使用關聯陣列呼叫

`with` 方法：

```
use Laravel\Socialite\Facades\Socialite;

return Socialite::driver('google')
    ->with(['hd' => 'example.com'])
    ->redirect();
```

注意：使用 `with` 方法時, 注意不要傳遞任何保留的關鍵字，例如 `state` 或 `response_type`。

69.6 檢索使用者詳細資訊

在將使用者重新導向回你的身份驗證回呼路由之後，你可以使用 Socialite 的 `user` 方法檢索使用者的詳細資訊。`user` 方法為返回的使用者對象提供了各種屬性和方法，你可以使用這些屬性和方法在你自己的資料庫中儲存有關該使用者的資訊。

你可以使用不同的屬性和方法這取決於要進行身份驗證的 OAuth 提供程序是否支援 OAuth 1.0 或 OAuth 2.0：

```
use Laravel\Socialite\Facades\Socialite;

Route::get('/auth/callback', function () {
    $user = Socialite::driver('github')->user();

    // OAuth 2.0 providers...
    $token = $user->token;
    $refreshToken = $user->refreshToken;
    $expiresIn = $user->expiresIn;

    // OAuth 1.0 providers...
    $token = $user->token;
    $tokenSecret = $user->tokenSecret;

    // All providers...
    $user->getId();
    $user->getNickname();
    $user->getName();
    $user->getEmail();
    $user->getAvatar();
});
```

69.6.1.1 從令牌中檢索使用者詳細資訊 (OAuth2)

如果你已經有了一個使用者的有效訪問令牌，你可以使用 Socialite 的 `userFromToken` 方法檢索其詳細資訊：

```
use Laravel\Socialite\Facades\Socialite;

$user = Socialite::driver('github')->userFromToken($token);
```

69.6.1.2 從令牌中檢索使用者詳細資訊 (OAuth2)

如果你已經有了一對有效的使用者令牌/秘鑰，你可以使用 Socialite 的 `userFromTokenAndSecret` 方法檢索他們的詳細資訊：

```
use Laravel\Socialite\Facades\Socialite;

$user = Socialite::driver('twitter')->userFromTokenAndSecret($token, $secret);
```

69.6.1.3 無認證狀態

`stateless` 方法可用於停用 session 狀態驗證。 這在向 API 新增社交身份驗證時非常有用：

```
use Laravel\Socialite\Facades\Socialite;

return Socialite::driver('google')->stateless()->user();
```

注意：Twitter 驅動程式不支援無狀態身份驗證，它使用 OAuth 1.0 進行身份驗證

70 Telescope 偵錯工具

70.1 簡介

Laravel Telescope 是 Laravel 本地開發環境的絕佳伴侶。Telescope 可以洞察你的應用程式的請求、異常、日誌條目、資料庫查詢、排隊的作業、郵件、消息通知、快取操作、定時工作排程、變數列印等。

70.2 安裝

你可以使用 Composer 將 Telescope 安裝到 Laravel 項目中：

```
composer require laravel/telescope
```

安裝 Telescope 後，你應使用 `telescope:install` 命令來發佈其公共資源，然後運行 `migrate` 命令執行資料庫變更來建立和保存 Telescope 需要的資料：

```
php artisan telescope:install

php artisan migrate
```

70.2.1.1 自訂遷移

如果不打算使用 Telescope 的默認遷移，則應在應用程式的 `App\Providers\AppServiceProvider` 類的 `register` 方法中呼叫 `Telescope::ignoreMigrations` 方法。你可以使用以下命令匯出默認遷移：`php artisan vendor:publish --tag=telescope-migrations`

70.2.2 僅本地安裝

如果你僅打算使用 Telescope 來幫助你的本地開發，你可以使用 `--dev` 標記安裝 Telescope：

```
composer require laravel/telescope --dev

php artisan telescope:install

php artisan migrate
```

運行 `telescope:install` 後，應該從應用程式的 `config/app.php` 組態檔案中刪除 `TelescopeServiceProvider` 服務提供者註冊。手動在 `App\Providers\AppServiceProvider` 類的 `register` 方法中註冊 telescope 的服務提供者來替代。在註冊提供者之前，我們會確保當前環境是 `local`：

```
/**
 * 註冊應用服務。
 */
public function register(): void
{
    if ($this->app->environment('local')) {
        $this->app->register(\Laravel\Telescope\TelescopeServiceProvider::class);
        $this->app->register(TelescopeServiceProvider::class);
    }
}
```

最後，你還應該將以下內容新增到你的 `composer.json` 檔案中來防止 Telescope 擴展包被 自動發現：

```
"extra": {
    "laravel": {
        "dont-discover": [
```

```
            "laravel/telescope"
        ]
    }
},
```

70.2.3 組態

發佈 Telescope 的資原始檔後，其主要組態檔案將位於 `config/telescope.php`。此組態檔案允許你組態監聽 觀察者選項，每個組態選項都包含其用途說明，因此請務必徹底瀏覽此檔案。

如果需要，你可以使用 `enabled` 組態選項完全停用 Telescope 的資料收集：

```
'enabled' => env('TELESCOPE_ENABLED', true),
```

70.2.4 資料修改

有了資料修改， `telescope_entries` 表可以非常快速地累積記錄。 為了緩解這個問題，你應該使用 調度 每天運行 `telescope:prune` 命令：

```
$schedule->command('telescope:prune')->daily();
```

默認情況下，將獲取超過 24 小時的所有資料。在呼叫命令時可以使用 `hours` 選項來確定保留 `Telescope` 資料的時間。例如，以下命令將刪除 48 小時前建立的所有記錄：

```
$schedule->command('telescope:prune --hours=48')->daily();
```

70.2.5 儀表板授權

訪問 `/telescope` 即可顯示儀表盤。默認情況下，你只能在 `local` 環境中訪問此儀表板。 在 `app/Providers/TelescopeServiceProvider.php` 檔案中，有一個 gate 授權 。此授權能控制在 **非本地** 環境中對 Telescope 的訪問。你可以根據需要隨意修改此權限以限制對 Telescope 安裝和訪問：

```
use App\Models\User;

/**
 * 註冊 Telescope gate。
 *
 * 該 gate 確定誰可以在非本地環境中訪問 Telescope
 */
protected function gate(): void
{
    Gate::define('viewTelescope', function (User $user) {
        return in_array($user->email, [
            'taylor@laravel.com',
        ]);
    });
}
```

> 注意：你應該確保在生產環境中將 `APP_ENV` 環境變數更改為 `Production`。 否則，你的 Telescope 偵錯工具將公開可用。

70.3 更新 Telescope

升級到 Telescope 的新主要版本時，務必仔細閱讀 升級指南.

此外，升級到任何新的 Telescope 版本時，你都應該重建 Telescope 實例：

```
php artisan telescope:publish
```

為了使實例保持最新狀態並避免將來的更新中出現問題，可以在應用程式的 `composer.json` 檔案中的 `post-update-cmd` 指令碼新增 `telescope:publish` 命令：

```
{
    "scripts": {
        "post-update-cmd": [
            "@php artisan vendor:publish --tag=laravel-assets --ansi --force"
        ]
    }
}
```

70.4 過濾

70.4.1 單項過濾

你可以通過在 `App\Providers\TelescopeServiceProvider` 類中定義的 `filter` 閉包來過濾 Telescope 記錄的資料。 默認情況下，此回呼會記錄 `local` 環境中的所有資料以及異常、失敗任務、工作排程和帶有受監控標記的資料：

```
use Laravel\Telescope\IncomingEntry;
use Laravel\Telescope\Telescope;

/**
 * 註冊應用服務
 */
public function register(): void
{
    $this->hideSensitiveRequestDetails();

    Telescope::filter(function (IncomingEntry $entry) {
        if ($this->app->environment('local')) {
            return true;
        }

        return $entry->isReportableException() ||
            $entry->isFailedJob() ||
            $entry->isScheduledTask() ||
            $entry->isSlowQuery() ||
            $entry->hasMonitoredTag();
    });
}
```

70.4.2 批次過濾

`filter` 閉包過濾單個條目的資料， 你也可以使用 `filterBatch` 方法註冊一個閉包，該閉包過濾給定請求或控制台命令的所有資料。如果閉包返回 `true`，則所有資料都由 Telescope 記錄：

```
use Illuminate\Support\Collection;
use Laravel\Telescope\IncomingEntry;
use Laravel\Telescope\Telescope;

/**
 *  註冊應用服務
 */
public function register(): void
{
    $this->hideSensitiveRequestDetails();

    Telescope::filterBatch(function (Collection $entries) {
        if ($this->app->environment('local')) {
```

```
            return true;
        }

        return $entries->contains(function (IncomingEntry $entry) {
            return $entry->isReportableException() ||
                $entry->isFailedJob() ||
                $entry->isScheduledTask() ||
                $entry->isSlowQuery() ||
                $entry->hasMonitoredTag();
            });
    });
}
```

70.5 標籤

Telescope 允許你通過 「tag」 搜尋條目。通常，標籤是 Eloquent 模型的類名或經過身份驗證的使用者 ID， 這些標籤會自動新增到條目中。有時，你可能希望將自己的自訂標籤附加到條目中。 你可以使用 `Telescope::tag` 方法。 `tag` 方法接受一個閉包，該閉包應返回一個標籤陣列。返回的標籤將與 Telescope 自動附加到條目的所有標籤合併。你應該在 `App\Providers\TelescopeServiceProvider` 類中的 `register` 方法呼叫 `tag` 方法：

```
use Laravel\Telescope\IncomingEntry;
use Laravel\Telescope\Telescope;

/**
 * 註冊應用服務
 */
public function register(): void
{
    $this->hideSensitiveRequestDetails();

    Telescope::tag(function (IncomingEntry $entry) {
        return $entry->type === 'request'
                    ? ['status:'.$entry->content['response_status']]
                    : [];
    });
 }
```

70.6 可用的觀察者

Telescope 「觀察者」 在執行請求或控制台命令時收集應用資料。你可以在 `config/telescope.php` 組態檔案中自訂啟用的觀察者列表：

```
'watchers' => [
    Watchers\CacheWatcher::class => true,
    Watchers\CommandWatcher::class => true,
    ...
],
```

一些監視器還允許你提供額外的自訂選項：

```
'watchers' => [
    Watchers\QueryWatcher::class => [
        'enabled' => env('TELESCOPE_QUERY_WATCHER', true),
        'slow' => 100,
    ],
    ...
],
```

70.6.1 批次監視器

批次監視器記錄佇列 批次任務 的資訊，包括任務和連接資訊。

70.6.2 快取監視器

當快取鍵被命中、未命中、更新和刪除時，快取監視器會記錄資料。

70.6.3 命令監視器

只要執行 Artisan 命令，命令監視器就會記錄參數、選項、退出碼和輸出。如果你想排除監視器記錄的某些命令，你可以在 `config/telescope.php` 檔案的 `ignore` 選項中指定命令：

```
'watchers' => [
    Watchers\CommandWatcher::class => [
        'enabled' => env('TELESCOPE_COMMAND_WATCHER', true),
        'ignore' => ['key:generate'],
    ],
    ...
],
```

70.6.4 輸出監視器

輸出監視器在 Telescope 中記錄並顯示你的變數輸出。使用 Laravel 時，可以使用全域 `dump` 函數輸出變數。必須在瀏覽器中打開資料監視器選項卡，才能進行輸出變數，否則監視器將忽略此次輸出。

70.6.5 事件監視器

事件監視器記錄應用分發的所有 事件 的有效負載、監聽器和廣播資料。事件監視器忽略了 Laravel 框架的內部事件。

70.6.6 異常監視器

異常監視器記錄應用拋出的任何可報告異常的資料和堆疊跟蹤。

70.6.7 Gate（攔截）監視器

Gate 監視器記錄你的應用的 gate 和策略 檢查的資料和結果。如果你希望將某些屬性排除在監視器的記錄之外，你可 `config/telescope.php` 檔案的 `ignore_abilities` 選項中指定它們：

```
'watchers' => [
    Watchers\GateWatcher::class => [
        'enabled' => env('TELESCOPE_GATE_WATCHER', true),
        'ignore_abilities' => ['viewNova'],
    ],
    ...
],
```

70.6.8 HTTP 客戶端監視器

HTTP 客戶端監視器記錄你的應用程式發出的傳出 HTTP 客戶端請求。

70.6.9 任務監視器

任務監視器記錄應用程式分發的任何 任務 的資料和狀態。

70.6.10 日誌監視器

日誌監視器記錄應用程式寫入的任何日誌的 日誌資料。

默認情況下，Telescope 將只記錄 [錯誤] 等級及以上的日誌。但是，你可以修改應用程式的 `config/tescope.php` 組態檔案中的 `level` 選項來修改此行為：

```
'watchers' => [
    Watchers\LogWatcher::class => [
        'enabled' => env('TELESCOPE_LOG_WATCHER', true),
        'level' => 'debug',
    ],

    // ...
],
```

70.6.11 郵件監視器

郵件監視器允許你查看應用傳送的 郵件 及其相關資料的瀏覽器內預覽。你也可以將該電子郵件下載為 `.eml` 檔案。

70.6.12 模型監視器

每當調度 Eloquent 的 模型事件 時，模型監視器就會記錄模型更改。你可以通過監視器的 `events` 選項指定應記錄哪些模型事件：

```
'watchers' => [
    Watchers\ModelWatcher::class => [
        'enabled' => env('TELESCOPE_MODEL_WATCHER', true),
        'events' => ['eloquent.created*', 'eloquent.updated*'],
    ],
    ...
],
```

如果你想記錄在給定請求期間融合的模型數量，請啟用 `hydrations` 選項：

```
'watchers' => [
    Watchers\ModelWatcher::class => [
        'enabled' => env('TELESCOPE_MODEL_WATCHER', true),
        'events' => ['eloquent.created*', 'eloquent.updated*'],
        'hydrations' => true,
    ],
    ...
],
```

70.6.13 消息通知監視器

消息通知監聽器記錄你的應用程式傳送的所有 消息通知 。如果通知觸發了電子郵件並且你啟用了郵件監聽器，則電子郵件也可以在郵件監視器螢幕上進行預覽。

70.6.14 資料查詢監視器

資料查詢監視器記錄應用程式執行的所有查詢的原始 SQL、繫結和執行時間。監視器還將任何慢於 100 毫秒的查詢標記為 `slow`。你可以使用監視器的 `slow` 選項自訂慢查詢閾值：

```
'watchers' => [
    Watchers\QueryWatcher::class => [
        'enabled' => env('TELESCOPE_QUERY_WATCHER', true),
        'slow' => 50,
    ],
    ...
],
```

70.6.15 Redis 監視器

Redis 監視器記錄你的應用程式執行的所有 Redis 命令。如果你使用 Redis 進行快取，Redis 監視器也會記錄快取命令。

70.6.16 請求監視器

請求監視器記錄與應用程式處理的任何請求相關聯的請求、要求標頭、 session 和響應資料。你可以通過 `size_limit`（以 KB 為單位）選項限制記錄的響應資料：

```
'watchers' => [
    Watchers\RequestWatcher::class => [
        'enabled' => env('TELESCOPE_REQUEST_WATCHER', true),
        'size_limit' => env('TELESCOPE_RESPONSE_SIZE_LIMIT', 64),
    ],
    ...
],
```

70.6.17 定時任務監視器

定時任務監視器記錄應用程式運行的任何 工作排程 的命令和輸出。

70.6.18 檢視監視器

檢視監視器記錄渲染檢視時使用的 檢視 名稱、路徑、資料和「composer」元件。

70.7 顯示使用者頭像

Telescope 儀表盤顯示保存給定條目時會有登錄使用者的使用者頭像。 默認情況下，Telescope 將使用 Gravatar Web 服務檢索頭像。 但是，你可以通過在 `App\Providers\TelescopeServiceProvider` 類中註冊一個回呼來自訂頭像 URL。 回呼將收到使用者的 ID 和電子郵件地址，並應返回使用者的頭像 URL：

```
use App\Models\User;
use Laravel\Telescope\Telescope;

/**
 * 註冊應用服務
 */
public function register(): void
{
    // ...
```

```
    Telescope::avatar(function (string $id, string $email) {
        return '/avatars/'.User::find($id)->avatar_path;
    });
}
```