Laravel 10 官方文件

安全相關

使用者認證 | 使用者授權 | Email 認證 | 加密解密 | 雜湊 | 重設密碼

資料庫

快速入門 | 查詢生成器 | 分頁 | 遷移 | 資料填充 | Redis

Eloquent ORM

快速入門 | 關聯 | 集合 | 屬性修改器 | API 資源 | 序列化 | 資料工廠

測試相關

入門 | HTTP 測試 | 命令列測試 | Laravel Dusk | 資料庫測試 | Mocking

官方擴展包

Envoy 部署工具 | Fortify 與前端無關的身份認證後端實現 | Horizon 佇列管理工具 | Octane 加速引擎 | Passport OAuth 認證 | Pennant 測試新功能 | Pint 程式碼風格 | Sanctum API 授權 | Scout 全文搜尋 | Socialite 第三方登入 | Telescope 偵錯工具

目錄

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

controller 方法	策略方法
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函數來處理模型事件，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:castArtisan 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;
    });
}
```