

一、序言

新版特性



Laravel 5.6 在 Laravel 5.5 的基础上继续进行优化，包括日志系统、单机任务调度、模型序列化优化、动态频率限制、广播频道类、API 资源控制器生成、Eloquent 日期格式化优化、Blade 组件别名、Argon2 密码哈希支持、引入 Collision 扩展包等等等等。此外，所有的前端脚手架代码都已升级到 Bootstrap 4，Laravel 底层使用的 Symfony 组件都已升级到 Symfony~4.0 版本。

Laravel 5.6 版本的发布恰逢 Spark 6.0 的发布，所以这也是自 Laravel Spark 发布以来第一次重大版本升级。Spark 6.0 为 Stripe 和 Braintree 引入了按座定价功能，以及本地化、Bootstrap 4、增强 UI 和 Stripe Elements 支持。

注：本文档只是概述了框架大部分引人注目的重要升级，要了解详细升级日志可以查看 GitHub 上到 [change logs](#)。

日志优化

Laravel 5.6 带来了日志系统的重大升级，所有日志配置都存放在新的 `config/logging.php` 配置文件，你现在可以轻松构建发送日志消息到多个处理器的日志“堆栈”。例如，你可以发送所有 `debug` 级别消息到系统日志同时发送 `error` 级别消息到 Slack 以便团队成员可以快速响应：

```
'channels' => [
    'stack' => [
        'driver' => 'stack',
        'channels' => ['syslog', 'slack'],
    ],
],
```

此外，现在可以使用日志系统的新“tap”功能很轻松地自定义已存在的日志频道。想要了解更多细节，请查看[完整日志文档](#)。

单机任务调度

注：要使用这个新特性，必须使用 `memcached` 或 `redis` 缓存驱动作为应用默认缓存驱动。此外，所有服务器必须和同一个中心缓存服务器进行通信。

如果你的应用运行在多个服务器上，现在可以限定只在一台机器上运行调度任务。例如，假设你有一个在每周五晚上生成新报告的调度任务，如果任务调度器运行在三个服务器上，这个调度任务就会在三台机器上运行并生成同样的报告三次，这样很不优雅，甚至很糟糕！

要指定任务只在一台机器上运行，可以在定义调度任务时使用 `onOneServer` 方法，第一台获取到任务的机器会给这个任务上一把原子级别的锁来阻止其他服务器同时运行同一个任务：

```
$schedule->command('report:generate')
    ->fridays()
    ->at('17:00')
    ->onOneServer();
```

动态频率限制

当我们在之前版本的路由群组中指定了 [频率限制](#) 后，必须要硬编码最大请求次数：

```
Route::middleware('auth:api', 'throttle:60,1')->group(function () {
    Route::get('/user', function () {
        //
    });
});
```

在 Laravel 5.6 中，你可以基于认证用户模型属性指定一个动态的最大请求次数，如果 `User` 模型包含 `rate_limit` 属性，可以将属性名传递给 `throttle` 中间件，以便用于计算最大请求次数计数：

```
Route::middleware('auth:api', 'throttle:rate_limit,1')->group(function () {
    Route::get('/user', function () {
        //
    });
});
```

广播频道类

如果你的应用消费多个不同的频道，`routes/channels.php` 文件可能会变得很臃肿，所以，作为使用闭包来授权频道的替代方案，你现在可以使用频道类。要生成一个频道类，可以使用 Artisan 命令 `make:channel`。该命令会将新生成的频道类存放到 `app/Broadcasting` 目录下：

```
php artisan make:channel OrderChannel
```

接下来，在 `routes/channels.php` 文件中注册这个频道类：

```
use App\Broadcasting\OrderChannel;

Broadcast::channel('order.{order}', OrderChannel::class);
```

最后，可以将频道的授权逻辑放到频道类的 `join` 方法。`join` 方法中的代码等同于之前位于频道授权闭包中的处理逻辑。当然，你还可以使用频道模型绑定：

```
<?php

namespace App\Broadcasting;

use App\User;
use App\Order;

class OrderChannel
{
    /**
     * Create a new channel instance.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Authenticate the user's access to the channel.
     *
     * @param \App\User $user
     * @param \App\Order $order
     * @return array|bool
     */
    public function join(User $user, Order $order)
    {
        return $user->id === $order->user_id;
    }
}
```

API 控制器生成

声明被 API 消费的资源控制器时，通常你会排除输出 HTML 模板的路由，例如 `create` 和 `edit`，要生成不包含这些方法的资源控制器，可以在使用 Artisan 命令执行 `make:controller` 时使用 `--api` 开关：

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

模型序列化优化

在之前版本的 Laravel 中，队列中的模型在反序列化后不会带有完整的已加载关联关系。在 Laravel 5.6 中，模型上已加载的关联关系在队列任务被处理时会自动进行重新加载。

Eloquent 日期转化

现在你可以单独自定义 Eloquent 日期字段转化格式了，开始之前，需要在转化声明中指定目标日期格式。指定好之后，该格式就会在模型序列化为数组/JSON 时使用：

```
protected $casts = [
    'birthday' => 'date:Y-m-d',
    'joined_at' => 'datetime:Y-m-d H:00',
];
```

Blade 组件别名

如果你的 Blade 组件存放在子目录中，现在起可以为它们起个别名以便使用。例如，假设一个 Blade 组件存放 在 `resources/views/components/alert.blade.php`，你可以使用 `component` 方法将这个组件名从 `components.alert` 改为别名 `alert`：

```
Blade::component('components.alert', 'alert');
```

组件起了别名之后，就可以使用别名来渲染：

```
@component('alert')
    You are not allowed to access this resource!
@endcomponent
```

或者，如果组件没有额外插槽，可以使用组件别名作为 Blade 指令：

```
@alert
    You are not allowed to access this resource!
@endalert
```

Argon2 密码哈希

如果你在构建一个基于 PHP 7.2.0+ 的应用，Laravel 现在可以支持通过 Argon2 算法进行密码哈希，默认的应用哈希驱动通过新增的 `config/hashing.php` 配置文件来控制。

UUID 方法

Laravel 5.6 引入了两个新的方法来生成 UUID：`Str::uuid` 和 `Str::orderedUuid`，`orderedUuid` 方法会生成一个时间戳最靠前的 UUID，通过诸如 MySQL 的数据库来索引，更简单，也更高效。两个方法都会返回 `Ramsey\Uuid\Uuid` 对象：

```
use Illuminate\Support\Str;
return (string) Str::uuid();
return (string) Str::orderedUuid();
```

Collision

默认的 `laravel/laravel` 应用现在为 Collision 包含了一个 `dev` Composer 依赖，这个扩展包在通过命令行与 Laravel 应用交互时提供了美观的错误报告：

```
nunomaduro@mac: ~/dev/laravel-zero
→ laravel-zero git:(release-4.0.0) ✘ php application bootstrap
Error : Call to undefined method App\Commands\BootstrapCommand::bootstrap()

at /Users/nunomaduro/dev/laravel-zero/app/Commands/BootstrapCommand.php: 31
27:     * @return void
28:     */
29:     public function handle(): void
30:     {
31:         $this->bootstrap();
32:     }
33:
34:     /**
35:      * Define the command's schedule.
36:      *

Exception trace:
1  App\Commands\BootstrapCommand::handle()
   /Users/nunomaduro/dev/laravel-zero/vendor/illuminate/container/BoundMethod.php : 29
2  call_user_func_array([])
   /Users/nunomaduro/dev/laravel-zero/vendor/illuminate/container/BoundMethod.php : 29

Please use the argument -v to see all trace.
```

Bootstrap 4

所有前端脚手架例如用户登录认证模板和 Vue 示例组件都已经升级到 [Bootstrap 4](#)。默认情况下，生成的分页链接现在也已升级到 Bootstrap 4。

升级指南

预计升级时间：10-30 分钟

PHP

Laravel 5.6 需要 PHP 7.1.3 或更高版本。

更新依赖

在 `composer.json` 中更新 `laravel/framework` 依赖到 `5.6.*`，更新 `fideloper/proxy` 依赖到 `~4.0`。
此外，如果你使用以下官方扩展包，也要升级它们到最新版本：

- Dusk (升级到 `~3.0`)
- Passport (升级到 `~5.0`)
- Scout (升级到 `~4.0`)

当然，不要忘了检查应用所使用的第三方扩展包是否支持 Laravel 5.6，如果需要升级的话也要更新。

Symfony 4

Laravel 底层使用的所有 Symfony 组件都已經升级到 Symfony `~4.0` 版本。如果你在应用代码中使用了 Symfony 组件，需要查看 [Symfony 更新日志](#) 以确认是否需要修改代码。

PHPUnit

需要更新应用的 `phpunit/phpunit` 依赖到 `~7.0`。

数组

`Arr::wrap` 方法

在 Laravel 5.6 中，传递 `null` 到 `Arr::wrap` 方法将会返回空数组。

Artisan

`optimize` 命令

之前版本中废弃的 `optimize` 命令已经被彻底移除。由于 PHP 自身的性能优化，`optimize` 命令已经不能给应用提供显著的性能提升，所以，你需要从 `composer.json` 文件的 `scripts` 部分移除 `php artisan optimize`。

Blade

HTML 实体编码

在之前版本的 Laravel 中，Blade 不会对 HTML 实体进行双重编码。这并不是底层 `htmlspecialchars` 函数的默认行为，而且会在渲染内容或传递内联 JSON 内容到 JavaScript 框架时导致预期之外的结果。

在 Laravel 5.6 中，Blade 以及辅助函数 `e` 默认会对特殊字符进行双重编码，从而与 PHP 底层 `htmlspecialchars` 函数的默认行为保持一致。如果你想要维持不进行双重编码的旧状，可以使用 `Blade::withoutDoubleEncoding` 方法：

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Blade::withoutDoubleEncoding();
    }
}
```

缓存

频率限制器 `tooManyAttempts` 方法

该方法签名中未使用的 `$decayMinutes` 参数被移除。如果你通过自己的实现重写了该方法，也要从方法签名中移除该参数。

数据库

Morph 列的索引顺序

为了获得更好的性能，迁移方法 `morphs` 构建的列索引顺序倒过来了，如果你在某个迁移中使用了 `morphs` 方法，尝试运行该迁移的 `down` 方法时会报错。如果应用还在开发中，可以使用 `migrate:fresh` 命令重构数据库结构；如果应用已经上线，需要显式传递索引名称到 `morphs` 方法。

MigrationRepositoryInterface 新增方法

`MigrationRepositoryInterface` 中新增了一个 `getMigrationsBatches` 方法。如果非常不巧你正在自定义该类的实现，需要添加这个方法的实现。你可以以框架的默认实现作为示例。

Eloquent

`getDateFormat` 方法

`getDateFormat` 方法的可见性从 `protected` 调整为 `public`。

哈希

新配置文件

所有哈希配置现在位于独立的 `config/hashing.php` 配置文件。你可以拷贝一份 [默认配置文件](#) 到你的应用。大多数情况下，我们都会将 `bcrypt` 驱动作为默认驱动。不过，也支持 `argon`。

辅助函数

e

在之前版本的 Laravel 中，Blade（以及辅助函数 `e`）不会对 HTML 实体进行双重编码。这并不是底层 `htmlspecialchars` 函数的默认行为，而且会在渲染内容或传递内联 JSON 内容到 JavaScript 框架时导致预期之外的结果。

在 Laravel 5.6 中，Blade 以及辅助函数 `e` 默认会对特殊字符进行双重编码，从而与 PHP 底层 `htmlspecialchars` 函数的默认行为保持一致。如果你想要维持不进行双重编码的旧状，可以传递 `false` 作为第二个参数到 `e` 函数：

```
<?php echo e($string, false); ?>
```

日志

新配置文件

所有的日志配置现在都存放在独立的 `config/logging.php` 配置文件。你可以拷贝一份 [默认的配置文件](#) 到你的应用然后基于应用需要进行设置。

`log` 和 `log_level` 配置项都可以从配置文件 `config/app.php` 里移除了。

`configureMonologUsing` 方法

如果你在使用 `configureMonologUsing` 方法为应用自定义 `Monolog` 实例，现在需要创建一个 `custom` 日志频道。更多关于如何创建自定义频道的信息，可以查看[完整的日志文档](#)。

日志 `Writer` 类

`Illuminate\Log\Writer` 类被重命名为 `Illuminate\Log\Logger`, 如果你在应用的某个类中对这个类进行了显式的类型提示作为依赖注入, 需要更新该类的引用为新的类名。或者, 作为替代方案, 你可以考虑将类型提示调整为标准的 `Psr\Log\LoggerInterface` 接口。

`Illuminate\Contracts\Logging\Log` 接口

该接口已经被移除, 因为它和 `Psr\Log\LoggerInterface` 接口完全重合, 需要将引用它的地方都调整为 `Psr\Log\LoggerInterface` 接口。

邮件

`withSwiftMessage` 回调

在之前版本的 Laravel 中, 使用 `withSwiftMessage` 注册的 Swift 消息自定义回调函数在内容已经被编码并添加到消息后被调用。这些回调现在在内容被添加前调用, 从而允许你自定义编码以及其他消息选项。

分页

Bootstrap 4

分页器生成的分页链接现在默认使用 Bootstrap 4, 要让分页器生成 Bootstrap 3 链接, 需要在 `AppServiceProvider` 的 `boot` 方法中调用 `Paginator::useBootstrapThree` 方法:

```
<?php

namespace App\Providers;

use Illuminate\Pagination\Paginator;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Paginator::useBootstrapThree();
    }
}
```

资源

`original` 属性

资源响应的 `original` 属性现在被设置为原始模型而不是 JSON 字符串/数组。这样在测试中就可以更好地检查响应的模型。

路由

返回新创建的模型

从路由中直接返回新创建的 Eloquent 模型时, 响应状态码由 `200` 调整为 `201`, 如果应用的任意相应测试显式期望 `200` 状态码, 那么需要将它们修改为 `201`。

信任代理

由于信任代码功能使用的底层 Symfony HttpFoundation 有改动, 所以必须微调 `App\Http\Middleware\TrustProxies` 中间件。

`$headers` 属性之前是个数组, 现在是一个接收几个不同值的属性。例如, 要信任所有转发头, 需要像这样更新 `$headers` 属性:

```
use Illuminate\Http\Request;

/**
 * The headers that should be used to detect proxies.
 *
 * @var string
 */
protected $headers = Request::HEADER_X_FORWARDED_ALL;
```

更多关于 `$headers` 变量值的信息可以查看完整的[信任代理文档](#)。

验证

`ValidatesWhenResolved` 接口

为了避免和 `$request->validate()` 方法冲突, `ValidatesWhenResolved` 接口/trait 的 `validate` 方法被重命名为 `validateResolved`。

杂项

我们还鼓励你查看 [laravel/laravel 代码仓库](#) 的更新日志。尽管其中的很多更新不是必须的，但是你可以将应用中的这些文件与代码仓库保持同步。其中的一些更新已经在这篇升级指南中覆盖到了，但是还有很多其他的小更新比如配置文件或注释的微调，就不会一一指出。你可以通过 [GitHub 比较工具](#) 轻松查看变更以便选择那些对你而言更为重要的更新。

二、快速入门

安装配置

服务器要求

Laravel 框架对 PHP 版本和扩展有一定要求，不过这些要求 [Laravel Homestead](#) 都已经满足了，不过如果你没有使用 Homestead 的话（那真是一件很遗憾的事情），有必要了解下这些以便确认自己的环境满足要求：

- PHP >= 7.1.3
- PHP OpenSSL 扩展
- PHP PDO 扩展
- PHP Mbstring 扩展
- PHP Tokenizer 扩展
- PHP XML 扩展
- PHP Ctype 扩展
- PHP JSON 扩展

满足以上需求之后，就可以开始安装 Laravel 了。

安装 Laravel

Laravel 使用 [Composer](#) 管理依赖，所以，安装之前确保已经在机器上安装了 Composer（如果尚未安装的话参考[这份文档](#)去安装吧）。

通过 Laravel 安装器

首先，通过 Composer 安装 Laravel 安装器：

```
composer global require "laravel/installer"
```

确保 `$HOME/.composer/vendor/bin` 在系统路径中（Mac 中对应路径是 `~/.composer/vendor/bin`，Windows 对应路径是 `~/AppData/Roaming/Composer/vendor/bin`，其中 `~` 表示当前用户家目录），否则不能在命令行任意路径下调用 `laravel` 命令。
安装完成后，通过简单的 `laravel new` 命令即可在当前目录下创建一个新的 Laravel 应用，例如，`laravel new blog` 将会创建一个名为 `blog` 的新应用，且包含所有 Laravel 依赖。该安装方法比通过 Composer 安装要快很多：

```
laravel new blog
```

如果之前已经安装过旧版本的 Laravel 安装器，需要更新后才能安装最新的 Laravel 5.6 框架应用：

```
composer global update
```

通过 Composer Create-Project

你还可以在终端中通过 Composer 的 `create-project` 命令来安装 Laravel 应用：

```
composer create-project --prefer-dist laravel/laravel blog
```

如果要下载安装 Laravel 其他版本应用，比如 5.5 版本，可以使用这个命令：

```
composer create-project --prefer-dist laravel/laravel blog 5.5.*.
```

本地开发服务器

如果你在本地安装了 PHP，并且想要使用 PHP 内置的开发环境服务器为应用提供服务，可以使用 Artisan 命令 `serve`：

```
php artisan serve
```

该命令将会在本地启动开发环境服务器，这样在浏览器中通过 `http://localhost:8000` 即可访问应用：

Laravel

[DOCUMENTATION](#)[LARACASTS](#)[NEWS](#)[FORGE](#)[GITHUB](#)

当然，更强大的本地开发环境选择还是 Homestead 和 Valet。

配置 Laravel

初始化配置

公共目录

安装完 Laravel 后，需要将 Web 服务器的 document/web 根目录指向 Laravel 应用的 public 目录，该目录下的 index.php 文件作为前端控制器（单一入口），所有 HTTP 请求都会通过该文件进入应用。

配置文件

Laravel 框架的所有配置文件都存放在 config 目录下，所有的配置项都有注释，所以你可以轻松遍览这些配置文件以便熟悉所有配置项。

目录权限

安装完 Laravel 后，需要配置一些目录的读写权限：storage 和 bootstrap/cache 目录对 Web 服务器指定的用户而言应该是可写的，否则 Laravel 应用将不能正常运行。如果你使用 Homestead 虚拟机做为开发环境，这些权限已经设置好了。

应用 key

接下来要做的事情就是将应用的 key (APP_KEY) 设置为一个随机字符串，如果你是通过 Composer 或者 Laravel 安装器安装的话，该 key 的值已经通过 php artisan key:generate 命令生成好了。

通常，该字符串应该是 32 位长，通过 .env 文件中的 APP_KEY 进行配置，如果你还没有将 .env.example 文件重命名为 .env，现在立即这样做。

如果应用 key 没有被设置，用户 Session 和其它加密数据将会有安全隐患！

更多配置

Laravel 几乎不再需要其它任何配置就可以正常使用了，不过，你最好再看看 config/app.php 文件，其中包含了一些基于应用可能需要进行改变的配置，比如 timezone 和 locale（分别用于配置时区和本地化）。

你可能还想要配置 Laravel 的一些其它组件，比如缓存、数据库、Session 等，关于这些我们将会在后续文档一一探讨。

Web 服务器配置

关于虚拟主机的配置（映射域名到 Laravel 应用目录）略过，如果了解细节可参考[这篇教程](#)，当然也可以留待下一篇讲 Homestead 和 Valet 再去了解。本文只探讨如何美化 URL 让其更具有可读性。

Apache

框架中自带的 public/.htaccess 文件支持隐藏 URL 中的 index.php，如过你的 Laravel 应用使用 Apache 作为服务器，需要先确保 Apache 启用了 mod_rewrite 模块以支持 .htaccess 解析。

如果 Laravel 自带的 .htaccess 文件不起作用，试试将其中内容做如下替换：

```
Options +FollowSymLinks
RewriteEngine On

RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^ index.php [L]
```

Nginx

如果你使用的是 Nginx，使用如下站点配置指令就可以支持 URL 美化：

```
location / {
    try_files $uri $uri/ /index.php?$query_string;
}
```

当然，使用 Homestead 或 Valet 的话，以上配置已经为你配置好，无需额外操作。

环境配置

基于应用运行的环境不同设置不同的配置值能够给我们开发带来极大的方便，比如，我们通常在本地和线上环境配置不同的缓存驱动，这一功能在 Laravel 中很容易实现。

Laravel 使用 PHP 扩展库 DotEnv 来实现这一功能，在新安装的 Laravel 中，根目录下有一个 .env.example 文件，如果 Laravel 是通过 Composer 安装的，那么该文件已经被重命名为 .env，否则的话你要自己手动重命名该文件。

不要试图将 `.env` 文件提交到版本控制系统（如 Git 或 Svn）中，一方面，开发环境和线上环境配置值不一样，提交没有意义，更重要的是，`.env` 包含了很多应用敏感信息，如数据库用户名及密码等，如果不慎将代码提交到 Github 公开仓库，后果将不堪设想！

如果你是在一个团队中进行开发，则需要将 `.env.example` 文件随你的应用代码一起提交到源码控制中：将一些配置值以占位符的方式放置在 `.env.example` 文件中，这样其他开发者就会很清楚运行你的应用需要配置哪些环境变量。

还可以创建一个 `.env.testing` 文件，该文件会在运行 PHPUnit 测试或执行带有 `--env=testing` 选项的 Artisan 命令时覆盖从 `.env` 文件读取的值。

注：`.env` 文件中的所有变量都可以被外部环境变量覆盖，例如服务器级别或系统级别的环境变量。

获取环境变量配置值

应用每次接受请求时，`.env` 中列出的所有配置及其对应值都会被载入到 PHP 超全局变量 `$_ENV` 中，然后你就可以在应用中通过辅助函数 `env` 来获取这些配置值。实际上，如果你去查看 Laravel 的配置文件，就会发现很多地方已经在使用这个辅助函数了：

```
'debug' => env('APP_DEBUG', false),
```

传递到 `env` 函数的第二个参数是默认值，如果环境变量没有被配置将会使用该默认值。

判断当前应用环境

当前应用环境由 `.env` 文件中的 `APP_ENV` 变量决定，你可以通过 `App` 门面上的 `environment` 方法来访问其值：

```
$environment = App::environment();
```

你也可以向 `environment` 方法传递参数来判断当前环境是否匹配给定值，如果需要的话你甚至可以传递多个值。如果当前环境与给定值匹配，该方法返回 `true`：

```
if (App::environment('local')) {
    // The environment is local
}

if (App::environment('local', 'staging')) {
    // The environment is either local OR staging...
}
```

注：当前应用环境判断可以被服务器级别环境变量 `APP_ENV` 覆盖。当你需要在不同环境配置间共享同一应用时很有用，你可以在服务器配置中设置一个给定主机来匹配给定环境。

访问配置值

你可以使用全局辅助函数 `config` 在应用代码的任意位置访问配置值，配置值以文件名+“.”+配置项的方式进行访问，当配置项没有被配置的时候返回默认值：

```
$value = config('app.timezone');
```

如果要在运行时设置配置值，传递数组参数到 `config` 方法即可：

```
config(['app.timezone' => 'Asia/Shanghai']);
```

缓存配置文件

为了给应用加速，你可以使用 Artisan 命令 `config:cache` 将所有配置文件的配置缓存到单个文件里，这将会将所有配置选项合并到单个文件从而被框架快速加载。

应用每次上线，都要运行一次 `php artisan config:cache`，但是在本地开发时，没必要经常运行该命令，因为配置值经常会改变。

注：如果在部署过程中执行 `config:cache` 命令，需要确保只在配置文件中调用了 `env` 方法。一旦配置文件被缓存后，`.env` 文件将不能被加载，所有对 `env` 函数的调用都会返回 `null`。

维护模式

当你的应用处于维护模式时，所有对应用的请求都应该返回同一个自定义视图。这一功能在对应用进行升级或者维护时，使得“关闭”站点变得轻而易举。对维护模式的判断代码位于应用默认的中间件栈中，如果应用处于维护模式，访问应用时状态码为 503 的 `MaintenanceModeException` 将会被抛出。

要开启维护模式，关闭站点，只需执行 Artisan 命令 `down` 即可：

```
php artisan down
```

还可以提供 `message` 和 `retry` 选项给 `down` 命令。`message` 的值用于显示或记录自定义消息，而 `retry` 的值用于设置 HTTP 请求头的 `Retry-After`：

```
php artisan down --message="Upgrading Database" --retry=60
```

要关闭维护模式，开启站点，对应的 Artisan 命令是 `up`：

```
php artisan up
```

注：你可以通过定义自己的模板来定制默认的维护模式模板，自定义模板视图位于 `resources/views/errors/503.blade.php`。

维护模式 & 队列

当你的站点处于维护模式中时，所有的队列任务都不会执行；当应用退出维护模式这些任务才会被继续正常处理。

维护模式的替代方案

由于维护模式命令的执行需要几秒时间，你可以考虑使用 `Envoyer` 实现零秒下线作为替代方案。

目录结构

简介

Laravel 默认的目录结构试图为不管是大型应用还是小型应用提供一个良好的起点。当然，你也可以按照自己的喜好重新组织应用的目录结构，因为 Laravel 对于指定类在何处被加载没有任何限制 —— 只要 Composer 可以自动载入它们即可。

Models 目录在哪里？

许多初学者都会困惑 Laravel 为什么没有提供 `models` 目录，我可以负责任的告诉大家，这是故意的。因为 `models` 这个词对不同人而言有不同的含义，容易造成歧义，有些开发者认为应用的模型指的是业务逻辑，另外一些人则认为模型指的是与关联数据库的交互。

正是因为这个原因，我们默认将 Eloquent 的模型直接放置到 `app` 目录下，开发者可以自行选择放置的位置。

这是 Laravel 框架作者的想法，不过对于国内开发者，尤其是 PHP 开发者来说，`models` 目录用于存放与数据库交互的模型类应该没有什么异议，而业务逻辑应该放到 `services` 这种目录之下。所以推荐大家在生成模型类的时候指定生成到 `app/Models` 目录下：

```
php artisan make:model Models/Test
```

The screenshot shows a code editor with a sidebar labeled "FOLDERS". The sidebar lists the following directory structure:

- laravel55
- app
 - Console
 - Exceptions
 - Http
 - Models
 - Test.php
 - Providers
 - User.php
- bootstrap
- config
- database
- public
- resources
- routes
- storage
- tests
- vendor

The main editor window displays the contents of the `Test.php` file:

```

Test.php
1 <?php
2
3 namespace App\Models;
4
5 use Illuminate\Database\Eloquent\Model;
6
7 class Test extends Model
8 {
9     //
10 }
11

```

根目录

App 目录

`app` 目录包含了应用的核心代码，注意不是框架的核心代码，框架的核心代码在 `/vendor/laravel/framework` 里面，此外你为应用编写的代码绝大多数也会放在这里，当然，如果你基于 Composer 做了 PHP 组件化开发的话，这里面存放的恐怕也只有一些入口性的代码了；

Bootstrap 目录

`bootstrap` 目录包含了少许文件，用于框架的启动和自动载入配置，还有一个 `cache` 文件夹，里面包含了框架为提升性能所生成的文件，如路由和服务缓存文件；

Config 目录

`config` 目录包含了应用所有的配置文件，建议通读一遍这些配置文件以便熟悉 Laravel 所有默认配置项；

Database 目录

`database` 目录包含了数据库迁移文件及填充文件，如果有使用 SQLite 的话，你还可以将其作为 SQLite 数据库存放目录；

Public 目录

`public` 目录包含了应用入口文件 `index.php` 和前端资源文件（图片、JavaScript、CSS 等），该目录也是 Apache 或 Nginx 等 Web 服务器所指向的应用根目录，这样做好处是隔离了应用核心文件直接暴露于 Web 根目录之下，如果权限系统没做好或服务器配置有漏洞的话，很可能导致应用敏感文件被黑客窃取，进而对网站安全造成威胁；

Resources 目录

`resources` 目录包含了应用视图文件和未编译的原生前端资源文件（LESS、SASS、JavaScript），以及本地化语言文件；

Routes 目录

`routes` 目录包含了应用定义的所有路由。Laravel 默认提供了四个路由文件用于给不同的入口使用：`web.php`、`api.php`、`console.php` 和 `channels.php`。

`web.php` 文件包含的路由都位于 `RouteServiceProvider` 所定义的 `web` 中间件组约束之内，因而支持 Session、CSRF 保护以及 Cookie 加密功能，如果应用无需提供无状态的、RESTful 风格的 API，那么路由基本上都要定义在 `web.php` 文件中。

`api.php` 文件包含的路由位于 `api` 中间件组约束之内，支持频率限制功能，这些路由是无状态的，所以请求通过这些路由进入应用需要通过 token 进行认证并且不能访问 Session 状态。

`console.php` 文件用于定义所有基于闭包的控制台命令，每个闭包都被绑定到一个控制台命令并且允许与命令行 IO 方法进行交互，尽管这个文件并不定义 HTTP 路由，但是它定义了基于控制台的应用入口（路由）。

`channels.php` 文件用于注册应用支持的所有事件广播频道。

Storage 目录

`storage` 目录包含了编译后的 Blade 模板、基于文件的 Session、文件缓存，以及其它由框架生成的文件，该目录被细分为成 `app`、`framework` 和 `logs` 子目录，`app` 目录用于存放应用生成的文件，`framework` 目录用于存放框架生成的文件和缓存，最后，`logs` 目录存放的是应用的日志文件。
`storage/app/public` 目录用于存储用户生成的文件，比如可以被公开访问的用户头像，要达到被 Web 用户访问的目的，你还需要在 `public`（应用根目录下的 `public` 目录）目录下生成一个软连接 `storage` 指向这个目录。你可以通过 `php artisan storage:link` 命令生成这个软链接。

Tests 目录

`tests` 目录包含自动化测试文件，其中默认已经提供了一个开箱即用的 `PHPUnit` 示例；每一个测试类都要以 `Test` 开头，你可以通过 `phpunit` 或 `php vendor/bin/phpunit` 命令来运行测试。

Vendor 目录

`vendor` 目录包含了应用所有通过 Composer 加载的依赖。

App 目录

应用的核心代码位于 `app` 目录下，默认情况下，该目录位于命名空间 `App` 下，并且被 Composer 通过 `PSR-4 自动载入标准` 自动加载。
`app` 目录下包含多个子目录，如 `Console`、`Http`、`Providers` 等。`Console` 和 `Http` 目录提供了进入应用核心的 API，HTTP 协议和 CLI 是和应用进行交互的两种机制，但实际上并不包含应用逻辑。换句话说，它们只是两个向应用发送命令的方式。`Console` 目录包含了所有开发者编写的 Artisan 命令，`Http` 目录包含了控制器、中间件和请求等。

其他目录会在你通过 Artisan 命令 `make` 生成相应类的时候自动生成到 `app` 目录下。例如，`app/Jobs` 目录直到你执行 `make:job` 命令生成任务类时才会出现在 `app` 目录下。

注：`app` 目录中的很多类都可以通过 Artisan 命令生成，要查看所有有效的命令，可以在终端中运行 `php artisan list make` 命令。

Broadcasting 目录

`Broadcasting` 目录包含了应用所需的所有广播频道类，这些类通过 `make:channel` 命令生成。该目录默认不存在，但是当你通过命令第一次创建频道类时会自动生成。想要了解更多关于频道的信息，可以查看 [事件广播文档](#)。

Console 目录

`Console` 目录包含应用所有自定义的 Artisan 命令，这些命令类可以使用 `make:command` 命令生成。该目录下还有 `Console/Kernel` 类，在这里可以注册自定义的 Artisan 命令以及定义 [调度任务](#)。

Events 目录

这个目录默认不存在，但是可以通过 `event:generate` 和 `make:event` 命令创建。该目录用于存放 [事件类](#)。事件类用于告知应用其他部分某个事件发生情况并提供灵活的、解耦的处理机制。

Exceptions 目录

`Exceptions` 目录包含应用的异常处理器，同时还是处理应用抛出的任何异常的好地方。如果你想要自定义异常如何记录或渲染，需要编辑该目录下的 `Handler` 类。

Http 目录

`Http` 目录包含了控制器、中间件以及表单请求等，几乎所有通过 Web 进入应用的请求处理都在这里进行。

Jobs 目录

该目录默认不存在，可以通过执行 `make:job` 命令生成，`Jobs` 目录用于存放队列任务，应用中的任务可以被 [推送到队列](#)，也可以在当前请求生命周期内同步执行。同步执行的任务有时也被看作命令，因为它们实现了 [命令模式](#)。

Listeners 目录

这个目录默认不存在，可以通过执行 `event:generate` 和 `make:listener` 命令创建。`Listeners` 目录包含处理事件的类（事件监听器），事件监听器接收一个事件并提供对该事件发生后的响应逻辑，例如，`UserRegistered` 事件可以被 `SendWelcomeEmail` 监听器处理。

Mail 目录

这个目录默认不存在，但是可以通过执行 `make:mail` 命令生成，`Mail` 目录包含应用所有邮件相关类，邮件对象允许你在一个地方封装构建邮件所需的所有业务逻辑，然后使用 `Mail::send` 方法发送邮件。

Notifications 目录

这个目录默认不存在，你可以通过执行 `make:notification` 命令连带创建，`Notifications` 目录包含应用发送的所有通知，比如事件发生通知。 Laravel 的通知功能将通知发送和通知驱动解耦，你可以通过邮件，也可以通过 Slack、短信或者数据库发送通知。

Policies 目录

这个目录默认不存在，你可以通过执行 `make:policy` 命令生成策略类来创建，`Policies` 目录包含了应用所有的授权策略类，策略用于判断某个用户是否有权限去访问指定资源。更多详情，请查看 [授权文档](#)。

Providers 目录

`Providers` 目录包含应用的所有 [服务提供者](#)。服务提供者在应用启动过程中绑定服务到容器、注册事件以及执行其他任务为即将到来的请求处理做好准备工作。

在新安装的 Laravel 应用中，该目录已经包含了一些服务提供者，你可以按需添加自己的服务提供者到该目录。

Rules 目录

该目录默认不存在，但是会伴随你执行 Artisan 命令 `make:rule` 自动生成。`Rules` 目录包含应用的自定义验证规则对象，这些规则用于在单个对象中封装复杂的验证逻辑，想要了解更多的话，请参考 [验证文档](#)。

Homestead

简介

Laravel 为开发者提供了一套完善的重量级本地开发环境——Laravel Homestead。

Laravel Homestead 实际是一个打包好各种 Laravel 开发所需软件和工具的 Vagrant 盒子（关于 Vagrant 盒子的释义请参考 [Vagrant 官方文档](#)），该盒子为我们提供了一个优秀的开发环境，有了它，我们不再需要在本地环境安装 PHP、Composer、Nginx、MySQL、Memcached、Redis、Node 等其它工具软件，我们也完全不用再担心误操作搞乱操作系统——因为 Vagrant 盒子是一次性的，如果出现错误，可以在数分钟内销毁并重新创建该 Vagrant 盒子！

为什么说它是重量级的开发环境呢？谁用谁知道，要使用上这个开发环境，你需要安装 Vagrant、VirtualBox，下载 Homestead 对应的 Vagrant 盒子，经历一系列下载、安装和配置之后才能使用（可能需要花费数小时），如果是 Mac 或 Linux 系统可能还比较顺利，如果是 Windows 系统的话就得先烧柱香拜拜菩萨再开始，保佑一切顺利，哈哈，开玩笑啦，不过 Windows 确实相对而言出问题的概率比较大。

有人要说了，听上去这么复杂，我就不用了，不过相信我吧，这点时间都会在日后因为 Homestead 强大完善的功能在开发过程中补回来，正所谓磨刀不误砍柴功，Homestead 不仅为你提供了一整套日后开发所需要的工具，而且与 Laravel 配置文件默认配置无缝结合，省去了很多配置的麻烦，此外，如果是在团队中开发的话，Homestead 还为你们提供了一致的开发环境，避免因为不同开发人员使用的工具软件版本不同造成线上的问题，这三个理由，我想足够可以说服你了。

当然，如果你只是想简单尝鲜，不使用 Homestead 也无可厚非，毕竟 Mac 下有 Valet，Windows 下则可以使用 Xampp 之类的便捷工具包，但是如果是工程化开发，走正规军路子还是推荐使用 Homestead。

注：如果你使用的是 Windows，需要开启系统的硬件虚拟化（VT-x），这通常可以通过 BIOS 来开启。如果你是在 UEFI 系统上使用 Hyper-V，则需要关闭 Hyper-V 以便可以访问 VT-x。

预装软件

Homestead 可以运行在 Windows、Mac 以及 Linux 等主流操作系统上，预装的软件和工具列表如下：

- Ubuntu 16.04
- Git
- PHP 7.2
- PHP 7.1
- PHP 7.0
- PHP 5.6
- Nginx
- Apache (可选)
- MySQL
- MariaDB (可选)
- SQLite3
- PostgresSQL
- Composer
- Node (With Yarn, Bower, Grunt, and Gulp)
- Redis
- Memcached
- Beanstalkd
- Mailhog
- Elasticsearch (可选)
- ngrok

安装 & 设置

首次安装

在使用 Homestead 之前，需要先安装 Virtual Box 5.2、VMWare、Parallels 或 Hyper-V（四选一，我们通常选择 VirtualBox，因为只有它是免费的）以及 Vagrant，所有这些软件都为常用操作系统提供了一个便于使用的可视化安装器，通过安装界面引导就可以完成安装。

要使用 VMware 的话，需要购买 VMware Fusion (Mac) / Workstation (Windows) 以及 VMware Vagrant 插件，尽管不便宜，一套下来要 1000 多块人民币，但是 VMware 可以提供更好的性能和体验（废话，不然谁用，不过考虑到这个价格，只能呵呵了）。

要使用 Parallels 的话，需要安装 Parallels Vagrant 插件，这是免费的（仅仅是插件免费哈）。

由于 Vagrant 限制的因素，Hyper-V 提供者会忽略所有网络设置。

安装 Homestead Vagrant 盒子

VirtualBox/VMWare 和 Vagrant 安装好了之后，在终端中使用如下命令将

Homestead Vagrant 盒子 `laravel/homestead` 添加到 Vagrant 中。下载该盒子将会花费一些时间，具体时间长短主要取决于你的网络连接速度：

```
vagrant box add laravel/homestead
```

```
sunqiangdeMacBook-Pro:~ sunqiang$ vagrant box add laravel/homestead
=> box: Loading metadata for box 'laravel/homestead'
  box: URL: https://atlas.hashicorp.com/laravel/homestead
This box can work with multiple providers! The providers that it
can work with are listed below. Please review the list and choose
the provider you will be working with.

1) parallels
2) virtualbox
3) vmware_desktop

Enter your choice: 2
=> box: Adding box 'laravel/homestead' (v3.0.0) for provider: virtualbox
  box: Downloading: https://vagrantcloud.com/laravel/boxes/homestead/versions/3.0.0/providers/virtualbox.box
=> box: Box download is resuming from prior download progress
  box: Progress: 0% (Rate: 227k/s, Estimated time remaining: 6:40:20)
```

如果上述命令执行失败，需要确认 Vagrant 是否是最新版本。

运行命令会列出一个选择列表，选择 `virtualbox` 对应选项即可，然后进入漫长的下载等待，看看多喜人，还要 6 小时 40 分钟才能下载完成，基本上一觉醒来的节奏，还是在使用了 VPN 翻墙的情况下，如果一直提示网络超时的话只能去 Vagrant 官网下载了：

<https://atlas.hashicorp.com/laravel/boxes/homestead/versions/0.5.0/providers/virtualbox.box>, 通过这种方式下载的话需要手动将其添加到 Vagrant:

```
vagrant box add laravel/homestead ~/Downloads/virtualbox.box
```

运行上述命令有可能报错:

```
Check your Homestead.yaml file, the path to your private key does not exist.
```

解决办法如下:

```
ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
eval "$(ssh-agent -s)"
ssh-add -K ~/.ssh/id_rsa
```

再次运行上述命令即可添加成功:

```
localhost:Homestead sunqiang$ vagrant box add laravel/homestead ~/Downloads/virtualbox.box
=> box: Box file was not detected as metadata. Adding it directly...
=> box: Adding box 'laravel/homestead' (v0) for provider:
  box: Unpacking necessary files from: file:///Users/sunqiang/Downloads/virtualbox.box
=> box: Successfully added box 'laravel/homestead' (v0) for 'virtualbox'!
```

安装 Homestead

你可以通过克隆仓库代码来实现 Homestead 安装。将仓库克隆到用户目录下的 Homestead 目录，这样 Homestead 盒子就可以作为所有其他 Laravel 项目的主机:

```
cd ~
git clone https://github.com/laravel/homestead.git Homestead
```

克隆完成后，你需要检查 Homestead 的版本标签，因为 master 分支不会总是稳定版本，你可以在 [GitHub Release Page](#) 查找到最新稳定版本然后在本地将其检出:

```
cd Homestead

// Clone the desired release...
git checkout v7.1.2
```

接下来，在 Homestead 目录下运行 `bash init.sh` 命令来创建 Homestead.yaml 配置文件，生成的 Homestead.yaml 配置文件文件位于当前 Homestead 目录:

```
// Mac/Linux...
bash init.sh

// Windows...
init.bat
```

配置 Homestead

设置 Provider

Homestead.yaml 文件中的 provider 键表示使用哪个 Vagrant 提供者: virtualbox、vmware_fusion、vmware_workstation、parallels 或 hyperv，你可以将其设置为自己选择的提供者，当然对大部分人来说也没得选:

```
provider: virtualbox
```

配置共享文件夹

Homestead.yaml 文件中的 folders 属性列出了所有主机和 Homestead 虚拟机共享的文件夹，一旦这些目录中的文件有了修改，将会在本地和 Homestead 虚拟机之间保持同步，如果有需要的话，你可以配置多个共享文件夹:

```
folders:
  - map: ~/Development
    to: /home/vagrant/Code
```

如果你只是创建了很少的站点，使用一般的映射就够了。不过，随着站点数量的增加，你就会遇到性能问题，尤其是在包含大量文件的低端机器或项目中，性能问题可能会非常明显。如果你不幸遇到了这个问题，可以尝试映射每个项目到各自的 Vagrant 文件夹:

```
folders:
  - map: ~/code/project1
    to: /home/vagrant/code/project1

  - map: ~/code/project2
    to: /home/vagrant/code/project2
```

如果要开启 NFS，只需简单添加一个标识到同步文件夹配置:

```
folders:
  - map: ~/Development
    to: /home/vagrant/Code
    type: "nfs"
```

注: 使用 NFS 的话，需要考虑安装 `vagrant-bindfs` 插件。该插件可用于在 Homestead 盒子中为文件和目录维护正确的用户/组权限。

你还可以通过 `options` 传递其他 Vagrant 支持的同步文件夹选项：

```
folders:
  - map: ~/code
    to: /home/vagrant/code
    type: "rsync"
    options:
      rsync_args: ["--verbose", "--archive", "--delete", "-zz"]
      rsync_exclude: ["node_modules"]
```

配置 Nginx 站点

对 Nginx 不熟？没关系！通过 `sites` 属性你可以方便地将“域名”映射到 Homestead 虚拟机的指定目录，`Homestead.yaml` 中默认已经配置了一个示例站点。和共享文件夹一样，你可以配置多个站点：

```
sites:
  - map: homestead.app
    to: /home/vagrant/Code/Laravel/public
```

如果你是在 Homestead 盒子启动之后进行了上述修改，需要运行 `vagrant reload --provision` 更新虚拟机上的 Nginx 配置。

Hosts 文件

不要忘记把 Nginx 站点配置中的域名添加到本地机器上的 `hosts` 文件中，该文件会将对本地域名的请求重定向到 Homestead 虚拟机，在 Mac 或 Linux 上，该文件位于 `/etc/hosts`，在 Windows 上，位于 `C:\Windows\System32\drivers\etc\hosts`，添加方式如下：

```
192.168.10.10 homestead.test
```

确保 IP 地址和你的 `Homestead.yaml` 文件中列出的一致，一旦你将域名添加到 `hosts` 文件，你就可以在浏览器中通过该域名访问站点了：

```
http://homestead.test
```

注：在真正可以访问之前还需要通过 Vagrant 启动虚拟机上的 Homestead 盒子。

启动 Vagrant 盒子

配置好 `Homestead.yaml` 文件后，在 `Homestead` 目录下运行 `vagrant up` 命令，Vagrant 将会启动虚拟机并自动配置共享文件夹以及 Nginx 站点，初次启动需要花费一点时间进行初始化：

```
localhost:Homestead sunqiang$ vagrant up
Bringing machine 'homestead-7' up with 'virtualbox' provider...
==> homestead-7: Box 'laravel/homestead' could not be found. Attempting to find and install...
    homestead-7: Box Provider: virtualbox
    homestead-7: Box Version: >= 3.0.0
==> homestead-7: Loading metadata for box 'laravel/homestead'
    homestead-7: URL: https://atlas.hashicorp.com/laravel/homestead
==> homestead-7: Adding box 'laravel/homestead' (v3.0.0) for provider: virtualbox
    homestead-7: Downloading: https://vagrantcloud.com/laravel/boxes/homestead/versions/3.0.0/providers/virtualbox.box
==> homestead-7: Box download is resuming from prior download progress
==> homestead-7: Successfully added box 'laravel/homestead' (v3.0.0) for 'virtualbox'!
==> homestead-7: Importing base box 'laravel/homestead'...
==> homestead-7: Matching MAC address for NAT networking...
==> homestead-7: Checking if box 'laravel/homestead' is up to date...
==> homestead-7: Setting the name of the VM: homestead-7
==> homestead-7: Clearing any previously set network interfaces...
==> homestead-7: Preparing network interfaces based on configuration...
```

启动之后，你可以在浏览器中通过 `http://homestead.test` 访问 Laravel 应用了（前提是 Web 目录下已经部署 Laravel 应用代码）：


[DOCUMENTATION](#)
[LARACASTS](#)
[NEWS](#)
[FORGE](#)
[GITHUB](#)

要登录到该虚拟机，使用 `vagrant ssh` 命令；关闭该虚拟机，可以使用 `vagrant halt` 命令；销毁该虚拟机，可以使用 `vagrant destroy --force` 命令。

为指定项目安装 Homestead

全局安装 Homestead 将会使每个项目共享同一个 Homestead 盒子，你还可以为每个项目单独安装 Homestead，这样就会在该项目下创建 `Vagrantfile`，允许其他人在该项目中执行 `vagrant up` 命令，在指定项目根目录下使用 Composer 执行安装命令如下：

```
composer require laravel/homestead --dev
```

这样就在项目中安装了 Homestead。Homestead 安装完成后，使用 `make` 命令生成 `Vagrantfile` 和 `Homestead.yaml` 文件，`make` 命令将会自动配置 `Homestead.yaml` 中的 `sites` 和 `folders` 属性。该命令执行方式如下：

Mac/Linux:

```
php vendor/bin/homestead make
```

Windows:

```
vendor\bin\homestead make
```

接下来，在终端中运行 `vagrant up` 命令然后在浏览器中通过 `http://homestead.test` 访问站点。不要忘记在 `/etc/hosts` 文件中添加域名 `homestead.test`（已配置的话忽略）。

安装 MariaDB

如果你希望使用 MariaDB 来替代 MySQL，可以添加 `mariadb` 配置项到 `Homestead.yaml` 文件，该选项会移除 MySQL 并安装 MariaDB，MariaDB 是 MySQL 的替代品，完全兼容 MySQL，所以在应用数据库配置中你仍然可以使用 `mysql` 驱动：

```
box: laravel/homestead
ip: "192.168.10.10"
memory: 2048
cpus: 4
provider: virtualbox
mariadb: true
```

安装 Elasticsearch

要安装 Elasticsearch，需要添加 `elasticsearch` 到 `Homestead.yaml` 文件并指定一个支持的版本。默认安装会创建一个名为「homestead」的集群，不要给 Elasticsearch 分配超过操作系统一半的内存，因此确保 Homestead 机器内存至少是分配给 Elasticsearch 的两倍：

```
box: laravel/homestead
ip: "192.168.10.10"
memory: 4096
cpus: 4
provider: virtualbox
elasticsearch: 6
```

注：查看 [Elasticsearch 文档](#) 学习如何自定义配置。

别名

你可以在 Homestead 目录下通过编辑 `aliases` 文件为 Homestead 机器添加 Bash 别名：

```
alias c='clear'
alias ..='cd ..'
```

更新完 `aliases` 文件后，需要通过 `vagrant reload --provision` 命令重启 Homestead 机器，以确保新的别名在机器上生效。

日常使用

全局访问 Homestead

要想在文件系统的任意路径都能够运行 `vagrant up` 启动 Homestead 虚拟机，在 Mac/Linux 系统中，可以添加 `Bash` 函数到 `~/.bash_profile`；在 Windows 系统上，需要添加“批处理”文件到 `PATH`。这些脚本允许你在系统的任意位置运行 Vagrant 命令，并且把命令执行位置指向 Homestead 的安装路径。

Mac/Linux

```
function homestead() {
    ( cd ~/Homestead && vagrant $* )
}
```

确保将该函数中的 `~/Homestead` 路径调整为指向实际的 Homestead 安装路径。这样你就可以在系统的任意位置运行 `homestead up` 或 `homestead ssh` 来启动/登录虚拟机：

```
localhost:~ sunqiang$ homestead ssh
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-81-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

3 packages can be updated.
0 updates are security updates.
```

补充知识点：`/etc/profile` 和 `~/.bash_profile` 都可以用来设置系统 `PATH`，不同之处在于前者是给系统超级用户使用，后者是给普通登录用户使用的，此外要让 `~/.bash_profile` 修改后生效，有两种方法，一种是退出系统重新登录，一种是使用 `source ~/.bash_profile` 命令。

Windows

在系统的任意位置创建一个批处理文件 `homestead.bat`:

```
@echo off

set cwd=%cd%
set homesteadVagrant=C:\Homestead

cd /d %homesteadVagrant% && vagrant %*
cd /d %cwd%

set cwd=
set homesteadVagrant=
```

你需要将脚本中实例路径 `C:\Homestead` 调整为 Homestead 实际安装路径。创建文件之后，添加文件路径到 `PATH`，这样你就可以在系统的任意位置运行 `homestead up` 或 `homestead ssh` 命令了。

通过 SSH 连接

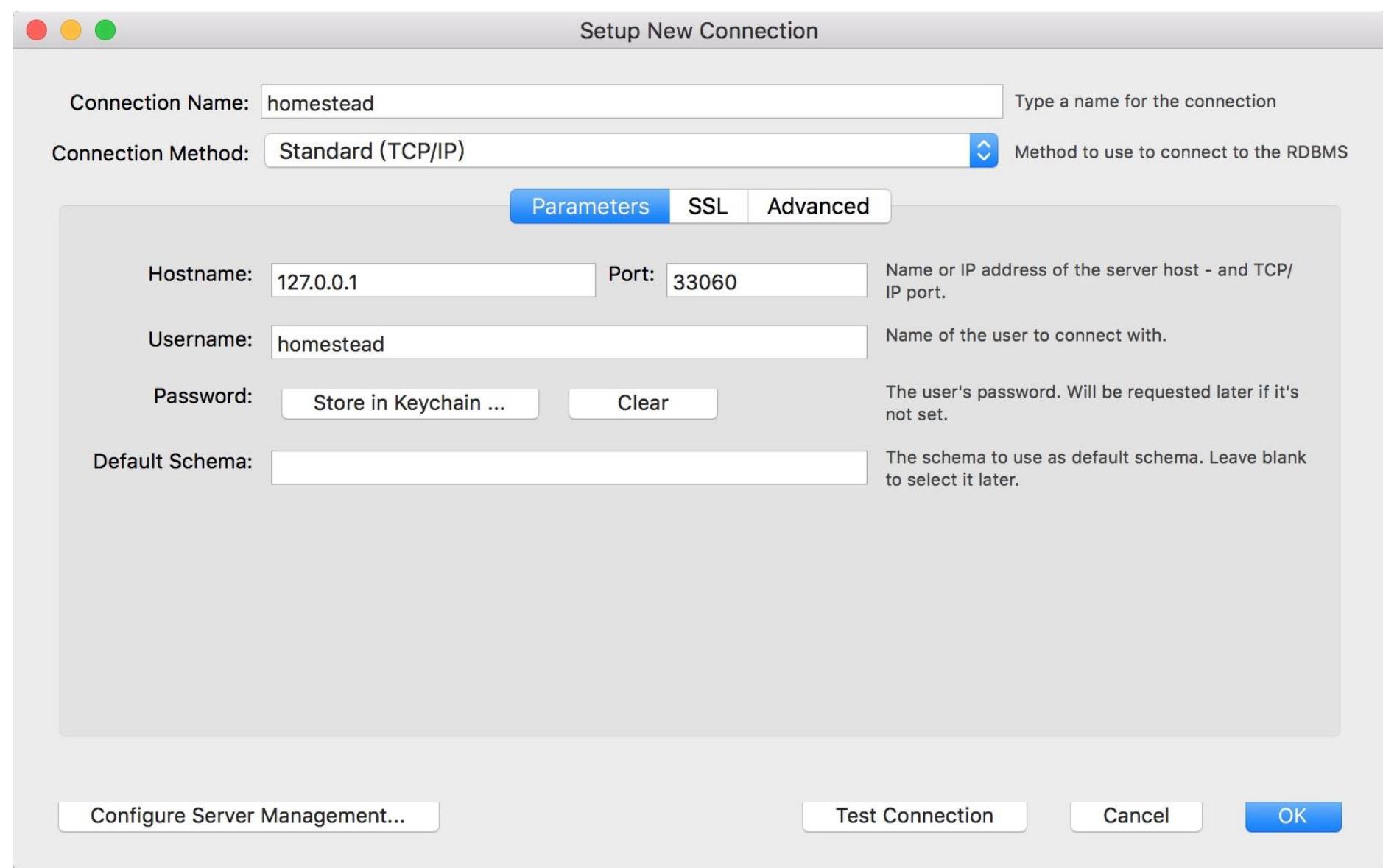
你可以在 `Homestead` 目录下通过运行 `vagrant ssh` 以 SSH 方式连接到虚拟机。如果你设置了全部访问 Homestead，也可以在任意路径下通过 `homestead ssh` 登录到虚拟机。

如果你需要以更简捷的方式连接到 Homestead，可以为主机添加一个别名来快速连接到 Homestead 盒子，创建完别名后，可以使用 `vm` 命令从任何地方以 SSH 方式连接到 Homestead 虚拟机:

```
alias vm="ssh vagrant@127.0.0.1 -p 2222"
```

连接到数据库

Homestead 默认已经在虚拟机中为 MySQL 和 Postgres 数据库做好了配置，更方便的是，这些配置值与 Laravel 的 `.env` 中默认提供的配置一致。想要通过本地的 Navicat 或 Sequel Pro 连接到 Homestead 上的 MySQL 或 Postgres 数据库，可以通过新建连接来实现，主机 IP 都是 `127.0.0.1`，对于 MySQL 而言，端口号是 `33060`，对 Postgres 而言，端口号是 `54320`，用户名/密码是 `homestead/secret`:



[88]

注：只有从本地连接 Homestead 的数据库时才能使用这些非标准的端口，在 Homestead 虚拟机中还是应该使用默认的 3306 和 5432 端口进行数据库连接配置。

添加更多站点

Homestead 虚拟机在运行时，可能需要添加多个 Laravel 应用到 Nginx 站点。如果是在单个 Homestead 环境中运行多个 Laravel 应用，添加站点很简单，只需将站点添加到 `Homestead.yaml` 文件，然后在 Homestead 目录中运行 `vagrant provision` 命令即可：

```
sites:
- map: homestead.test
  to: /home/vagrant/Code/Laravel/public
- map: another.test
  to: /home/vagrant/Code/another/public
```

如果 Vagrant 不是自动管理“hosts”文件，仍然需要添加站点域名到本地 `hosts` 文件：

```
192.168.10.10 homestead.test
192.168.10.10 another.test
```

添加完站点后，在 Homestead 目录下运行 `vagrant reload --provision` 命令重启虚拟机。

站点类型

Homestead 支持多种框架，所以即使你没有使用 Laravel 的话，也可以使用 Homestead，例如，我们可以通过 `symfony2` 站点类型轻松添加一个 Symfony 应用：

```
sites:
- map: symfony2.test
  to: /home/vagrant/Code/Symfony/web
  type: symfony2
```

目前支持的站点类型包括 `apache`、`laravel`（默认）、`proxy`、`silverstripe`、`statamic`、`symfony2` 和 `symfony4`。

站点参数

你也可以通过站点指令 `params` 添加额外的 Nginx `fastcgi_param` 值。例如我们可以添加一个 `FOO` 参数，对应参数值是 `BAR`：

```
sites:
- map: homestead.test
  to: /home/vagrant/Code/Laravel/public
  params:
    - key: FOO
      value: BAR
```

环境变量

你可以通过将变量添加到 `Homestead.yaml` 文件来设置全局环境变量：

```
variables:
- key: APP_ENV
  value: local
```

```
- key: FOO
  value: bar
```

更新完 `Homestead.yaml` 文件后，需要通过 `vagrant reload --provision` 命令重启机器，这将会更新所有已安装版本 PHP 的 PHP-FPM 配置并且为 `vagrant` 用户更新环境。

配置 Cron 调度任务

Laravel 提供了很方便的方式来调度 Cron 任务：只需每分钟调度运行一次 Artisan 命令 `schedule:run` 即可。`schedule:run` 会检查定义在 `App\Console\Kernel` 类中定义的调度任务并判断运行哪些任务。

如果想要为某个 Homestead 站点运行 `schedule:run` 命令，需要在定义站点时设置 `schedule` 为 `true`：

```
sites:
- map: homestead.app
  to: /home/vagrant/Code/Laravel/public
  schedule: true
```

该站点的 Cron 任务会被定义在虚拟机的 `/etc/cron.d` 目录下：

```
vagrant@homestead:/etc/cron.d$ ls -l
total 12
-rw-r--r-- 1 root root 589 Jul 16 2014 mdadm
-rw-r--r-- 1 root root 712 Jul 6 11:46 php
-rw-r--r-- 1 root root 396 Jan 27 2016 sysstat
```

配置 Maillog

通过 Maillog 可以轻松拦截发送出去的邮件并进行检查而不必真的将其发送给接收人。开始之前，需要更新 `.env` 文件使用如下邮件配置：

```
MAIL_DRIVER=smtp
MAIL_HOST=localhost
MAIL_PORT=1025
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
```

端口转发配置

默认情况下，Homestead 端口转发配置如下：

- SSH: 2222 → Forwards To 22
- ngrok UI: 4040 → Forwards To 4040
- HTTP: 8000 → Forwards To 80
- HTTPS: 44300 → Forwards To 443
- MySQL: 33060 → Forwards To 3306
- Postgres: 54320 → Forwards To 5432
- Mailhog: 8025 → Forwards To 8025

转发更多端口

如果你想要为 Vagrant 盒子添加更多端口转发，做如下转发协议设置即可：

```
ports:
- send: 50000
  to: 5000
- send: 7777
  to: 777
  protocol: udp
```

分享你的环境

有时候你可能希望和同事或客户分享自己当前的工作进度或成果，Vagrant 本身支持通过 `vagrant share` 来支持这个功能；不过，如果你在 `Homestead.yaml` 文件中配置了多个站点的话就不行了。

为了解决这个问题，Homestead 内置了自己的 `share` 命令，该功能实现的原理是通过 Ngrok 将本地服务分享到互联网上进行公开访问，关于该软件的细节我们这里不讨论，大家可以自行百度，我们主要关注在 Homestead 中如何使用这一功能。首先通过 `vagrant ssh` 登录到 Homestead 虚拟机然后运行 `share homestead.test` 命令，这样就可以分享 `homestead.test` 站点了，其他站点分享以此类推：

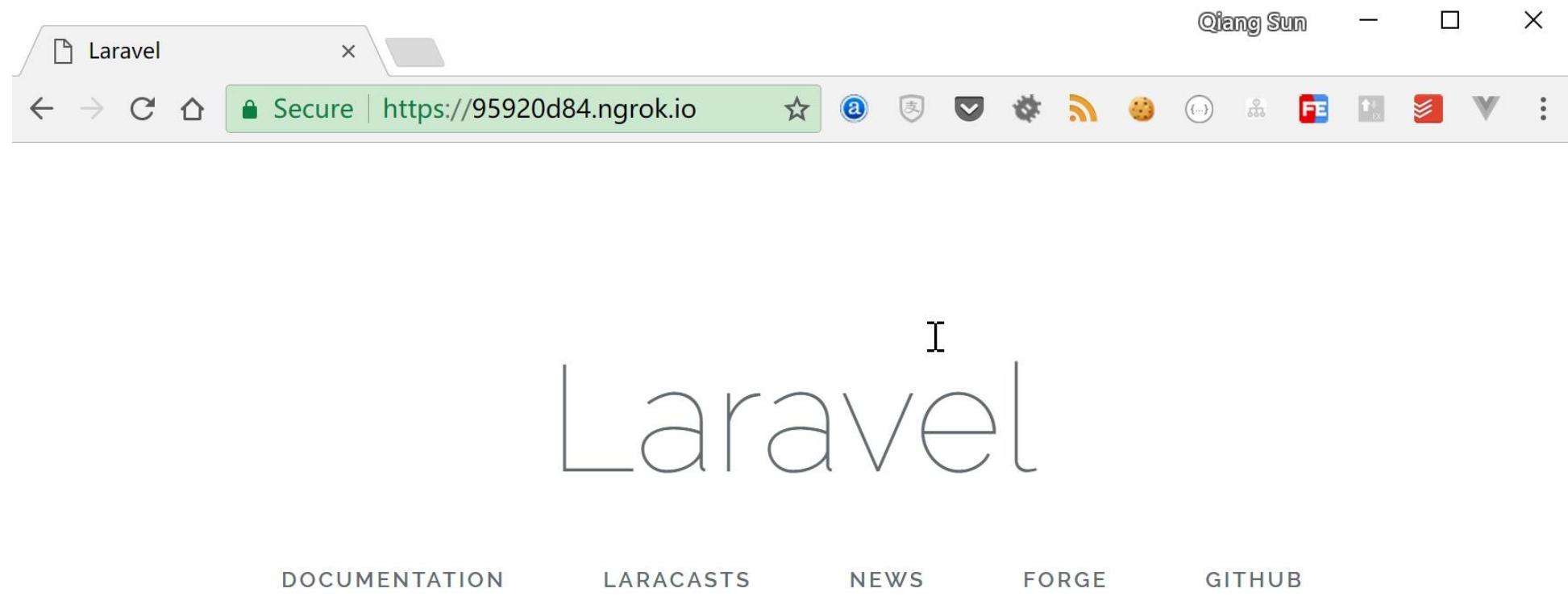
```
share homestead.test
```

运行完该命令之后，你就可以看到一个 Ngrok 界面出现，其中包含活动日志和分享站点所需的公开访问 URL：

ngrok by @inconschreveable

Session Status	online												
Version	2.2.8												
Region	United States (us)												
Web Interface	http://127.0.0.1:4040												
Forwarding	http://95920d84.ngrok.io -> localhost:80												
Forwarding	https://95920d84.ngrok.io -> localhost:80												
Connections	<table> <thead> <tr> <th>ttl</th> <th>opn</th> <th>rt1</th> <th>rt5</th> <th>p50</th> <th>p90</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0.00</td> <td>0.00</td> <td>0.00</td> <td>0.00</td> </tr> </tbody> </table>	ttl	opn	rt1	rt5	p50	p90	0	0	0.00	0.00	0.00	0.00
ttl	opn	rt1	rt5	p50	p90								
0	0	0.00	0.00	0.00	0.00								

可以看到我的分享 URL 是 <http://95920d84.ngrok.io/> 以及 <https://95920d84.ngrok.io/>，你可以通过这两个域名在任意联网机器上访问我的 Homestead 站点：



如果你想要指定一个自定义的区域，子域名或者其他 Ngrok 运行时选项，可以将它们添加到 `share` 命令：

```
share homestead.test -region=eu -subdomain=laravel
```

目前自定义域名只有付费用户才可以使用，所以 `subdomain` 会提示不可用。

注：记住两个点，一个是 Vagrant 并没有什么特别的安全防范措施，另一个是当你运行 `share` 命令的时候，你其实是在将自己的虚拟机暴露到互联网上。所以，当你要分享自己的站点之前，先想想安全隐患，并将其规避掉。

多个 PHP 版本

注：该功能只在 Nginx 下有效。

Homestead 6 引入了在单个虚拟机中支持多个 PHP 版本的功能，你可以在 `Homestead.yaml` 文件中为特定站点指定 PHP 版本，目前支持的 PHP 版本包括 5.6、7.0 和 7.1 和 7.2：

```
sites:
  - map: homestead.test
    to: /home/vagrant/Code/Laravel/public
    php: "5.6"
```

该功能实现的原理是通过 `Homestead.yaml` 中配置的 PHP 版本在 Homestead 中启动相应的 `php-fpm` 服务，然后更新 Nginx 中相应的站点配置：

```

location ~ \.php$ {
    fastcgi_split_path_info ^(.+\.php)(/.+)$;
    fastcgi_pass unix:/var/run/php/php7.0-fpm.sock;
    fastcgi_index index.php;
    include fastcgi_params;
    fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;

    fastcgi_intercept_errors off;
    fastcgi_buffer_size 16k;
    fastcgi_buffers 4 16k;
    fastcgi_connect_timeout 300;
    fastcgi_send_timeout 300;
    fastcgi_read_timeout 300;
}

```

此外，你可以在 CLI 中使用任意版本的 PHP:

```

php5.6 artisan list
php7.0 artisan list
php7.1 artisan list
php7.2 artisan list

```

Web 服务器

Homestead 使用 Nginx 作为默认 Web 服务器。不过，如果指定 apache 作为站点类型，也会安装 Apache。两个 Web 服务器可以同时安装，但是不能同时运行。Shell 命令 `flip` 可用于简化在不同 Web 服务器之间的切换处理，其工作原理是先自动判断哪个 Web 服务器正在运行，然后将其关闭，接着启动另一个服务器。要使用这个命令，需要 SSH 登录到 Homestead 机器并在终端中运行：

```
flip
```

网络接口

`Homestead.yaml` 的 `networks` 属性用于配置 Homestead 的网络接口，你可以想配多少就配多少：

```

networks:
  - type: "private_network"
    ip: "192.168.10.20"

```

要开启 `bridged`（桥接模式）接口，需要配置 `bridge` 设置并修改网络类型为 `public_network`：

```

networks:
  - type: "public_network"
    ip: "192.168.10.20"
    bridge: "en1: Wi-Fi (AirPort)"

```

要开启 `DHCP`（动态主机配置协议），只需要从配置中移除 `ip` 选项即可：

```

networks:
  - type: "public_network"
    bridge: "en1: Wi-Fi (AirPort)"

```

更新 Homestead

更新 Homestead 只需两步即可，首先，使用 `vagrant box update` 命令更新 Vagrant 盒子：

```
vagrant box update
```

接下来，需要更新 Homestead 源码，如果你是通过 Github 仓库安装的，只需在克隆仓库的地方运行 `git pull origin master` 即可。如果你是通过项目的 `composer.json` 文件安装的 Homestead，需要确保 `composer.json` 文件包含 `"laravel/homestead": "^7"` 并更新你的依赖：

```
composer update
```

虚拟机指定设置

VirtualBox

`natdnshostresolver`

默认情况下，Homestead 配置项 `natdnshostresolver` 被设置为 `on`，从而允许 Homestead 使用主机操作系统的 DNS 配置，如果你想要覆盖这个行为，添加如下行到 `Homestead.yaml` 文件：

```
provider: virtualbox
natdnshostresolver: off
```

Windows 上的符号链接

如果符号链接在 Windows 机器上不能正常工作，可能需要添加如下区块到 `Vagrantfile`:

```
config.vm.provider "virtualbox" do |v|
  v.customize ["setextradata", :id, "VBoxInternal2/SharedFoldersEnableSymlinksCreate/v-root",
  "1"]
end
```

Valet

简介

Valet 是为 Mac 打造的极简 Laravel 开发环境，没有 Vagrant，没有虚拟机，也无需配置 `/etc/hosts` 文件，还可以使用本地隧道公开分享你的站点。

启动 Mac 后，Laravel Valet 会在后台静默运行 Nginx，然后通过 DnsMasq，Valet 会代理所有针对 `*.test` 域名的请求指向本地安装的站点目录。此外，这样一个极速的 Laravel 开发环境只需要占用 7M 内存。Valet 并不是想要替代 Vagrant 或者 Homestead，只是提供了另外一种选择，更加灵活、极速、以及占用更小的内存空间。正是基于这些原因，我们将 Valet 称之为轻量级的开发环境。

Valet 开箱支持但不限于以下软件和工具：

- [Laravel](#)
- [Lumen](#)
- [Bedrock](#)
- [CakePHP 3](#)
- [Concrete5](#)
- [Contao](#)
- [Craft](#)
- [Drupal](#)
- [Jigsaw](#)
- [Joomla](#)
- [Katana](#)
- [Kirby](#)
- [Magento](#)
- [OctoberCMS](#)
- [Sculpin](#)
- [Slim](#)
- [Statamic](#)
- [Static HTML](#)
- [Symfony](#)
- [WordPress](#)
- [Zend](#)

以上支持的驱动文件位于 `~/.composer/vendor/laravel/valet/cli/drivers` 目录下，当然，你还可以通过自定义驱动扩展 Valet，自定义的驱动文件存放在 `~/.valet/Drivers` 目录。

Valet 还是 Homestead

正如我们[上篇文档](#)所介绍的，Laravel 还提供了另外一个开发环境 Homestead。Homestead 和 Valet 的不同之处在于两者的目标受众和本地开发方式。

Homestead 提供了一个完整的、包含自动化 Nginx 配置的 Ubuntu 虚拟机，如果你需要一个完整的虚拟化 Linux 开发环境或者使用的是 Windows/Linux 操作系统，那么 Homestead 无疑是最佳选择，此外，学院君以为如果是公司团队进行正规的工程化开发，还是使用 Homestead 为佳，原因我在上篇文档中已经提及。

Valet 官方默认只支持 Mac，并且要求本地自行安装 PHP 和数据库服务器，当然这可以通过 Homebrew 命令轻松实现（`brew install php72` 以及 `brew install mysql`），Valet 通过最小的资源消耗提供了一个极速的本地开发环境，如果你只需要 PHP/MySQL 并且不需要完整的虚拟化开发环境，那么 Valet 将是最好的选择，学院君建议如果是 Mac 环境本地尝鲜，所以尝鲜就是以学习为目的或者只是快速做个 Demo 原型，那 Valet 无疑是很棒的选择。

最后，建议归建议，Valet 和 Homestead 都是搭建本地 Laravel 开发环境的好工具，最终选择使用哪一个取决于你个人的喜好或团队的需求。

安装

注：已安装的直接跳到升级部分。

Valet 要求 Mac 操作系统并且已安装 Homebrew。安装之前，还要确保没有其他程序如 Apache 或 Nginx 绑定到本地的 80 端口。安装步骤如下：

- 使用 `brew update` 安装或更新 Homebrew 到最新版本；
- 通过 Homebrew 安装 PHP 7.2：`brew install homebrew/php/php72`；
- 通过 Composer 安装 Valet：`composer global require laravel/valet`
- 运行 `valet install` 命令，这将会配置并安装 Valet 和 DnsMasq，然后注册 Valet 后台随机启动。

安装完 Valet 后，尝试使用命令如 `ping foobar.test` 在终端 ping 一下任意 `*.test` 域名，如果 Valet 安装正确就会看到来自 `127.0.0.1` 的响应：

```
PING foobar.dev (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.069 ms
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.077 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.072 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.082 ms
```

每次系统启动的时候 Valet 会在后台自动启动，不需要再次手动运行 `valet start` 或 `valet install`。

使用其他域名

默认情况下，Valet 使用 `.test` 域名后缀，如果你想要使用其他域名，可以使用 `valet domain tld-name` 命令。

例如，你想使用 `.com` 域名后缀取代 `.test`，运行 `valet domain com`，Valet 将会自动将站点域名后缀改为 `*.com`。

数据库

注：已安装 MySQL 数据库忽略本条。

如果你需要数据库，可以在命令行通过 `brew install mysql` 安装 MySQL，安装完成后就可以通过 `brew services start mysql` 来启动它，然后通过用户名 `root` 和一个空密码连接到本地数据库。

升级

你可以在终端使用 `composer global update` 命令来升级 Valet。升级之后，最好运行下 `valet install` 命令以便 Valet 在必要情况下对配置文件进行升级：

```
Package operations: 1 install, 26 updates, 2 removals
- Removing symfony/debug (v3.3.8)
- Removing psr/log (1.0.2)
- Updating symfony/polyfill-mbstring (v1.5.0 => v1.7.0): Loading from cache
- Installing symfony/polyfill-php72 (v1.7.0): Loading from cache
- Updating symfony/var-dumper (v3.3.9 => v4.0.6): Loading from cache
- Updating tightenco/collect (v5.5.3 => v5.6.7): Loading from cache
- Updating symfony/process (v3.3.8 => v4.0.6): Loading from cache
- Updating symfony/console (v3.3.8 => v4.0.6): Loading from cache
- Updating mnapoli/silly (1.6.0 => 1.7.0): Loading from cache
- Updating psr/simple-cache (1.0.0 => 1.0.1): Loading from cache
- Updating illuminate/contracts (v5.5.2 => v5.6.7): Loading from cache
- Updating illuminate/container (v5.5.2 => v5.6.7): Loading from cache
- Updating laravel/valet (v2.0.5 => v2.0.8): Loading from cache
- Updating sebastian/comparator (2.0.2 => 2.1.3): Loading from cache
- Updating phpunit/phpunit-mock-objects (4.0.4 => 5.0.6): Loading from cache
- Updating phpunit/php-file-iterator (1.4.2 => 1.4.5): Loading from cache
- Updating phpunit/php-token-stream (2.0.1 => 2.0.2): Loading from cache
- Updating phpunit/php-code-coverage (5.2.2 => 5.3.0): Loading from cache
- Updating webmozart/assert (1.2.0 => 1.3.0): Loading from cache
- Updating phpdocumentor/reflection-common (1.0 => 1.0.1): Loading from cache
- Updating phpdocumentor/reflection-docblock (4.1.1 => 4.3.0): Loading from cache
- Updating phpspec/prophecy (v1.7.2 => 1.7.5): Loading from cache
- Updating myclabs深深 (1.6.1 => 1.7.0): Loading from cache
- Updating phpunit/phpunit (6.3.0 => 6.5.7): Loading from cache
- Updating symfony/filesystem (v3.3.8 => v4.0.6): Loading from cache
- Updating laravel/installer (v1.4.1 => v1.5.0): Loading from cache
- Updating nikic/php-parser (v3.1.1 => v3.1.5): Loading from cache
- Updating psy/psysh (v0.8.11 => v0.8.17): Loading from cache
- Updating squizlabs/php_codesniffer (3.1.1 => 3.2.3): Loading from cache

Writing lock file
Generating autoload files
```

升级到 Valet 2.0

注：先通过 `valet -version` 查看本机安装的 Valet 版本，如果版本已经大于 2.0 则可以跳过本段教程。

Valet 2.0 将 Valet 底层的 web 服务器从 Caddy 替换成了 Nginx，升级到这个版本之前需要运行以下命令来停止和卸载已经在后台运行的 Caddy：

```
valet stop
valet uninstall
```

接下来，需要升级到最新版本的 Valet。基于 Valet 的安装方式，你可以通过 Git 或 Composer 来实现，如果你是通过 Composer 安装的 Valet，需要通过如下方式更新到最新的主版本：

```
composer global require laravel/valet
```

最新的 Valet 源码下载好之后，运行 `install` 命令：

```
valet install
valet restart
```

升级之后，需要 re-park 和 re-link 站点。

访问站点

Valet 安装完成后，就可以启动服务站点，Valet 为此提供了两个命令：`park` 和 `link`。

park 命令

- 在 Mac 系统中创建一个新目录，例如 `mkdir ~/Sites`，然后进入这个目录并运行 `valet park`。这个命令会将当前所在目录作为 Web 根目录。
- 接下来，在新建的目录中创建一个新的 Laravel 站点：`laravel new blog`。
- 接下来，在浏览器中访问 `http://blog.com`（我通过 `valet domain com` 将域名后缀改成了 `.com`）。



Laravel

[DOCUMENTATION](#)

[LARACASTS](#)

[NEWS](#)

[FORGE](#)

[GITHUB](#)

这就是我们要做的全部工作。现在，所有在 `Sites` 目录中创建的 Laravel 项目都可以通过 `http://folder-name.com` 这种方式在浏览器中访问，是不是很方便？

link 命令

`link` 命令也可以用于访问本地 Laravel 站点，当你想要提供单个访问站点时这个命令很有用。

- 要使用这个命令，先切换到你的某个项目并运行 `valet link app-name`，这样 Valet 会在 `~/.valet/Sites` 中创建一个符号链接指向当前工作目录。
- 运行完 `link` 命令后，可以在浏览器中通过 `http://app-name.com` 访问站点。

要查看所有的链接目录，可以运行 `valet links` 命令。你也可以通过 `valet unlink app-name` 来删除符号链接。

注：你还可以使用 `valet link` 将多个（子）域名指向同一个应用，要添加子域名或其它域名到应用，可以在应用目录下运行 `valet link subdomain.app-name`，如这里我们在一个新应用下运行 `valet link forum.blog`：



Laravel

[DOCUMENTATION](#)

[LARACASTS](#)

[NEWS](#)

[FORGE](#)

[GITHUB](#)

一般来说我们使用 `park` 命令更方便一些，省去了后面新建应用重复执行命令，但是如果有了子域名这类特殊需求，只能使用 `link` 命令来实现了。

通过 TLS 让站点更安全

默认情况下，Valet 使用 HTTP 协议，如果你想要使用 HTTP/2 通过加密的 TLS 为站点提供服务，可以使用 `secure` 命令。例如，如果你的站点域名是 `blog.com`，可以使用如下命令：

```
valet secure blog
```



Laravel

[DOCUMENTATION](#)
[LARACASTS](#)
[NEWS](#)
[FORGE](#)
[GITHUB](#)

要想回到“非安全”的 HTTP，可以使用 `unsecure` 命令。和 `secure` 命令一样，该命令接收主机名作为参数：

```
valet unsecure blog
```

分享站点

Valet 还提供了一个命令用于将本地站点共享给其他人，这不需要任何额外工具即可实现，和 Homestead 一样，底层也是通过 Ngrok 实现。

要共享站点，切换到站点所在目录并运行 `valet share`，这会生成一个可以公开访问的 URL 并插入剪贴板，以便你直接复制到浏览器地址栏，就是这么简单：

ngrok by @inconshreveable

Session Status	online
Version	2.1.18
	United States (us)
Web Interface	4040
Forwarding	<code>http://4c59137d.ngrok.io -> blog.com:80</code>
Forwarding	<code>https://4c59137d.ngrok.io -> blog.com:80</code>
Connections	ttl opn rt1 rt5 p50 p90
	0 0 0.00 0.00 0.00 0.00

你可以通过 `http://4c59137d.ngrok.io` 或 `https://4c59137d.ngrok.io` 从任意联网机器访问站点（因为已经公开到互联网上）：



Laravel

[DOCUMENTATION](#)
[LARACASTS](#)
[NEWS](#)
[FORGE](#)
[GITHUB](#)

要停止共享站点，使用 `Control + C` 快捷键结束该命令即可。

注：`valet share` 目前尚不支持分享使用 `valet secure` 命令进行安全处理的站点，所以需要先通过 `valet unsecure` 命令解除安全访问。

自定义 Valet 驱动

你还可以编写自定义的 Valet 驱动为非 Valet 原生支持的 PHP 应用提供服务。安装完 Valet 时系统会创建一个 `~/.valet/Drivers` 目录，该目录中有一个 `SampleValetDriver.php` 文件，这个文件中有一个演示如何编写自定义驱动的示例。编写一个驱动只需要实现三个方法：`serves`、`isStaticFile` 和 `frontControllerPath`。

这三个方法接收 `$sitePath`、`$siteName` 和 `$uri` 值作为参数，其中 `$sitePath` 表示站点目录，如 `~/Sites/my-project`，`$siteName` 表示主域名部分，如 `my-project`，而 `$uri` 则是输入的请求地址，如 `/foo/bar`。

编写好自定义的 Valet 驱动后，将其放到 `~/.valet/Drivers` 目录并遵循 `FrameworkValetDriver.php` 这种命名方式，举个例子，如果你是在为 WordPress 编写自定义的 Valet 驱动，对应的文件名称为 `WordPressValetDriver.php`。

下面我们来具体讨论并演示自定义 Valet 驱动需要实现的三个方法。

`serves` 方法

如果自定义驱动需要继续处理输入请求，`serves` 方法会返回 `true`，否则该方法返回 `false`。因此，在这个方法中应该判断给定的 `$sitePath` 是否包含你服务类型的项目。

例如，假设我们编写的是 `WordPressValetDriver`，那么对应 `serves` 方法如下：

```
/**
 * Determine if the driver serves the request.
 *
 * @param string $sitePath
 * @param string $siteName
 * @param string $uri
 * @return void
 * @translator laravelacademy.org
 */
public function serves($sitePath, $siteName, $uri)
{
    return is_dir($sitePath . '/wp-admin');
}
```

`isStaticFile` 方法

`isStaticFile` 方法会判断输入请求是否是静态文件，例如图片或样式文件，如果文件是静态的，该方法会返回磁盘上的完整路径，如果输入请求不是请求静态文件，则返回 `false`：

```
/**
 * Determine if the incoming request is for a static file.
 *
 * @param string $sitePath
 * @param string $siteName
 * @param string $uri
 * @return string|false
 */
public function isStaticFile($sitePath, $siteName, $uri)
{
    if (file_exists($staticFilePath = $sitePath . '/public/' . $uri)) {
        return $staticFilePath;
    }

    return false;
}
```

注：`isStaticFile` 方法只有在 `serves` 方法返回 `true` 并且请求 URI 不是 `/` 的时候才会被调用。

`frontControllerPath` 方法

`frontControllerPath` 方法会返回前端控制器的完整路径，通常是 `index.php`：

```
/**
 * Get the fully resolved path to the application's front controller.
 *
 * @param string $sitePath
 * @param string $siteName
 * @param string $uri
 * @return string
 */
public function frontControllerPath($sitePath, $siteName, $uri)
{
    return $sitePath . '/public/index.php';
}
```

关于自定义 Valet 驱动可以参考学院君为 Flarum 论坛编写的扩展教程：在 Mac 开发环境 Laravel Valet 中配置运行 Flarum 论坛系统。

本地驱动

如果你想要为单应用程序定义一个自定义的 Valet 驱动，在应用根目录下创建一个 `LocalValetDriver.php` 文件，自定义驱动类可以继承自 `ValetDriver` 基类或者继承自己存在的应用指定驱动类如 `LaravelValetDriver`：

```
class LocalValetDriver extends LaravelValetDriver
{
```

```

/**
 * Determine if the driver serves the request.
 *
 * @param string $sitePath
 * @param string $siteName
 * @param string $uri
 * @return bool
 */
public function serves($sitePath, $siteName, $uri)
{
    return true;
}

/**
 * Get the fully resolved path to the application's front controller.
 *
 * @param string $sitePath
 * @param string $siteName
 * @param string $uri
 * @return string
 */
public function frontControllerPath($sitePath, $siteName, $uri)
{
    return $sitePath.'/public_html/index.php';
}
}

```

其他常用 Valet 命令

命令	描述
<code>valet forget</code>	从“parked”目录运行该命令以便从 parked 目录列表中移除该目录
<code>valet paths</code>	查看你的“parked”路径
<code>valet restart</code>	重启 Valet
<code>valet start</code>	启动 Valet
<code>valet stop</code>	关闭 Valet
<code>valet uninstall</code>	卸载 Valet

部署

简介

当你准备部署 Laravel 应用到生产环境时，有一些重要的事情可以确保应用尽可能高效地运行，在本文档中我们就来探讨下这些要做的事情从而确保应用以最佳方式部署。

服务器配置

Nginx

如果部署应用的服务器运行的是 Nginx，可以使用下面的配置文件配置 Web 服务器。大部分情况下，这个文件需要根据服务器的配置做一些调整（如果你想要一个工具来协助管理服务器，可以考虑使用 [Laravel Forge](#)）：

```

server {
    listen 80;
    server_name example.com;
    root /example.com/public;

    add_header X-Frame-Options "SAMEORIGIN";
    add_header X-XSS-Protection "1; mode=block";
    add_header X-Content-Type-Options "nosniff";

    index index.html index.htm index.php;
}

```

```

charset utf-8;

location / {
    try_files $uri $uri/ /index.php?$query_string;
}

location = /favicon.ico { access_log off; log_not_found off; }
location = /robots.txt { access_log off; log_not_found off; }

error_page 404 /index.php;

location ~ \.php$ {
    fastcgi_split_path_info ^(.+\.php)(/.+)$;
    fastcgi_pass unix:/var/run/php/php7.1-fpm.sock;
    fastcgi_index index.php;
    include fastcgi_params;
}

location ~ /\.well-known.* {
    deny all;
}
}

```

优化

自动加载优化

部署应用到生产环境时，确保优化过 Composer 的类自动加载映射以便 Composer 可以快速为给定类找到要加载的文件：

```
composer install --optimize-autoloader
```

注：除了优化自动加载器之外，还要在项目代码仓库中包含 `composer.lock` 文件，这样的话项目依赖可以更快安装。

优化配置加载

部署应用到生产环境时，需要确保在部署过程中运行了 Artisan 命令 `config:cache`：

```
php artisan config:cache
```

这个命令会将所有 Laravel 配置文件合并成一个缓存文件，从而极大减少框架在加载配置值时对文件系统的 IO 操作。

注：如果你在部署过程中执行了 `config:cache` 命令，需要确保只在配置文件中调用了 `env` 函数。一旦配置被缓存后，就不会再加载 `.env` 文件，因此所有对 `env` 函数的调用都会返回 `null`。

优化路由加载

如果你正在构建一个包含大量路由的大型应用，需要确保在部署过程中运行了 Artisan 命令 `route:cache`：

```
php artisan route:cache
```

这个命令会生成一个缓存文件将所有路由注册浓缩到单个方法调用，从而在注册大量路由时提升路由注册性能。

注：由于该特性使用了 PHP 序列化功能，所以只能缓存基于控制器的应用路由，因为 PHP 不能序列化闭包。

使用 Forge 部署

如果你对自己管理服务器配置、安装各种工具软件以及维护大型应用所需服务没有信心，或者觉得这些操作过于繁琐，那么 [Laravel Forge](#) 是一个不错的选择。

Laravel Forge 可以在不同的云服务供应商（例如 DigitalOcean、Linode、AWS 等）中创建服务器，此外，Forge 还会帮你安装并管理构建大型 Laravel 应用所需的所有工具，例如 Nginx、MySQL、Redis、Memcached、Beanstalk 等等。

三、底层原理

一次请求的生命周期

简介

当我们使用现实世界中的任何工具时，如果理解了该工具的工作原理，那么用起来就会得心应手，应用开发也是如此。当你理解了开发工具如何工作，用起来就会更加游刃有余。

这篇文档的目标就是从更高层面向你阐述 Laravel 框架的工作原理。通过对框架更全面的了解，一切都不再那么神秘，你将会更加自信地构建应用。如果你不能马上理解所有这些条款，不要失去信心！先试着掌握一些基本的东西，你的知识水平将会随着对文档的探索而不断提升。

生命周期概览

第一件事

Laravel 应用的所有请求入口都是 `public/index.php` 文件，所有请求都会被 web 服务器（Apache/Nginx）导向这个文件。`index.php` 文件包含的代码并不多，但是，这里是加载框架其它部分的起点。

`index.php` 文件载入 Composer 生成的自动加载设置，然后从 `bootstrap/app.php` 脚本获取 Laravel 应用实例，Laravel 的第一个动作就是创建服务容器实例。

HTTP/Console 内核

接下来，请求被发送到 HTTP 内核或 Console 内核（分别用于处理 Web 请求和 Artisan 命令），这取决于进入应用的请求类型。这两个内核是所有请求都要经过的中央处理器，现在，就让我们聚焦在位于 `app/Http/Kernel.php` 的 HTTP 内核。

HTTP 内核继承自 `Illuminate\Foundation\Http\Kernel` 类，该类定义了一个 `bootstrappers` 数组，这个数组中的类在请求被执行前运行，这些 `bootstrappers` 配置了错误处理、日志、检测应用环境以及其它在请求被处理前需要执行的任务。

HTTP 内核还定义了一系列所有请求在处理前需要经过的 HTTP 中间件，这些中间件处理 HTTP 会话的读写、判断应用是否处于维护模式、验证 CSRF 令牌等等。

HTTP 内核的 `handle` 方法签名相当简单：获取一个 `Request`，返回一个 `Response`，可以把该内核想象作一个代表整个应用的大黑盒子，输入 HTTP 请求，返回 HTTP 响应。

服务提供者

内核启动过程中最重要的动作之一就是为应用载入服务提供者，应用的所有服务提供者都被配置在 `config/app.php` 配置文件的 `providers` 数组中。首先，所有提供者的 `register` 方法被调用，然后，所有提供者被注册之后，`boot` 方法被调用。

服务提供者负责启动框架的所有各种各样的组件，比如数据库、队列、验证器，以及路由组件等，正是因为他们启动并配置了框架提供的所有特性，所以服务提供者是整个 Laravel 启动过程中最重要的部分。

分发请求

一旦应用被启动并且所有的服务提供者被注册，`Request` 将会被交给路由器进行分发，路由器将会分发请求到路由或控制器，同时运行所有路由指定的中间件。

聚焦服务提供者

服务提供者是启动 Laravel 应用中最关键的部分，应用实例被创建后，服务提供者被注册，请求被交给启动后的应用进行处理，整个过程就是这么简单！

对 Laravel 应用如何通过服务提供者构建和启动有一个牢固的掌握非常有价值，当然，应用默认的服务提供者存放在 `app/Providers` 目录下。

默认情况下，`AppServiceProvider` 是空的，这里是添加自定义启动和服务容器绑定的最佳位置，当然，对大型应用，你可能希望创建多个服务提供者，每一个都有着更加细粒度的启动。

注：更多细节，可参考 [Laravel 5.x 启动过程分析](#)。

服务容器

简介

Laravel 服务容器是一个用于管理类依赖和执行依赖注入的强大工具。依赖注入听上去很花哨，其实质是通过构造函数或者某些情况下通过 `setter` 方法将类依赖注入到类中。

让我们看一个简单的例子：

```
<?php

namespace App\Http\Controllers;

use App\User;
use App\Repositories\UserRepository;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * The user repository implementation.
     *
     * @var UserRepository
     */
    protected $users;

    /**
     * Create a new controller instance.
     *
     * @param UserRepository $users
     * @return void
    
```

```

/*
public function __construct(UserRepository $users)
{
    $this->users = $users;
}

/**
 * Show the profile for the given user.
 *
 * @param int $id
 * @return Response
 */
public function show($id)
{
    $user = $this->users->find($id);
    return view('user.profile', ['user' => $user]);
}

```

在本例中，`UserController` 需要从数据源获取用户，所以，我们注入了一个可以获取用户的服务 `UserRepository`，其扮演的角色类似使用 `Eloquent` 从数据库获取用户信息。注入 `UserRepository` 后，我们可以在其基础上封装其他实现，也可以模拟或者创建一个假的 `UserRepository` 实现用于测试。

深入理解 Laravel 服务容器对于构建功能强大的大型 Laravel 应用而言至关重要，对于贡献代码到 Laravel 核心也很有帮助。

绑定

绑定基础

几乎所有的服务容器绑定都是在 `服务提供者` 中完成。因此本文档的演示例子用到的容器都是在服务提供者中绑定。

注：如果一个类没有基于任何接口那么就没有必要将其绑定到容器。容器并不需要被告知如何构建对象，因为它会使用 PHP 的反射服务自动解析出具体的对象。

简单的绑定

在一个服务提供者中，可以通过 `$this->app` 变量访问容器，然后使用 `bind` 方法注册一个绑定，该方法需要两个参数，第一个参数是我们想要注册的类名或接口名称，第二个参数是返回类的实例的闭包：

```

$this->app->bind('HelpSpot\API', function ($app) {
    return new HelpSpot\API($app->make('HttpClient'));
});

```

注意到我们将容器本身作为解析器的一个参数，然后我们可以使用该容器来解析我们正在构建的对象的子依赖。

绑定一个单例

`singleton` 方法绑定一个只会解析一次的类或接口到容器，然后接下来对容器的调用将会返回同一个对象实例：

```

$this->app->singleton('HelpSpot\API', function ($app) {
    return new HelpSpot\API($app->make('HttpClient'));
});

```

绑定实例

你还可以使用 `instance` 方法绑定一个已存在的对象实例到容器，随后调用容器将总是返回给定的实例：

```

$api = new HelpSpot\API(new HttpClient);
$this->app->instance('HelpSpot\Api', $api);

```

绑定原始值

你可能有一个接收注入类的类，同时需要注入一个原生的数值比如整型，可以结合上下文轻松注入这个类需要的任何值：

```

$this->app->when('App\Http\Controllers\UserController')
    ->needs('$variableName')
    ->give($value);

```

绑定接口到实现

服务容器的一个非常强大的功能是其绑定接口到实现。我们假设有一个 `EventPusher` 接口及其实现类 `RedisEventPusher`，编写完该接口的 `RedisEventPusher` 实现后，就可以将其注册到服务容器：

```

$this->app->bind(
    'App\Contracts\EventPusher',
    'App\Services\RedisEventPusher'
);

```

这段代码告诉容器当一个类需要 `EventPusher` 的实现时将会注入 `RedisEventPusher`，现在我们可以在构造器或者任何其它通过服务容器注入依赖的地方进行 `EventPusher` 接口的依赖注入：

```

use App\Contracts\EventPusher;

/**

```

```
* 创建一个新的类实例

*
* @param EventPusher $pusher
* @return void
*/
public function __construct(EventPusher $pusher) {
    $this->pusher = $pusher;
}
```

上下文绑定

有时候我们可能有两个类使用同一个接口，但我们希望在每个类中注入不同实现，例如，两个控制器依赖 `Illuminate\Contracts\Filesystem\Filesystem` 契约的不同实现。Laravel 为此定义了简单、平滑的接口：

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

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

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

标签

少数情况下，我们需要解析特定分类下的所有绑定，例如，你正在构建一个接收多个不同 `Report` 接口实现的报告聚合器，在注册完 `Report` 实现之后，可以通过 `tag` 方法给它们分配一个标签：

```
$this->app->bind('SpeedReport', function () {
    //
});

$this->app->bind('MemoryReport', function () {
    //
});

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

这些服务被打上标签后，可以通过 `tagged` 方法来轻松解析它们：

```
$this->app->bind('ReportAggregator', function ($app) {
    return new ReportAggregator($app->tagged('reports'));
});
```

扩展绑定

`extend` 方法允许对解析服务进行修改。例如，当服务被解析后，可以运行额外代码装饰或配置该服务。`extend` 方法接收一个闭包来返回修改后的服务：

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

解析

make 方法

有很多方式可以从容器中解析对象，首先，你可以使用 `make` 方法，该方法接收你想要解析的类名或接口名作为参数：

```
$fooBar = $this->app->make('HelpSpot\API');
```

如果你所在的代码位置访问不了 `$app` 变量，可以使用辅助函数 `resolve`：

```
$api = resolve('HelpSpot\API');
```

某些类的依赖不能通过容器来解析，你可以通过关联数组方式将其传递到 `makeWith` 方法来注入：

```
$api = $this->app->makeWith('HelpSpot\API', ['id' => 1]);
```

自动注入

最后，也是最常用的，你可以简单的通过在类的构造函数中对依赖进行类型提示来从容器中解析对象，[控制器](#)、[事件监听器](#)、[队列任务](#)、[中间件](#)等都是通过这种方式。在具体实践中，这是大多数对象从容器中解析的方式。

容器会自动为其解析类注入依赖，例如，你可以在控制器的构造函数中为应用定义的仓库进行类型提示，该仓库会自动解析并注入该类：

```
<?php

namespace App\Http\Controllers;

use App\Users\Repository as UserRepository;

class UserController extends Controller{
    /**
     * 用户仓库实例
     */
    protected $users;

    /**
     * 创建一个控制器实例
     *
     * @param UserRepository $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }

    /**
     * 通过指定 ID 显示用户
     *
     * @param int $id
     * @return Response
     */
    public function show($id)
    {
        //
    }
}
```

容器事件

服务容器在每一次解析对象时都会触发一个事件，可以使用 [resolving](#) 方法监听该事件：

```
$this->app->resolving(function ($object, $app) {
    // Called when container resolves object of any type...
});

$this->app->resolving(HelpSpot\API::class, function ($api, $app) {
    // Called when container resolves objects of type "HelpSpot\API"...
});
```

正如你所看到的，被解析的对象将会传递给回调函数，从而允许你在对象被传递给消费者之前为其设置额外属性。

PSR-11

Laravel 的服务容器实现了 [PSR-11](#) 接口。所以，你可以通过类型提示 PSR-11 容器接口来获取 Laravel 容器的实例：

```
use Psr\Container\ContainerInterface;

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

    //
});
```

注：如果绑定到容器的唯一标识有冲突调用 [get](#) 方法会抛出异常。

注：强烈推荐阅读[深入理解控制反转（IoC）和依赖注入（DI）](#)深入理解服务容器和服务提供者的实现原理。

服务提供者

简介

服务提供者是 Laravel 应用启动的中心，你自己的应用以及所有 Laravel 的核心服务都是通过服务提供者启动。

但是，我们所谓的“启动”指的是什么？通常，这意味着注册服务，包括注册服务容器绑定、事件监听器、中间件甚至路由。服务提供者是应用配置的中心。

如果你打开 Laravel 自带的 `config/app.php` 文件，将会看到一个 `providers` 数组，这里就是应用所要加载的所有服务提供者类，当然，其中很多是延迟加载的，也就是说不是每次请求都会被加载，只有真的用到它们的时候才会加载。

通过本文档，你将会学习如何编写自己的服务提供者并在 Laravel 应用中注册它们。

编写服务提供者

所有的服务提供者都继承自 `Illuminate\Support\ServiceProvider` 类。大部分服务提供者都包含两个方法：`register` 和 `boot`。在 `register` 方法中，你唯一要做的事情就是绑定服务到[服务容器](#)，不要尝试在该方法中注册事件监听器，路由或者任何其它功能。

通过 Artisan 命令 `make:provider` 即可生成一个新的提供者：

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

register 方法

正如前面所提到的，在 `register` 方法中只绑定服务到[服务容器](#)，而不要做其他事情，否则，一不小心就可能用到一个尚未被加载的服务提供者提供的服务。

现在让我们来看看一个基本的服务提供者长什么样：

```
<?php

namespace App\Providers;

use Riak\Connection;
use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider{
    /**
     * 在容器中注册绑定.
     *
     * @return void
     */
    public function register()
    {
        $this->app->singleton(Connection::class, function ($app) {
            return new Connection(config('riak'));
        });
    }
}
```

该服务提供者只定义了一个 `register` 方法，并使用该方法在服务容器中定义了一个 `Riak\Connection` 的实现。如果你不知道服务容器是如何工作的，请参考[其文档](#)。

`bindings` 和 `singletons` 属性

如果你的服务提供者注册了很多简单的绑定，你可能希望使用 `bindings` 和 `singletons` 属性来替代手动注册每个容器绑定以简化代码。当服务提供者被框架加载后，会自动检查这些属性并注册相应绑定：

```
<?php

namespace App\Providers;

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

class AppServiceProvider extends ServiceProvider
{
    /**
     * All of the container bindings that should be registered.
     *
```

```

 * @var array
 */
public $bindings = [
    ServerProvider::class => DigitalOceanServiceProvider::class,
];

/**
 * All of the container singletons that should be registered.
 *
 * @var array
 */
public $singletons = [
    DowntimeNotifier::class => PingdomDowntimeNotifier::class,
];
}

```

boot 方法

如果我们想要在服务提供者中注册视图 composer 该怎么做？这就要用到 `boot` 方法了。该方法在所有服务提供者被注册以后才会被调用，这就是说我们可以在其中访问框架已注册的所有其它服务：

```

<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class ComposerServiceProvider extends ServiceProvider{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        view()->composer('view', function () {
            //
        });
    }
}

```

boot 方法的依赖注入

我们可以在 `boot` 方法中对依赖进行类型提示，`服务容器`会自动注入你所需要的依赖：

```

use Illuminate\Contracts\Routing\ResponseFactory;

public function boot(ResponseFactory $response){
    $response->macro('caps', function ($value) {
        //
    });
}

```

注册服务提供者

所有服务提供者都是通过配置文件 `config/app.php` 中进行注册，该文件包含了一个列出所有服务提供者名字的 `providers` 数组，默认情况下，其中列出了所有核心服务提供者，这些服务提供者启动 Laravel 核心组件，比如邮件、队列、缓存等等。

要注册你自己的服务提供者，只需要将其追加到该数组中即可：

```

'providers' => [
    // 其它服务提供者
    App\Providers\ComposerServiceProvider::class,
],

```

延迟加载服务提供者

如果你的提供者仅仅只是在 `服务容器` 中注册绑定，你可以选择延迟加载该绑定直到注册绑定的服务真的需要时再加载，延迟加载这样的一个提供者将会提升应用的性能，因为它不会在每次请求时都从文件系统加载。

Laravel 编译并保存所有延迟服务提供者提供的服务及服务提供者的类名。然后，只有当你尝试解析其中某个服务时 Laravel 才会加载其服务提供者。

想要延迟加载一个提供者，设置 `defer` 属性为 `true` 并定义一个 `provides` 方法，该方法返回该提供者注册的服务容器绑定：

```

<?php

namespace App\Providers;

use Riak\Connection;
use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider{
    /**
     * 服务提供者加是否延迟加载.
     *
     * @var bool
     */
    protected $defer = true;

    /**
     * 注册服务提供者
     *
     * @return void
     */
    public function register()
    {
        $this->app->singleton(Connection::class, function ($app) {
            return new Connection($app['config']['riak']);
        });
    }

    /**
     * 获取由提供者提供的服务.
     *
     * @return array
     */
    public function provides()
    {
        return [Connection::class];
    }
}

```

注：强烈推荐阅读[深入理解控制反转（IoC）和依赖注入（DI）](#)深入理解服务容器和服务提供者的实现原理。

门面 (Facades)

简介

注：对门面这个概念不理解？可参考[PHP 设计模式系列 —— 门面模式（Facade）](#)。

门面为应用[服务容器](#)中的绑定类提供了一个“静态”接口。Laravel 内置了很多门面，你可能在不知道的情况下正在使用它们。Laravel 的门面作为服务容器中底层类的“静态代理”，相比于传统静态方法，在维护时能够提供更加易于测试、更加灵活、简明优雅的语法。

Laravel 的所有门面都定义在 `Illuminate\Support\Facades` 命名空间下，所以我们可以轻松访问到门面：

```

use Illuminate\Support\Facades\Cache;

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

```

在整个 Laravel 文档中，很多例子使用了门面来演示框架的各种功能特性。

何时使用门面

门面有诸多优点，其提供了简单、易记的语法，让我们无需记住长长的类名即可使用 Laravel 提供的功能特性，此外，由于他们对 PHP 动态方法的独到用法，使得它们很容易测试。

但是，使用门面也有需要注意的地方，一个最主要的危险就是类范围蠕变。由于门面如此好用并且不需要注入，在单个类中使用过多门面，会让类很容易变得越来越大。使用依赖注入则会让此类问题缓解，因为一个巨大的构造函数会让我们很容易判断出类在变大。因此，使用门面的时候要尤其注意类的大小，以便控制其有限职责。

注：构建与 Laravel 交互的第三方扩展包时，最好注入 Laravel [契约](#)而不是使用门面，因为扩展包在 Laravel 之外构建，你将不能访问 Laravel 的门面测试辅助函数。

门面 vs. 依赖注入

依赖注入的最大优点是可以替换注入类的实现，这在测试时很有用，因为你可以注入一个模拟或存根并且在存根上断言不同的方法。

但是在静态类方法上进行模拟或存根却行不通，不过，由于门面使用了动态方法对服务容器中解析出来的对象方法调用进行了代理，我们也可以像测试注入类实例那样测试门面。例如，给定以下路由：

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

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

我们可以这样编写测试来验证 `Cache::get` 方法以我们期望的方式被调用：

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

/**
 * A basic functional test example.
 *
 * @return void
 */
public function testBasicExample()
{
    Cache::shouldReceive('get')
        ->with('key')
        ->andReturn('value');

    $this->visit('/cache')
        ->see('value');
}
```

门面 vs 辅助函数

除了门面之外，Laravel 还内置了许多辅助函数用于执行通用任务，比如生成视图、触发事件、分配任务，以及发送 HTTP 响应等。很多辅助函数提供了和相应门面一样的功能，例如，下面这个门面调用和辅助函数调用是等价的：

```
return View::make('profile');
return view('profile');
```

门面和辅助函数之间并不存在实质性差别，使用辅助函数的时候，可以像测试相应门面那样测试它们。例如，给定以下路由：

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

在调用底层，`cache` 方法会去调用 `Cache` 门面上的 `get` 方法，因此，尽管我们使用这个辅助函数，我们还是可以编写如下测试来验证这个方法以我们期望的方式和参数被调用：

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

/**
 * A basic functional test example.
 *
 * @return void
 */
public function testBasicExample()
{
    Cache::shouldReceive('get')
        ->with('key')
        ->andReturn('value');

    $this->visit('/cache')
        ->see('value');
}
```

门面工作原理

在 Laravel 应用中，门面就是一个为容器中对象提供访问方式的类。该机制原理由 `Facade` 类实现。Laravel 自带的门面，以及我们创建的自定义门面，都会继承自 `Illuminate\Support\Facades\Facade` 基类。

门面类只需要实现一个方法：`getFacadeAccessor`。正是 `getFacadeAccessor` 方法定义了从容器中解析什么，然后 `Facade` 基类使用魔术方法 `__callStatic()` 从你的门面中调用解析对象。

下面的例子中，我们将会调用 Laravel 的缓存系统，浏览代码后，也许你会觉得我们调用了 `cache` 的静态方法 `get`：

```
<?php
```

```

namespace App\Http\Controllers;

use Cache;
use App\Http\Controllers\Controller;

class UserController extends Controller{
    /**
     * 为指定用户显示属性
     *
     * @param int $id
     * @return Response
     */
    public function showProfile($id)
    {
        $user = Cache::get('user:'.$id);

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

```

注意我们在顶部位置引入了 `Cache` 门面。该门面作为代理访问底层 `Illuminate\Contracts\Cache\Factory` 接口的实现。我们对门面的所有调用都会被传递给 Laravel 缓存服务的底层实例。

如果我们查看 `Illuminate\Support\Facades\Cache` 类的源码，将会发现其中并没有静态方法 `get`：

```

class Cache extends Facade
{
    /**
     * 获取组件注册名称
     *
     * @return string
     */
    protected static function getFacadeAccessor() {
        return 'cache';
    }
}

```

`Cache` 门面继承 `Facade` 基类并定义了 `getFacadeAccessor` 方法，该方法的工作就是返回服务容器绑定类的别名，当用户引用 `Cache` 类的任何静态方法时，Laravel 从服务容器中解析 `cache` 绑定，然后在解析出的对象上调用所有请求方法（本例中是 `get`）。

实时门面

使用实时门面，可以将应用中的任意类当做门面来使用。为了说明如何使用这个功能，我们先看一个替代方案。例如我们假设 `Podcast` 模型有一个 `publish` 方法，尽管如此，为了发布博客，我们需要注入 `Publisher` 实例：

```

<?php

namespace App;

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

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

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

```

因为可以模拟注入的发布服务，所以注入发布实现到该方法后允许我们轻松在隔离状态下测试该方法。不过，这要求我们每次调用 `publish` 方法都要传递一个发布服务实例，使用实时门面，我们可以在维持这种易于测试的前提下不必显式传递 `Publisher` 实例。要生成一个实时门面，在导入类前面加上 `Facades` 命名空间前缀即可：

```
<?php
```

```

namespace App;

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

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

        Publisher::publish($this);
    }
}

```

使用实时门面后，发布服务实现将会通过使用 `Facades` 前缀后的接口或类名在服务容器中解析。在测试的时候，我们可以使用 Laravel 自带的门面测试辅助函数来模拟这个方法调用：

```

<?php

namespace Tests\Feature;

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

class PodcastTest extends TestCase
{
    use RefreshDatabase;

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

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

        $podcast->publish();
    }
}

```

门面类列表

下面列出了每个门面及其对应的底层类，这对深入给定根门面的 API 文档而言是个很有用的工具。[服务容器绑定键](#)也被包含进来：

门面	类	服务容器绑定
App	Illuminate\Foundation\Application	<code>app</code>
Artisan	Illuminate\Contracts\Console\Kernel	<code>artisan</code>
Auth	Illuminate\Auth\AuthManager	<code>auth</code>
Auth (实例)	Illuminate\Contracts\Auth\Guard	<code>auth.driver</code>
Blade	Illuminate\View\Compilers\BladeCompiler	<code>blade.compiler</code>
Broadcast	Illuminate\Contracts\Broadcasting\Factory	
Broadcast (实例)	Illuminate\Contracts\Broadcasting\Broadcaster	
Bus	Illuminate\Contracts\Bus\Dispatcher	

门面	类	服务容器绑定
Cache	Illuminate\Cache\CacheManager	cache
Cache (实例)	Illuminate\Cache\Repository	cache.store
Config	Illuminate\Config\Repository	config
Cookie	Illuminate\Cookie\CookieJar	cookie
Crypt	Illuminate\Encryption\Encrypter	encrypter
DB	Illuminate\Database\DatabaseManager	db
DB (实例)	Illuminate\Database\Connection	db.connection
Event	Illuminate\Events\Dispatcher	events
File	Illuminate\Filesystem\Filesystem	files
Gate	Illuminate\Contracts\Auth\Access\Gate	
Hash	Illuminate\Contracts\Hashing\Hasher	hash
Lang	Illuminate\Translation\Translator	translator
Log	Illuminate\Log\Writer	log
Mail	Illuminate\Mail\Mailer	mailer
Notification	Illuminate\Notifications\ChannelManager	
Password	Illuminate\Auth\Passwords\PasswordBrokerManager	auth.password
Password (实例)	Illuminate\Auth\Passwords\PasswordBroker	auth.password.broker
Queue	Illuminate\Queue\QueueManager	queue
Queue (实例)	Illuminate\Contracts\Queue\Queue	queue.connection
Queue(基类)	Illuminate\Queue\Queue	
Redirect	Illuminate\Routing\Redirector	redirect
Redis	Illuminate\Redis\RedisManager	redis
Redis (实例)	Illuminate\Redis\Connections\Connection	redis.connection
Request	Illuminate\Http\Request	request
Response	Illuminate\Contracts\Routing\ResponseFactory	
Response (实例)	Illuminate\Http\Response	
Route	Illuminate\Routing\Router	router
Schema	Illuminate\Database\Schema\Builder	
Session	Illuminate\Session\SessionManager	session
Session (实例)	Illuminate\Session\Store	session.store
Storage	Illuminate\Filesystem\FilesystemManager	filesystem
Storage (实例)	Illuminate\Contracts\Filesystem\Filesystem	filesystem.disk
URL	Illuminate\Routing\UrlGenerator	url
Validator	Illuminate\Validation\Factory	validator
Validator (实例)	Illuminate\Validation\Validator	

门面	类	服务容器绑定
View	Illuminate\View\Factory	view
View (实例)	Illuminate\View\View	

契约 (Contracts)

简介

Laravel 中的契约是指框架提供的一系列定义核心服务的接口。例如，`Illuminate\Contracts\Queue\Queue` 契约定义了队列任务需要实现的方法，`Illuminate\Contracts\Mail\Mailer` 契约定义了发送邮件所需要实现的方法。

每一个契约都有框架提供的相应实现。例如，Laravel 为队列提供了多个驱动的实现，邮件则由 `SwiftMailer` 驱动实现。

所有 Laravel 契约都有其对应的 GitHub 库，这为所有有效的契约提供了快速入门指南，同时也可作为独立、解耦的包被开发者使用。

契约 Vs. 门面

Laravel 门面为 Laravel 服务的使用提供了便捷方式 —— 不再需要从服务容器中类型提示和契约解析即可直接通过静态门面调用。

不同于门面不需要再构造器中进行类型提示，契约允许你在类中定义显式的依赖。有些开发者喜欢门面带来的便捷，也有些开发者倾向于使用契约，他们喜欢定义明确的依赖。

注：大多数应用中，不管你使用门面还是契约，合适就好。不过，如果你是在构建一个扩展包，那么就应该使用契约，因为更容易测试。

何时使用契约

正如上面所讨论的，大多数情况下使用契约还是门面取决于个人或团队的喜好，契约和门面都可以用于创建强大的、测试友好的 Laravel 应用。只要你保持类的职责单一，你会发现使用契约和门面并没有什么实质性的差别。

但是，对契约你可能还是有些疑问。例如，为什么要全部使用接口？使用接口是不是更复杂？下面让我们从两个方面来扒一扒为什么使用接口：松耦合和简单。

松耦合

首先，让我们看看一些缓存实现的紧耦合代码：

```
<?php

namespace App\Orders;

class Repository
{
    /**
     * 缓存
     */
    protected $cache;

    /**
     * 创建一个新的 Repository 实例
     *
     * @param \SomePackage\Cache\Memcached $cache
     * @return void
     */
    public function __construct(\SomePackage\Cache\Memcached $cache)
    {
        $this->cache = $cache;
    }

    /**
     * 通过 ID 获取订单
     *
     * @param int $id
     * @return Order
     */
    public function find($id)
    {
        if ($this->cache->has($id)) {
            // ...
        }
    }
}
```

```

    }
}

}

```

在这个类中，代码和给定缓存实现紧密耦合，由于我们基于一个来自包的具体的缓存类，如果包的 API 变了，那么相应的，我们的代码必须做修改。

类似的，如果我们想要替换底层的缓存技术（Memcached）为别的技术实现（Redis），我们将再一次不得不修改我们的代码库。我们的代码库应该并不知道谁提供的数据或者数据是怎么提供的。

我们可以基于一种简单的、与提供者无关的接口来优化我们的代码，从而替代上述那种实现：

```

<?php

namespace App\Orders;

use Illuminate\Contracts\Cache\Repository as Cache;

class Repository
{
    /**
     * 创建一个新的 Repository 实例
     *
     * @param Cache $cache
     * @return void
     */
    public function __construct(Cache $cache)
    {
        $this->cache = $cache;
    }
}

```

现在代码就不与任何特定提供者耦合，甚至与 Laravel 都是无关的。由于契约包不包含任何实现和依赖，你可以轻松的为给定契约编写可选实现代码，你可以随意替换缓存实现而不用去修改任何缓存消费代码。

简单

当所有 Laravel 服务都统一在简单接口中定义，很容易判断给定服务提供的功能。契约可以充当框架特性的简明文档。

此外，基于简单接口，代码也更容易理解和维护。在一个庞大而复杂的类中，与其追踪哪些方法是有效的，不如转向简单、干净的接口。

如何使用契约

那么，如何实现契约呢？这很简单。

Laravel 中很多类都是通过[服务容器](#)进行解析，包括控制器，以及监听器、中间件、队列任务，甚至路由闭包。所以，要实现一个契约，需要在解析类的构造函数中类型提示这个契约接口。

例如，看看下面这个事件监听器：

```

<?php

namespace App\Listeners;

use App\User;
use App\Events\OrderWasPlaced;
use Illuminate\Contracts\Redis\Database;

class CacheOrderInformation
{
    /**
     * The Redis database implementation.
     */
    protected $redis;

    /**
     * Create a new event handler instance.
     *
     * @param Database $redis
     * @return void
     */
    public function __construct(Database $redis)
    {
        $this->redis = $redis;
    }
}

```

```

 * Handle the event.
 *
 * @param OrderWasPlaced $event
 * @return void
 */
public function handle(OrderWasPlaced $event)
{
    //
}
}

```

事件监听器被解析的时候，服务容器会读取构造函数中的类型提示，并注入适当的值。要学习更多关于服务容器的注册细节，参考[其文档](#)。

契约列表

下面是 Laravel 契约列表，以及其对应的“门面”：

契约	对应门面
Illuminate\Contracts\Auth\Access\Authorizable	
Illuminate\Contracts\Auth\Access\Gate	Gate
Illuminate\Contracts\Auth\Authenticatable	
Illuminate\Contracts\Auth\CanResetPassword	
Illuminate\Contracts\Auth\Factory	Auth
Illuminate\Contracts\Auth\Guard	Auth::guard()
Illuminate\Contracts\Auth>PasswordBroker	Password::broker()
Illuminate\Contracts\Auth>PasswordBrokerFactory	Password
Illuminate\Contracts\Auth\StatefulGuard	
Illuminate\Contracts\Auth\SupportsBasicAuth	
Illuminate\Contracts\Auth\UserProvider	
Illuminate\Contracts\Bus\Dispatcher	Bus
Illuminate\Contracts\Bus\QueueingDispatcher	Bus::dispatchToQueue()
Illuminate\Contracts\Broadcasting\Factory	Broadcast
Illuminate\Contracts\Broadcasting\Broadcaster	Broadcast::connection()
Illuminate\Contracts\Broadcasting\ShouldBroadcast	
Illuminate\Contracts\Broadcasting\ShouldBroadcastNow	
Illuminate\Contracts\Cache\Factory	Cache
Illuminate\Contracts\Cache\Lock	
Illuminate\Contracts\Cache\LockProvider	
Illuminate\Contracts\Cache\Repository	Cache::driver()
Illuminate\Contracts\Cache\Store	
Illuminate\Contracts\Config\Repository	Config
Illuminate\Contracts\Console\Application	
Illuminate\Contracts\Console\Kernel	Artisan
Illuminate\Contracts\Container\Container	App
Illuminate\Contracts\Cookie\Factory	Cookie

契约	对应门面
Illuminate\Contracts\Cookie\QueueingFactory	Cookie::queue()
Illuminate\Contracts\Database\ModelIdentifier	
Illuminate\Contracts\Debug\ExceptionHandler	
Illuminate\Contracts\Encryption\Encrypter	Crypt
Illuminate\Contracts\Events\Dispatcher	Event
Illuminate\Contracts\Filesystem\Cloud	Storage::cloud()
Illuminate\Contracts\Filesystem\Factory	Storage
Illuminate\Contracts\Filesystem\Filesystem	Storage::disk()
Illuminate\Contracts\Foundation\Application	App
Illuminate\Contracts\Hashing\Hasher	Hash
Illuminate\Contracts\Http\Kernel	
Illuminate\Contracts\Logging\Log	Log
Illuminate\Contracts\Mail\MailQueue	Mail::queue()
Illuminate\Contracts\Mail\Mailable	
Illuminate\Contracts\Mail\Mailer	Mail
Illuminate\Contracts\Notifications\Dispatcher	Notification
Illuminate\Contracts\Notifications\Factory	Notification
Illuminate\Contracts\Pagination\LengthAwarePaginator	
Illuminate\Contracts\Pagination\Paginator	
Illuminate\Contracts\Pipeline\Hub	
Illuminate\Contracts\Pipeline\Pipeline	
Illuminate\Contracts\Queue\EntityResolver	
Illuminate\Contracts\Queue\Factory	Queue
Illuminate\Contracts\Queue\Job	
Illuminate\Contracts\Queue\Monitor	Queue
Illuminate\Contracts\Queue\Queue	Queue::connection()
Illuminate\Contracts\Queue\QueueableCollection	
Illuminate\Contracts\Queue\QueueableEntity	
Illuminate\Contracts\Queue\ShouldQueue	
Illuminate\Contracts\Redis\Factory	Redis
Illuminate\Contracts\Routing\BindingRegistrar	Route
Illuminate\Contracts\Routing\Registrar	Route
Illuminate\Contracts\Routing\ResponseFactory	Response
Illuminate\Contracts\Routing\UrlGenerator	URL
Illuminate\Contracts\Routing\UrlRoutable	

契约	对应门面
Illuminate\Contracts\Session\Session	Session::driver()
Illuminate\Contracts\Support\Arrayable	
Illuminate\Contracts\Support\Htmlable	
Illuminate\Contracts\Support\Jsonable	
Illuminate\Contracts\Support\MessageBag	
Illuminate\Contracts\Support\MessageProvider	
Illuminate\Contracts\Support\Renderable	
Illuminate\Contracts\Support\Responsable	
Illuminate\Contracts\Translation\Loader	
Illuminate\Contracts\Translation\Translator	Lang
Illuminate\Contracts\Validation\Factory	Validator
Illuminate\Contracts\Validation\ImplicitRule	
Illuminate\Contracts\Validation\Rule	
Illuminate\Contracts\Validation\ValidatesWhenResolved	
Illuminate\Contracts\Validation\Validator	Validator::make()
Illuminate\Contracts\View\Engine	
Illuminate\Contracts\View\Factory	View
Illuminate\Contracts\View\View	View::make()

四、基础组件

路由

路由入门

最基本的 Laravel 路由只接收一个 URI 和一个闭包，并以此为基础提供一个非常简单优雅的路由定义方法：

```
Route::get('hello', function () {
    return 'Hello, Welcome to LaravelAcademy.org';
});
```

我们以在[安装配置](#)文档中新建的 `blog` 应用为例，在 `routes/web.php` 中定义该路由：



```
<?php
/*
Route::get('/', function () {
    return view('welcome');
});

Route::get('hello', function () {
    return 'Hello, welcome to LaravelAcademy.org';
});|
```

在浏览器中通过 `http://blog.test/hello`（我使用 Valet 作为开发环境，故而对应域名是 `blog.test`，实际域名以自己配置的为准）即可访问我们刚刚定义的路由，页面输出内容如下：

```
Hello, welcome to LaravelAcademy.org
```

默认路由文件

所有 Laravel 路由都定义在位于 `routes` 目录下的路由文件中，这些文件通过框架自动加载，相应逻辑位于 `app/Providers/RouteServiceProvider` 类。`routes/web.php` 文件定义了 Web 界面的路由，这些路由被分配到了 `web` 中间件组，从而可以使 Session 和 CSRF 保护等功能。`routes/api.php` 中的路由是无状态的，这是因为被分配到了 `api` 中间件组。对大多数应用而言，都是从 `routes/web.php` 文件开始定义路由。定义在 `routes/web.php` 中的路由可以通过在浏览器地址栏输入相应的 URL 进行访问，例如，你可以通过 `http://blog.test/user` 访问下面的路由：

```
Route::get('/user', 'UsersController@index');
```

正如前面所提到的，定义在 `routes/api.php` 文件中的路由通过 `app/Providers/RouteServiceProvider` 的处理被嵌套在一个路由群组中，在这个群组中，所有路由会被自动添加 `/api` 前缀，所以你不需要再到路由文件中为每个路由手动添加，你可以通过编辑 `RouteServiceProvider` 类来修改路由前缀以及其他路由群组选项：

```
/**
 * Define the "api" routes for the application.
 *
 * These routes are typically stateless.
 *
 * @return void
 */
protected function mapApiRoutes()
{
    Route::prefix('api')
        ->middleware('api')
        ->namespace($this->namespace)
        ->group(base_path('routes/api.php'));
}
```

[[查看](#)]

有效的路由方法

我们可以注册路由来响应任何 HTTP 请求动作：

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

有时候还需要注册一个路由响应多种 HTTP 请求动作 —— 这可以通过 `match` 方法来实现。或者，可以使用 `any` 方法注册一个路由来响应所有 HTTP 请求动作：

```
Route::match(['get', 'post'], 'foo', function () {
    return 'This is a request from get or post';
});

Route::any('bar', function () {
    return 'This is a request from any HTTP verb';
});
```

测试 GET 请求的时候直接在浏览器中输入请求地址即可，测试 POST 请求可以通过客户端工具，比如 Advanced REST Client，该工具可以在 Chrome 应用商店下载到，此外如果上面的路由是定义在 `routes/web.php` 的话，在测试 POST 请求之前，需要将对应路由取消 CSRF 保护检查，否则会返回 `419` 状态码导致无法请求成功，取消的方法是在 `app/Http/Middleware/VerifyCsrfToken` 中设置排除检查路由：



```
<?php

namespace App\Http\Middleware;

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

class VerifyCsrfToken extends Middleware
{
    /**
     * The URIs that should be excluded from CSRF verification.
     *
     * @var array
     */
    protected $except = [
        'foo', 'bar'
    ];
}
```

下面我们来测试下 POST 请求：

Raw headers: Content-Type: text/html

Raw payload: This is a request from get or post

Response: 200 OK 78.00 ms

Parsed: This is a request from get or post

如果路由是定义在 `routes/api.php` 的话，则无需关注 CSRF 保护问题，比如我们在 `routes/api.php` 定义 `bar` 路由，并且在 `VerifyCsrfToken` 的 `$except` 属性数组中移除 `bar`，然后我们测试下对 `http://blog.test` 的 POST 请求：

Raw headers: Content-Type: text/html

Raw payload: This is a request from any HTTP verb

Response: 200 OK 82.00 ms

Parsed: This is a request from any HTTP verb

正如我们所预测的，完全没有任何问题，背后的原因是因为 `web` 路由文件中定义的路由都位于 `web` 中间件群组，该群组默认启用 CSRF 保护检查，而 `api` 路由文件位于 `api` 路由群组，该群组下的路由主要用于第三方 API 请求，没办法进行 CSRF 检查，所以不需要做任何处理。

CSRF 保护

在 `routes/web.php` 路由文件中所有请求方式为 `PUT`、`POST` 或 `DELETE` 的路由对应的 HTML 表单都必须包含一个 CSRF 令牌字段，否则，请求会被拒绝。关于 CSRF 的更多细节，可以参考 [CSRF 文档](#)：

```
<form method="POST" action="/profile">
    {{ csrf_field() }}
    ...
</form>
```

还是以上面的 `foo` 路由为例，如果我们不在 `VerifyCsrfToken` 中间件中排除对它的检查（事实上，这样的操作也不安全），那么就需要在表单提交中带上 `csrf_token` 字段：

```
Route::match(['get', 'post'], 'foo', function () {
    return 'This is a request from get or post';
});

Route::get('form', function () {
    return '<form method="POST" action="/foo">' . csrf_field() . '<button type="submit">提交</button></form>';
});
```

[2]

这样，当我们访问 `http://blog.test/form` 然后在页面点击提交按钮后，页面会跳转到 `http://blog.test/foo` 并显示如下内容：

```
This is a request from get or post
```

路由重定向

如果你需要定义一个重定向到其他 URI 的路由，可以使用 `Route::redirect` 方法，该方法非常方便，以至于你不需要再定义额外的路由或控制器来执行简单的重定向逻辑：

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

其中 `here` 表示原路由，`there` 表示重定向之后的路由，`301` 是一个 HTTP 状态码，用于标识重定向。

路由视图

如果你的路由需要返回一个视图，可以使用 `Route::view` 方法，和 `redirect` 方法类似，这个方法也很方便，以至于你不需要在额外定义一个路由或控制器。`view` 方法接收一个 URI 作为第一个参数，以及一个视图名称作为第二个参数，此外，你还可以提供一个数组数据传递到该视图方法作为可选的第三个参数，该数组数据可用于视图中的数据渲染：

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

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

我们在 `routes/web.php` 定义一个路由视图如下：

```
Route::get('/', function () {
    return view('welcome', ['website' => 'Laravel']);
});

Route::view('view', 'welcome', ['website' => ' Laravel学院']);
```

为了保证可以共用 `welcome.blade.php` 这个视图文件，我们也对默认提供的 `/` 路由做了调整，接下来，我们需要修改 `resources/views/welcome.blade.php` 代码以支持 `website` 数据变量：

```
<div class="content">
    <div class="title m-b-md">
        {{ $website }}
    </div>

    <div class="links">
        <a href="https://laravel.com/docs">Documentation</a>
        <a href="https://laracasts.com">Laracasts</a>
        <a href="https://laravel-news.com">News</a>
        <a href="https://forge.laravel.com">Forge</a>
        <a href="https://github.com/laravel/laravel">GitHub</a>
    </div>
</div>
```

我们将原来写死的 `Laravel` 文本调整为支持变量传入的方式，这样，我们就可以在浏览器中通过 `http://blog.test/view` 访问路由视图了：

Laravel 学院

[DOCUMENTATION](#)[LARACASTS](#)[NEWS](#)[FORGE](#)[GITHUB](#)

路由参数

必选参数

有时我们需要在路由中获取 URI 请求参数。例如，如果要从 URL 中获取用户 ID，需要通过如下方式定义路由参数：

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

这样我们在浏览器中访问 `http://blog.test/user/1`，就会得到以下输出：

```
User 1
```

可以根据需要在路由中定义多个路由参数：

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

根据上面的示例，路由参数需要通过花括号 `{}` 进行包裹并且是拼音字母，这些参数在路由被执行时会被传递到路由的闭包。路由参数名称不能包含 `-` 字符，如果需要的话可以使用 `_` 替代，比如如果某个路由参数定义成 `{post-id}` 则访问路由会报错，应该修改成 `{post_id}` 才行。路由参数被注入到路由回调/控制器取决于它们的顺序，与回调/控制器名称无关。

可选参数

有必选参数就有可选参数，这可以通过在参数名后加一个 `?` 标记来实现，这种情况下需要给相应的变量指定默认值，当对应的路由参数为空时，使用默认值：

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

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

这时如果定义的路由是下面这个的话，访问 `http://blog.test/user` 会返回 `John`。

正则约束

可以通过路由实例上的 `where` 方法来约束路由参数的格式。`where` 方法接收参数名和一个正则表达式来定义该参数如何被约束：

```
Route::get('user/{name}', function ($name) {
    // $name 必须是字母且不能为空
})->where('name', '[A-Za-z]+');

Route::get('user/{id}', function ($id) {
    // $id 必须是数字
})->where('id', '[0-9]+');

Route::get('user/{id}/{name}', function ($id, $name) {
```

```
// 同时指定 id 和 name 的数据格式
})->where(['id' => '[0-9]+', 'name' => '[a-z]+']);
```

使用正则约束还有一个好处就是避免了 `user/{id}` 和 `user/{name}` 的混淆。

全局约束

如果想要路由参数在全局范围内被给定正则表达式约束，可以使用 `pattern` 方法。需要在 `RouteServiceProvider` 类的 `boot` 方法中定义这种约束模式：

```
/**
 * 定义路由模型绑定，模式过滤器等
 *
 * @param \Illuminate\Routing\Router $router
 * @return void
 * @translator http://laravelacademy.org
 */
public function boot()
{
    Route::pattern('id', '[0-9]+');
    parent::boot();
}
```

一旦模式被定义，将会自动应用到所有包含该参数名的路由中：

```
Route::get('user/{id}', function ($id) {
    // 只有当 {id} 是数字时才会被调用
});
```

除此之外，该模式还会被应用到诸如下面这些路由参数上：

```
Route::get('post/{id}', function ($id) {
    // 只有当 {id} 是数字时才会被调用
});

Route::get(`product/{id}', function ($id) {
    // 只有当 {id} 是数字时才会被调用
});
```

很显然这种方式让代码更简洁，也为我们实现同一参数统一约束带来了方便。

命名路由

命名路由为生成 URL 或重定向提供了方便，实现起来也很简单，在路由定义之后使用 `name` 方法链的方式来定义该路由的名称：

```
Route::get('user/profile', function () {
    // 通过路由名称生成 URL
    return 'my url: ' . route('profile');
})->name('profile');
```

还可以为控制器动作指定路由名称：

```
Route::get('user/profile', 'UserController@showProfile')->name('profile');
```

这样我们就可以通过以下方式定义重定向：

```
Route::get('redirect', function() {
    // 通过路由名称进行重定向
    return redirect()->route('profile');
});
```

为命名路由生成 URL

正如上面代码所展示的，为给定路由分配名称之后，就可以通过辅助函数 `route` 为该命名路由生成 URL 或者通过 `redirect` 函数进行重定向：

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

// 生成重定向
return redirect()->route('profile');
```

如果命名路由定义了参数，可以将该参数作为第二个参数传递给 `route` 函数。给定的路由参数将会自动插入到 URL 中：

```
Route::get('user/{id}/profile', function ($id) {
    $url = route('profile', ['id' => 1]);
    return $url;
})->name('profile');
```

这样，当我们访问 `http://blog.test/user/123/profile` 页面输出内容也是 `http://blog.test/user/123/profile`。

检查当前路由

如果你想要判断当前请求是否被路由到给定命名路由，可以使用 `Route` 实例上的 `named` 方法，例如，你可以从路由中间件中检查当前路由名称：

```
/**
 * 处理输入请求
 *
 * @param \Illuminate\Http\Request $request
 * @param Closure $next
 * @return mixed
 */
public function handle($request, Closure $next)
{
    if ($request->route()->named('profile')) {
        //
    }

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

路由分组

路由分组的目的是让我们在多个路由中共享相同的路由属性，比如中间件和命名空间等，这样的话我们定义了大量的路由时就不必为每一个路由单独定义属性。共享属性以数组的形式作为第一个参数被传递给 `Route::group` 方法。

中间件

要给某个路由分组中定义的所有路由分配中间件，可以在定义分组之前使用 `middleware` 方法。中间件将会按照数组中定义的顺序依次执行：

```
Route::middleware(['first', 'second'])->group(function () {
    Route::get('/', function () {
        // Uses first & second Middleware
    });

    Route::get('user/profile', function () {
        // Uses first & second Middleware
    });
});
```

关于中间件的使用我们在后面单独讲 [中间件](#) 时再进行示例演示，这里我们先了解这样使用就行。

命名空间

路由分组另一个通用的例子是使用 `namespace` 方法分配同一个 PHP 命名空间给该分组下的多个控制器：

```
Route::namespace('Admin')->group(function () {
    // Controllers Within The "App\Http\Controllers\Admin" Namespace
});
```

默认情况下，`RouteServiceProvider` 在一个命名空间分组下引入所有路由文件，并指定所有控制器类所在的默认命名空间是 `App\Http\Controllers`，因此，我们在定义控制器的时候只需要指定命名空间 `App\Http\Controllers` 之后的部分即可。关于命名空间后面我们单独讲控制器的时候还会再详细演示，这里先了解用法即可。

子域名路由

路由分组还可以被用于处理子域名路由，子域名可以像 URI 一样被分配给路由参数，从而允许捕获子域名的部分用于路由或者控制器，子域名可以在定义分组之前调用 `domain` 方法来指定：

```
Route::domain('{account}.blog.dev')->group(function () {
    Route::get('user/{id}', function ($account, $id) {
        return 'This is ' . $account . ' page of User ' . $id;
    });
});
```

比如我们设置会员子域名为 `account.blog.test`，那么就可以通过 `http://account.blog.test/user/1` 访问用户 ID 为 1 的会员信息了：

```
This is account page of User 1
```

路由前缀

`prefix` 方法可以用来为分组中每个路由添加一个给定 URI 前缀，例如，你可以为分组中所有路由 URI 添加 `admin` 前缀：

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

这样我们就可以通过 `http://blog.test/admin/users` 访问路由了。

路由名称前缀

`name` 方法可通过传入字符串为分组中的每个路由名称设置前缀，例如，你可能想要在所有分组路由的名称前添加 `admin` 前缀，由于给定字符串和指定路由名称前缀字符串完全一样，所以需要在前缀字符串末尾后加上 `.` 字符：

```
Route::name('admin.')->group(function () {
    Route::get('users', function () {
        // 新的路由名称为 "admin.users"...
    })->name('users');
});
```

路由模型绑定

注入模型 ID 到路由或控制器动作时，通常需要查询数据库才能获取相应的模型数据。Laravel 路由模型绑定让注入模型实例到路由变得简单，例如，你可以将匹配给定 ID 的整个 `User` 类实例注入到路由中，而不只是注入用户 ID。

隐式绑定

Laravel 会自动解析定义在路由或控制器动作（变量名匹配路由片段）中的 Eloquent 模型类型声明，例如（我们将这个路由定义在 `routes/api.php` 文件中）：

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

在这个例子中，由于类型声明了 Eloquent 模型 `App\User`，对应的变量名 `$user` 会匹配路由片段中的 `{user}`，这样，Laravel 会自动注入与请求 URI 中传入的 ID 对应的用户模型实例。如果匹配模型实例在数据库中不存在，会自动生成 404 响应。

在演示本功能之前，我们需要先创建数据表，由于我是在 Valet 开发环境中开发，需要自己创建数据库，我们将数据库命名为 `valet`，本地的数据库用户名为 `root`，密码为空，对应地，修改 `.env` 文件配置如下：

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=valet
DB_USERNAME=root
DB_PASSWORD=
```

具体配置值以你自己的开发环境设置为准。我们将基于 Laravel 强大的数据库迁移功能创建 `users` 表，关于数据库迁移后面在数据库部分会详细讨论，这里我们通过以下命令来生成 `users` 表即可：

```
php artisan migrate
```

进入数据库可以看到该表已经生成：

	Name	Rows	Data Length	Engine	Created Date	Modified Date	Collation	Comment
localhost	migrations	2	16.00 KB	InnoDB	2017-09-17 08:28:47	2017-09-17 08:28:47	utf8mb4_unicode_ci	
localhost	password_resets	0	16.00 KB	InnoDB	2017-09-17 08:28:47	2017-09-17 08:28:47	utf8mb4_unicode_ci	
localhost	users	0	16.00 KB	InnoDB	2017-09-17 08:28:47	2017-09-17 08:28:47	utf8mb4_unicode_ci	

这时，`users` 数据表还没有任何记录，如果数据库中找不到对应的模型实例，会自动生成 HTTP 404 响应，提示页面不存在，所以我们需要在这张表中插入一条记录，这里我们基于 Laravel 强大的数据库填充器来快速完成数据填充功能，首先通过如下命令生成 `users` 对应的数据表填充器：

```
php artisan make:seeder UsersTableSeeder
```

该命令会在 `database/seeds` 目录下生成一个 `UsersTableSeeder` 文件，编辑该文件内容如下：

```

<?php

use Illuminate\Database\Seeder;

class UsersTableSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        DB::table('users')->insert([
            'name' => str_random(length: 10),
            'email' => str_random(length: 10).'@laravelacademy.org',
            'password' => bcrypt(value: 'secret'),
        ]);
    }
}

```

然后编辑同目录下的 `DatabaseSeeder.php` 文件如下（取消调用数据表填充器前的注释）：

```

<?php

use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        $this->call(UsersTableSeeder::class);
    }
}

```

最后执行 `php artisan db:seed` 即可插入对应数据到 `users` 表了，这样我们在浏览器中再次访问 `http://blog.test/api/users/1` 的时候就会显示 `User` 模型数据了：

User {#209 ▼

```

#fillable: array:3 [▶]
#hidden: array:2 [▶]
#connection: "mysql"
#table: null
#primaryKey: "id"
#keyType: "int"
+incrementing: true
#with: []
#withCount: []
#perPage: 15
+exists: true
+wasRecentlyCreated: false
#attributes: array:7 [▶]
#original: array:7 [▼
    "id" => 1
    "name" => "jroJoGP71W"
    "email" => "Syin7cioqH@laravelacademy.org"
    "password" => "$2y$10$WX14KWPwQHrUI2Y.8ZMeKeROjyZV0GNJ0kDM7ATjNJI/dWxmylA9e"
    "remember_token" => null
    "created_at" => null
    "updated_at" => null
]
#changes: []
#casts: []
#dates: []
#dateFormat: null
#appends: []
#dispatchesEvents: []
#observables: []
#relations: []
#touches: []
+timestamps: true
#visible: []
#guarded: array:1 [▶]
#rememberTokenName: "remember_token"
}

```

接下来，你就可以在应用代码中直接拿 `$user` 模型去做你想做的事情了，而不需要自己去数据库查询，从而提高了开发的效率。

自定义键名

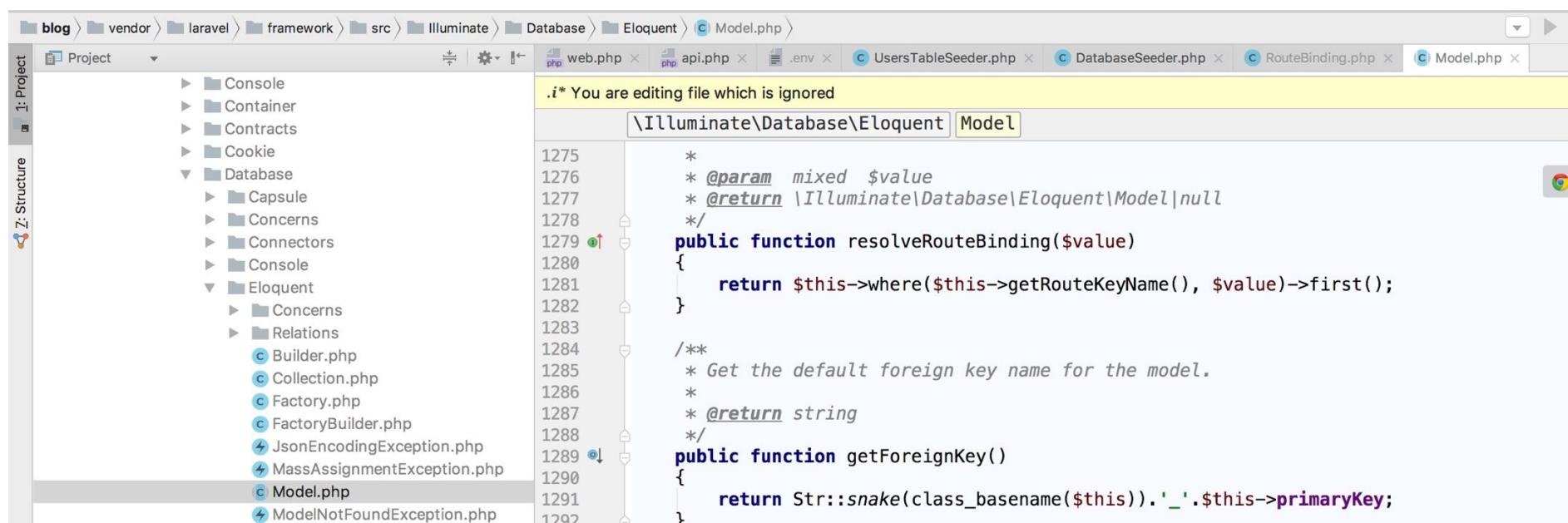
如果你想要在隐式模型绑定中使用数据表的其它字段而不是 `id` 字段，可以重写 Eloquent 模型类的 `getRouteKeyName` 方法，以 `User` 模型为例，可以在该模型类中添加这个方法：

```

/**
 * Get the route key for the model.
 *
 * @return string
 */
public function getRouteKeyName()
{
    return 'name';
}

```

这样我们就可以通过 `http://blog.test/api/users/jroJoGP71W` 访问同一个模型实例了。这里需要注意的点是如果该字段不是唯一键，则会返回结果集的第一条记录，对应的底层实现在这里：

**显式绑定**

有隐式绑定，就有显式绑定。要注册显式绑定，可以使用路由器的 `model` 方法来为给定参数指定绑定类。你需要在 `RouteServiceProvider` 类的 `boot` 方法中定义显式模型绑定：

```

public function boot()
{
}

```

```
parent::boot();
Route::model('user_model', App\User::class);
}
```

接下来，在 `routes/api.php` 中定义一个包含 `{user}` 参数的路由：

```
$router->get('profile/{user_model}', function(App\User $user) {
    dd($user);
});
```

由于我们已经绑定 `{user_model}` 参数到 `App\User` 模型，`User` 实例会被注入到该路由。因此，如果请求 URL 是 `http://blog.test/api/profile/1`，就会注入一个用户 ID 为 1 的 `User` 实例。

如果匹配的模型实例在数据库不存在，会自动生成并返回 HTTP 404 响应。

自定义解析逻辑

如果你想要使用自定义的解析逻辑，可以在 `RouteServiceProvider` 类的 `boot` 方法中使用 `Route::bind` 方法，传递到 `bind` 方法的闭包会获取到 URI 请求参数中的值，并且返回你想要在该路由中注入的类实例：

```
public function boot()
{
    parent::boot();

    Route::bind('user', function($value) {
        return App\User::where('name', $value)->first() ?? abort(404);
    });
}
```

有了这些方法，基本上可以满足你对路由模型绑定的各种需求了。

频率限制

Laravel 自带了一个中间件用于限制对应用路由的访问频率。开始使用该功能之前，分配 `throttle` 中间件到某个路由或路由分组，`throttle` 中间件接收两个参数用于判断给定时间内（单位：分钟）的最大请求次数。例如，我们指定登录用户每分钟只能访问下面的分组路由 60 次：

```
Route::middleware('auth:api', 'throttle:60,1')->group(function () {
    Route::get('/user', function () {
        //
    });
});
```

超出访问次数后，会返回 `429` 状态码并提示“Too many requests”。

动态频率限制

此外，还可以基于 `User` 模型的属性来动态设置最大请求次数。例如，如果 `User` 模型包含 `rate_limit` 属性，就可以将其这个属性名传递到 `throttle` 中间件，这样就可以将属性值作为计算最大请求数的数据来源：

```
Route::middleware('auth:api', 'throttle:rate_limit,1')->group(function () {
    Route::get('/user', function () {
        //
    });
});
```

表单方法伪造

HTML 表单不支持 `PUT`、`PATCH` 或者 `DELETE` 请求方法，因此，在 HTML 表单中调用 `PUT`、`PATCH` 或 `DELETE` 路由时，需要添加一个隐藏的 `_method` 字段，其值被用作该表单的 HTTP 请求方法：

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

还可以直接使用 Blade 指令 `@method` 来生成 `_method` 字段：

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

访问当前路由

你可以使用 `Route` 门面上的 `current`、`currentRouteName` 和 `currentRouteAction` 方法来访问处理当前输入请求的路由信息：

```
// 获取当前路由实例
$route = Route::current();
// 获取当前路由名称
```

```
$name = Route::currentRouteName();
// 获取当前路由 action 属性
$action = Route::currentRouteAction();
```

参考 API 文档了解[路由门面底层类](#)以及 [Route 实例](#)的更多可用方法。

中间件

简介

中间件为过滤进入应用的 HTTP 请求提供了一套便利的机制。例如，Laravel 内置了一个中间件来验证用户是否经过认证（如登录），如果用户没有经过认证，中间件会将用户重定向到登录页面，而如果用户已经经过认证，中间件就会允许请求继续往前进入下一步操作。

当然，除了认证之外，中间件还可以被用来处理很多其它任务。比如：CORS 中间件可以用于为离开站点的响应添加合适的头（跨域）；日志中间件可以记录所有进入站点的请求，从而方便我们构建系统日志系统。

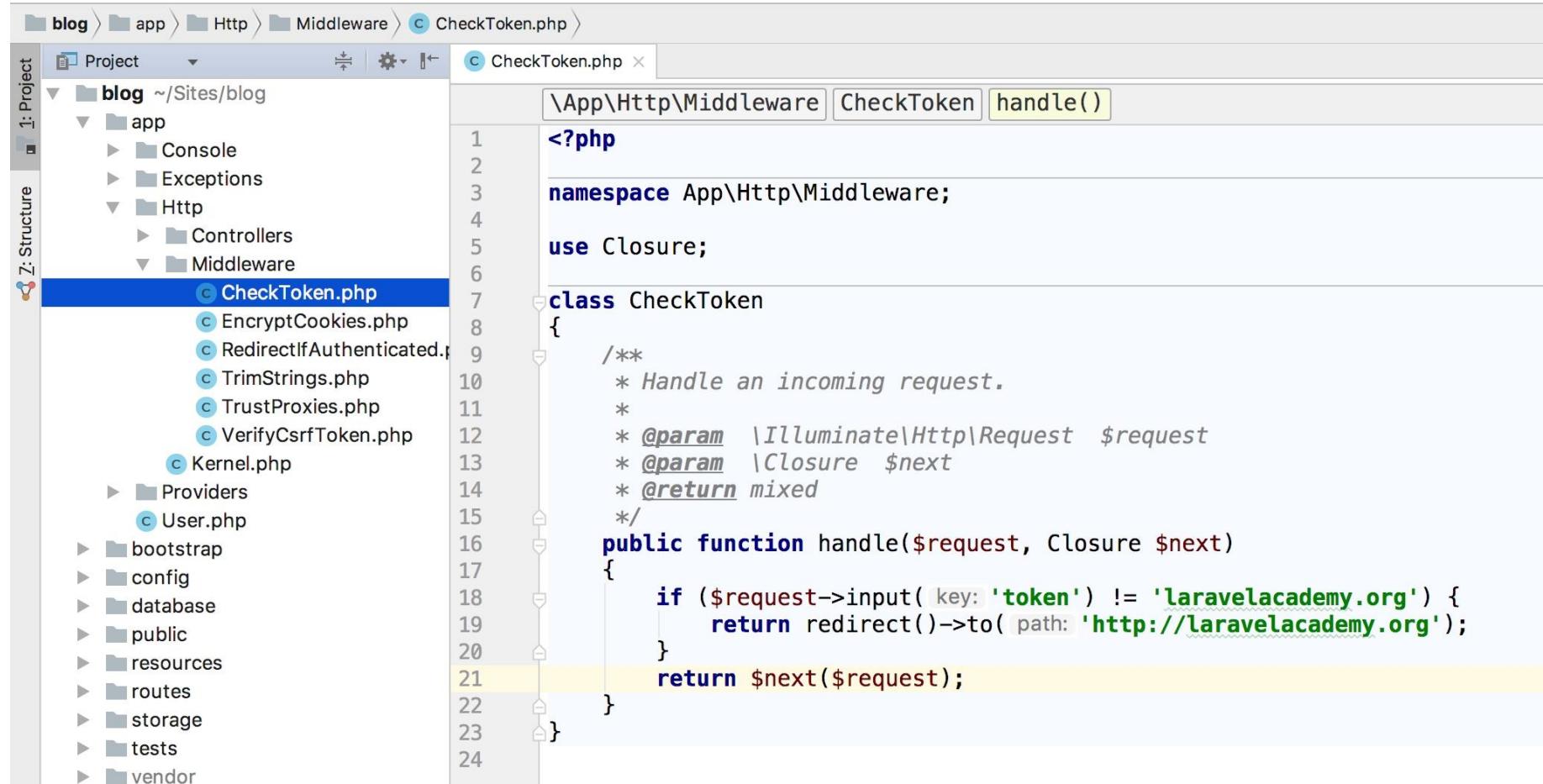
Laravel 框架自带了一些中间件，包括认证、CSRF 保护中间件等等。所有的中间件都位于 [app/Http/Middleware](#) 目录下。

定义中间件

要创建一个新的中间件，可以通过 Artisan 命令 `make:middleware`：

```
php artisan make:middleware CheckToken
```

这个命令会在 [app/Http/Middleware](#) 目录下创建一个新的中间件类 `CheckToken`，在这个中间件中，我们只允许提供的 `token` 等于指定值 `laravelacademy.org` 的请求访问路由，否则，我们将跳转到 Laravel 学院网站：



正如你所看到的，如果 `token != 'laravelacademy.org'`，中间件会返回一个 HTTP 重定向到 Laravel 学院；否则，请求会被传递下去。将请求往下传递可以通过调用回调函数 `$next` 并传入当前 `$request`。

注：此时只是定义好了中间件的逻辑，要让这个中间件生效，还要将其注册到指定路由中，我们很快就会在下面的注册中间件部分教你怎么做。

理解中间件的最好方式就是将中间件看做 HTTP 请求到达目标动作之前必须经过的“层”，每一层都会检查请求并且可以完全拒绝它。

请求之前/之后的中间件

一个中间件是请求前还是请求后执行取决于中间件本身。比如，以下中间件会在请求处理前执行一些任务：

```
<?php
namespace App\Http\Middleware;
use Closure;
class BeforeMiddleware
{
    public function handle($request, Closure $next)
    {
        // 执行动作
    }
}
```

```

    return $next($request);
}
}

```

而下面这个中间件则会在请求处理后执行其任务：

```

<?php

namespace App\Http\Middleware;

use Closure;

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

        // 执行动作

        return $response;
    }
}

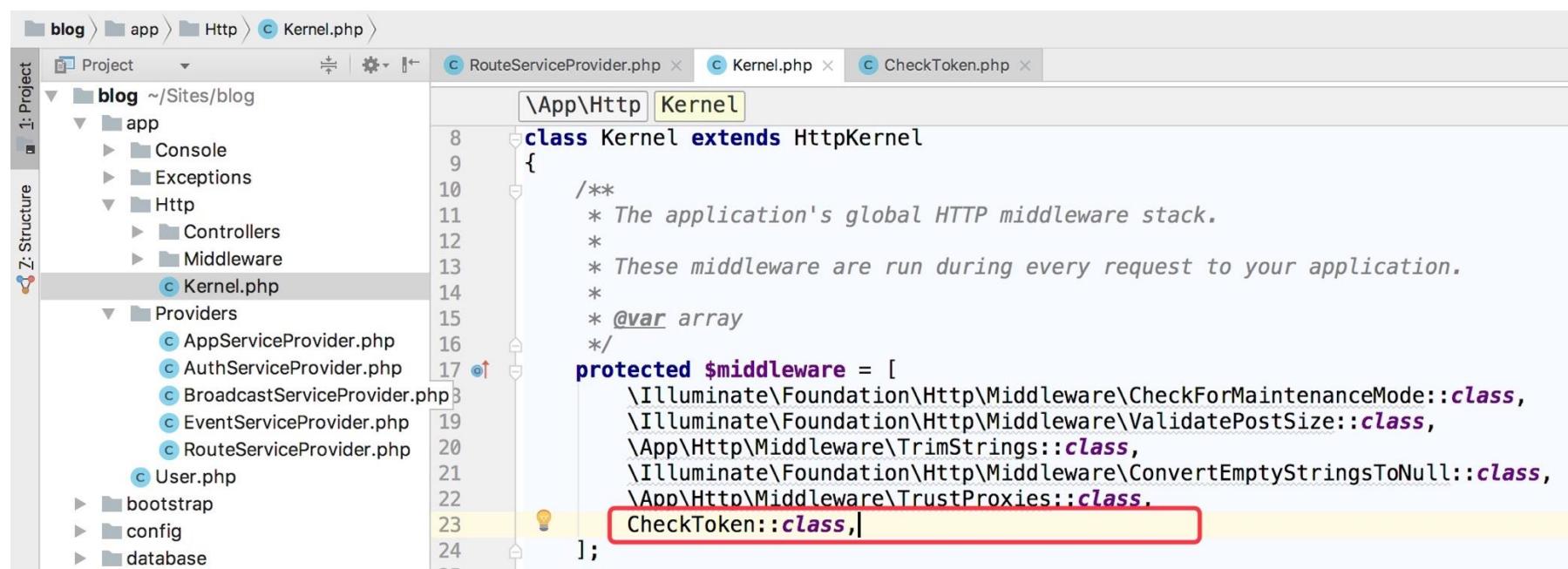
```

注册中间件

中间件分三类，分别是全局中间件、中间件组和指定路由中间件：

全局中间件

如果你想要定义的中间件在每一个 HTTP 请求时都被执行，只需要将相应的中间件类添加到 `app/Http/Kernel.php` 的数组属性 `$middleware` 中即可：



但除非真的需要，否则我们一般不会把业务级别的中间件放到全局中间件中。

分配中间件到指定路由

如果你想要分配中间件到指定路由，首先应该在 `app/Http/Kernel.php` 文件中分配给该中间件一个 `key`，默认情况下，该类的 `$routeMiddleware` 属性包含了 Laravel 自带的中间件，要添加你自己的中间件，只需要将其追加到后面并为其分配一个 `key`，例如：

```

// 在 App\Http\Kernel 类中...

/**
 * 应用的路由中间件列表
 *
 * 这些中间件可以分配给路由组或者单个路由
 *
 * @var array
 */
protected $routeMiddleware = [
    'auth' => \Illuminate\Auth\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
];

```

```
'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
'token' => CheckToken::class
];
```

中间件在 HTTP Kernel 中被定义后，可以使用 `middleware` 方法将其分配到路由：

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

这样，当我们在浏览器中访问 `http://blog.test` 时就会跳到 `http://laravelacademy.org`，只有当访问 `http://blog.test?token=laravelacademy.org` 时才能看到如下效果：


[DOCUMENTATION](#)
[LARACASTS](#)
[NEWS](#)
[FORGE](#)
[GITHUB](#)

可以使用数组分配多个中间件到路由：

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

分配中间件的时候还可以传递完整的类名（不过不推荐这种方式）：

```
use App\Http\Middleware\CheckToken;

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

中间件组

有时候你可能想要通过指定一个键名的方式将相关中间件分到同一个组里面，这样可以更方便地将其分配到路由中，这可以通过使用 HTTP Kernel 提供的 `$middlewareGroups` 属性实现。

Laravel 自带了开箱即用的 `web` 和 `api` 两个中间件组，分别包含可以应用到 Web 和 API 路由的通用中间件：

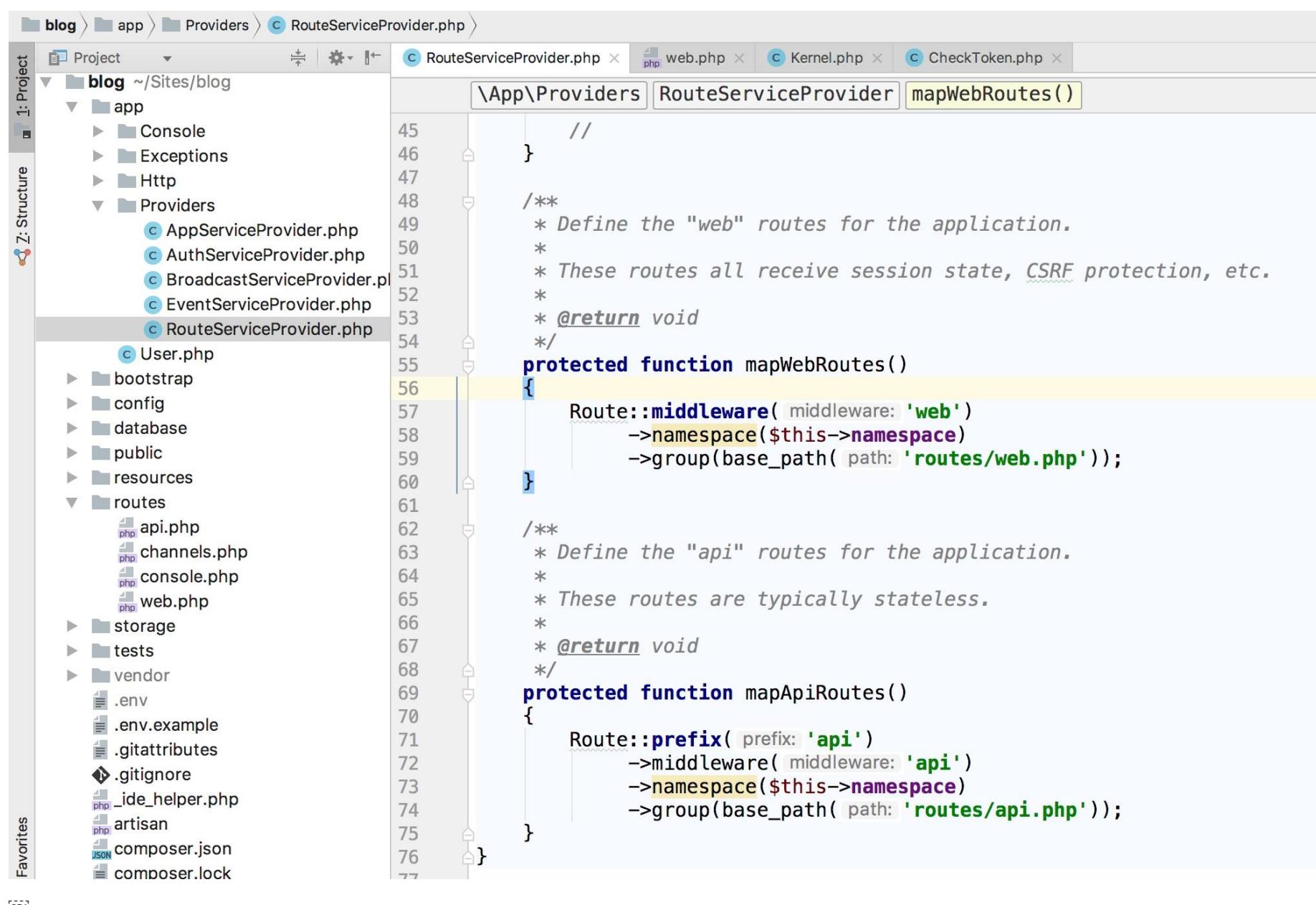
```
/**
 * 应用的中间件组
 *
 * @var array
 */
protected $middlewareGroups = [
    'web' => [
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \App\Http\Middleware\VerifyCsrfToken::class,
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],
    'api' => [
        'throttle:60,1',
        'auth:api',
    ],
];
```

中间件组使用和分配单个中间件同样的语法被分配给路由和控制器动作。再次申明，中间件组的目的只是让一次分配给路由多个中间件的实现更加方便：

```
Route::get('/', function () {
```

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

默认情况下，`RouteServiceProvider` 自动将中间件组 `web` 应用到 `routes/web.php` 文件，将中间件组 `api` 应用到 `routes/api.php`：



当然我们可以自己设置自己的中间件组，以实现更灵活的中间件分配策略：

```
/**  
 * 应用的中间件组.  
 *  
 * @var array  
 */  
protected $middlewareGroups = [  
    'web' => [  
        \App\Http\Middleware\EncryptCookies::class,  
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,  
        \Illuminate\Session\Middleware\StartSession::class,  
        // \Illuminate\Session\Middleware\AuthenticateSession::class,  
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,  
        \App\Http\Middleware\VerifyCsrfToken::class,  
        \Illuminate\Routing\Middleware\SubstituteBindings::class,  
    ],  
  
    'api' => [  
        'throttle:60,1',  
        'auth:api',  
    ],  
  
    'blog' => [  
        'token',  
    ]  
];
```

我们修改 `routes/web.php` 下面的中间件分配方式：

```
Route::group(['middleware'=>['blog']],function(){  
    Route::get('/', function () {  
        return view('welcome', ['website' => 'Laravel']);  
    });
```

```
Route::view('/view', 'welcome', ['website' => 'Laravel 学院']);
});
```

这样我们访问 `http://blog.test` 和 `http://blog.test/view` 的时候都要带上 `token=laravelacademy.org` 参数，否则就会跳转到 Laravel 学院网站。

中间件参数

中间件还可以接收额外的自定义参数，例如，如果应用需要在执行给定动作之前验证认证用户是否拥有指定的角色，可以创建一个 `CheckRole` 来接收角色名作为额外参数。

额外的中间件参数会在 `$next` 参数之后传入中间件：

```
<?php

namespace App\Http\Middleware;

use Closure;

class CheckRole
{
    /**
     * 处理输入请求
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure $next
     * @param string $role
     * @return mixed
     * translator http://laravelacademy.org
     */
    public function handle($request, Closure $next, $role)
    {
        if (! $request->user()->hasRole($role)) {
            // Redirect...
        }

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

中间件参数可以在定义路由时通过 `:` 分隔中间件名和参数名来指定，多个中间件参数可以通过逗号分隔：

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

根据上面的演示示例，这个功能实现起来也比较简单，就不再单独演示了。

终端中间件

终端中间件，可以理解为一个善后的后台处理中间件。有时候中间件可能需要在 HTTP 响应发送到浏览器之后做一些工作，比如，Laravel 内置的 `session` 中间件会在响应发送到浏览器之后将 Session 数据写到存储器中，为了实现这个功能，需要定义一个终止中间件并添加 `terminate` 方法到这个中间件：

```
<?php

namespace Illuminate\Session\Middleware;

use Closure;

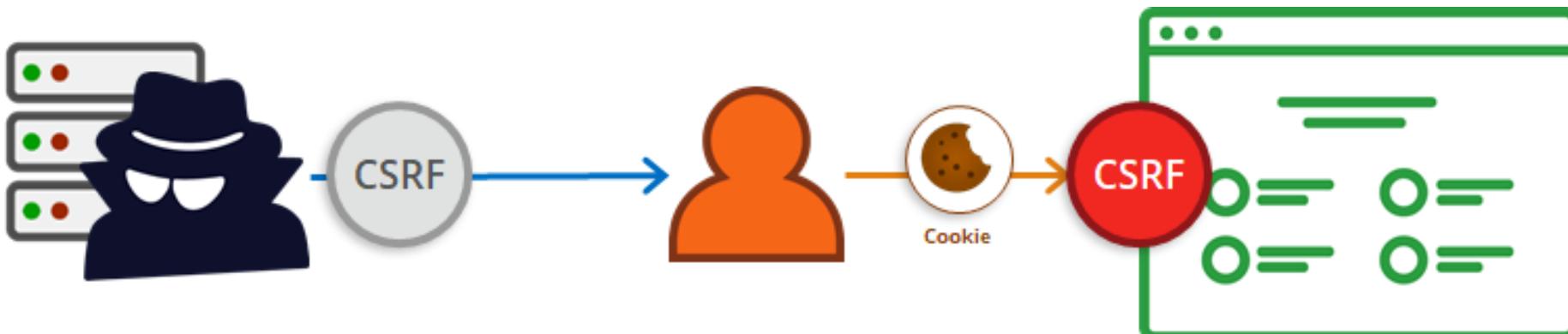
class StartSession
{
    public function handle($request, Closure $next)
    {
        return $next($request);
    }

    public function terminate($request, $response)
    {
        // 存储 session 数据...
    }
}
```

}

`terminate` 方法将会接收请求和响应作为参数。定义了一个终端中间件之后，还需要将其加入到 `app/Http/Kernel.php` 文件的全局中间件列表中。当调用中间件上的 `terminate` 方法时，Laravel 将会从服务容器中取出一个该中间件的新实例，如果你想要在调用 `handle` 和 `terminate` 方法时使用同一个中间件实例，则需要使用容器提供的 `singleton` 方法以单例的方式将该中间件注册到容器中。关于服务容器我们就会在后面讲到，暂时不深入展开了。

CSRF 保护



简介

跨站请求伪造（CSRF）是一种通过伪装授权用户的请求来攻击授信网站的恶意漏洞。

Laravel 通过自带的 CSRF 保护中间件让避免应用遭到跨站请求伪造攻击变得简单：Laravel 会自动为每一个被应用管理的有效用户会话生成一个 CSRF “令牌”，然后将该令牌存放在 Session 中，该令牌用于验证授权用户和发起请求者是否是同一个人。

任何时候在 Laravel 应用中定义 HTML 表单，都需要在表单中引入 CSRF 令牌字段，这样 CSRF 保护中间件才能够对请求进行验证。要想生成包含 CSRF 令牌的隐藏输入字段，可以使用 Blade 指令 `@csrf`：

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

中间件组 `web` 中的中间件 `VerifyCsrfToken` 会自动为我们验证请求输入的 token 值和 Session 中存储的 token 是否一致，如果没有传递该字段或者传递过来的字段值和 Session 中存储的数值不一致，则会抛出异常。

为了演示该功能，我们在 `routes/web.php` 中定义一组测试路由：

```
Route::get('form_without_csrf_token', function () {
    return '<form method="POST" action="/hello_from_form"><button type="submit">提交</button></form>';
});

Route::get('form_with_csrf_token', function () {
    return '<form method="POST" action="/hello_from_form">' . csrf_field() . '<button type="submit">提交</button></form>';
});

Route::get('hello_from_form', function () {
    return 'hello laravel!';
});
```

我们在浏览器中访问 `http://blog.test/form_without_csrf_token` 并点击页面上的提交按钮时，页面报错，抛出 `MethodNotAllowedHttpException` 异常，出现这个异常往往就是意味着没有传递 CSRF 令牌字段或者传递的令牌字段不正确：

而当我们访问 `http://blog.test/form.csrf.token` 并点击页面上的提交按钮时，页面显示正常。

注：CSRF 中间件只只作用于 `routes/web.php` 中定义的路由，因为该文件下的路由分配了 `web` 中间件组，而 `VerifyCsrfToken` 位于 `web` 中间件组中。

CSRF 令牌 & JavaScript

构建 JavaScript 驱动的应用时，为方便起见，可以让 JavaScript HTTP 库自动在每个请求中添加 CSRF 令牌。默认情况下，`resources/assets/js/bootstrap.js` 文件会将 `csrf-token` meta 标签值注册到 Axios HTTP 库。如果你没有使用这个库，则需要手动在应用中配置该实现。

排除指定 URL

有时候我们需要从 CSRF 保护中间件中排除一些 URL，例如，如果你使用了第三方支付系统（如支付宝或微信支付）来处理支付并用到他们提供的回调功能，这时候就需要从 Laravel 的 CSRF 保护中间件中排除回调处理器路由，因为第三方支付系统并不知道要传什么 token 值给我们定义的路由。

通常我们需要将这种类型的路由放到文件 `routes/web.php` 之外，比如 `routes/api.php`。不过，如果必须要加到 `routes/web.php` 中的话，你也可以在 `VerifyCsrfToken` 中间件中将要排除的 URL 添加到 `$except` 属性数组：

```
<?php

namespace App\Http\Middleware;

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

class VerifyCsrfToken extends Middleware
{
    /**
     * 从 CSRF 验证中排除的 URL
     *
     * @var array
     */
    protected $except = [
        'alipay/*',
        'http://example.com/foo/bar',
        'http://example.com/foo/*',
    ];
}
```

注：运行测试时 CSRF 中间件会自动禁止。

X-CSRF-Token

除了将 CSRF 令牌作为 POST 参数进行验证外，还可以通过设置 `X-CSRF-Token` 请求头来实现验证，`VerifyCsrfToken` 中间件会检查 `X-CSRF-TOKEN` 请求头。实现方式如下，首先创建一个 `meta` 标签并将令牌保存到该 `meta` 标签：

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

然后在 js 库（如 jQuery）中添加该令牌到所有请求头，这为基于 AJAX 的请求提供了简单、方便的方式来避免 CSRF 攻击：

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

X-XSRF-Token

Laravel 还会将 CSRF 令牌保存到名为 `XSRF-TOKEN` 的 Cookie 中，你可以使用该 Cookie 值来设置 `X-XSRF-TOKEN` 请求头。一些 JavaScript 框架，比如 Angular 和 Axios，会为你自动进行上述设置，基本上你不太需要手动设置这个值。

最后，`VerifyCsrfToken` 中间件框架底层实现源码位于 `vendor/laravel/framework/src/Illuminate/Foundation/Http/Middleware/VerifyCsrfToken.php`，感兴趣的同学可以去一窥究竟。

控制器

简介

我们之前的演示示例都是将所有的请求处理逻辑放在路由文件的闭包函数中，这显然是不合理的，我们需要使用控制器类组织管理相对复杂的业务逻辑处理。控制器用于将相关的 HTTP 请求封装到一个类中进行处理，这些控制器类存放在 `app/Http/Controllers` 目录下。

控制器入门

定义控制器

下面是一个基本控制器类的例子。首先我们使用 Artisan 命令快速创建一个控制器：

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

所有的 Laravel 控制器应该继承自 Laravel 自带的控制器基类 `App\Http\Controllers\Controller`，我们为该控制器添加一个 `show` 方法：

```
<?php

namespace App\Http\Controllers;

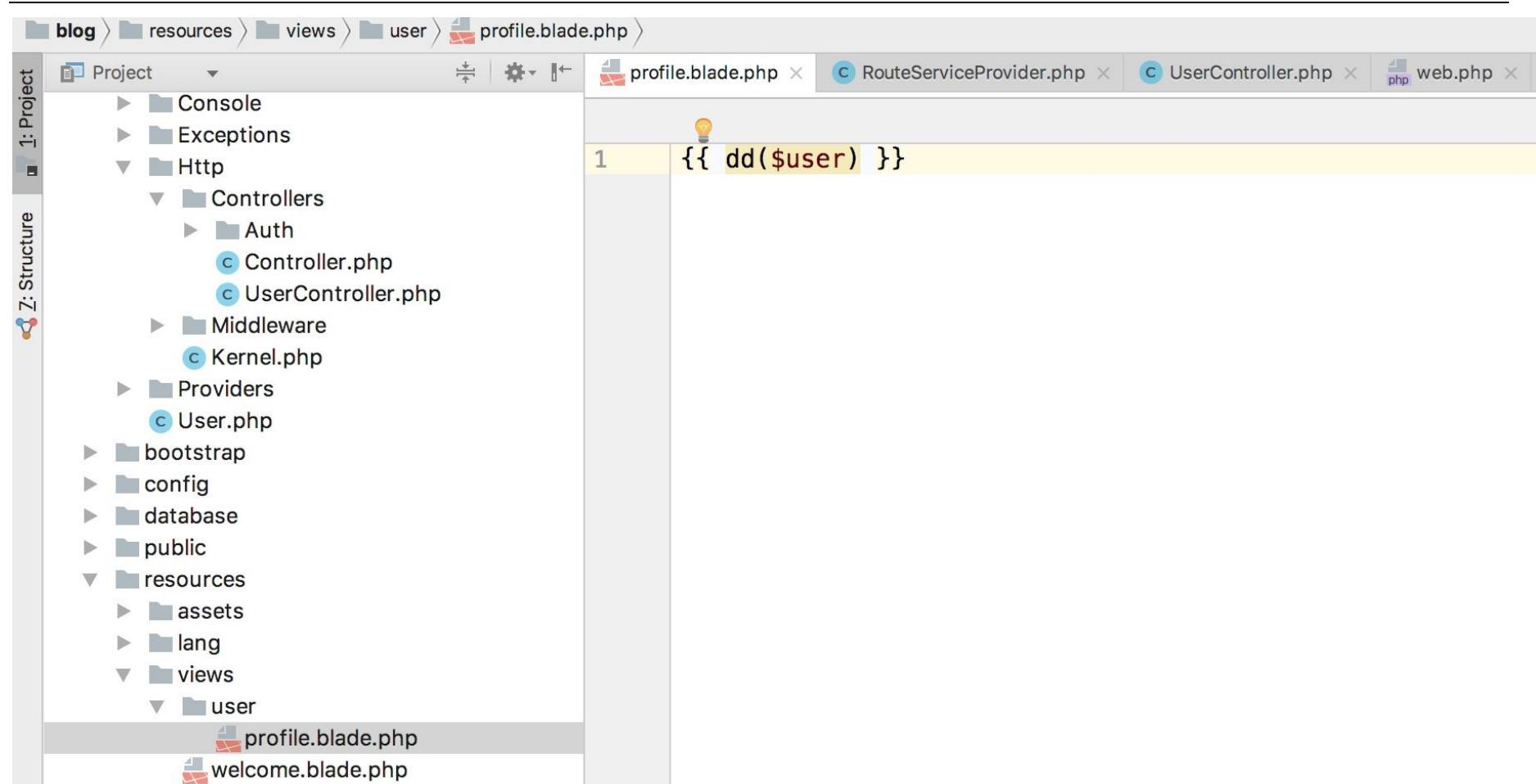
use App\User;
use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * 为指定用户显示详情
     *
     * @param int $id
     * @return Response
     * @author LaravelAcademy.org
     */
    public function show($id)
    {
        return view('user.profile', ['user' => User::findOrFail($id)]);
    }
}
```

我们可以像这样定义指向该控制器动作的路由：

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

现在，如果一个请求匹配上面的路由 URI，`UserController` 的 `show` 方法就会被执行，当然，路由参数也会被传递给这个方法。此外，这里 `show` 方法里面还用到了 `view` 方法，该方法用于将 `user` 变量渲染到 `user/profile` 视图中，后面我们讲视图的时候会继续讨论该方法的使用，现在我们只是做简单演示，在 `resources/views` 目录下创建 `user` 子目录，然后在 `user` 目录下新建 `profile.php` 文件，编辑文件内容如下：



这样我们在浏览器中访问 `http://blog.test/user/1`, 就会看到打印结果了:

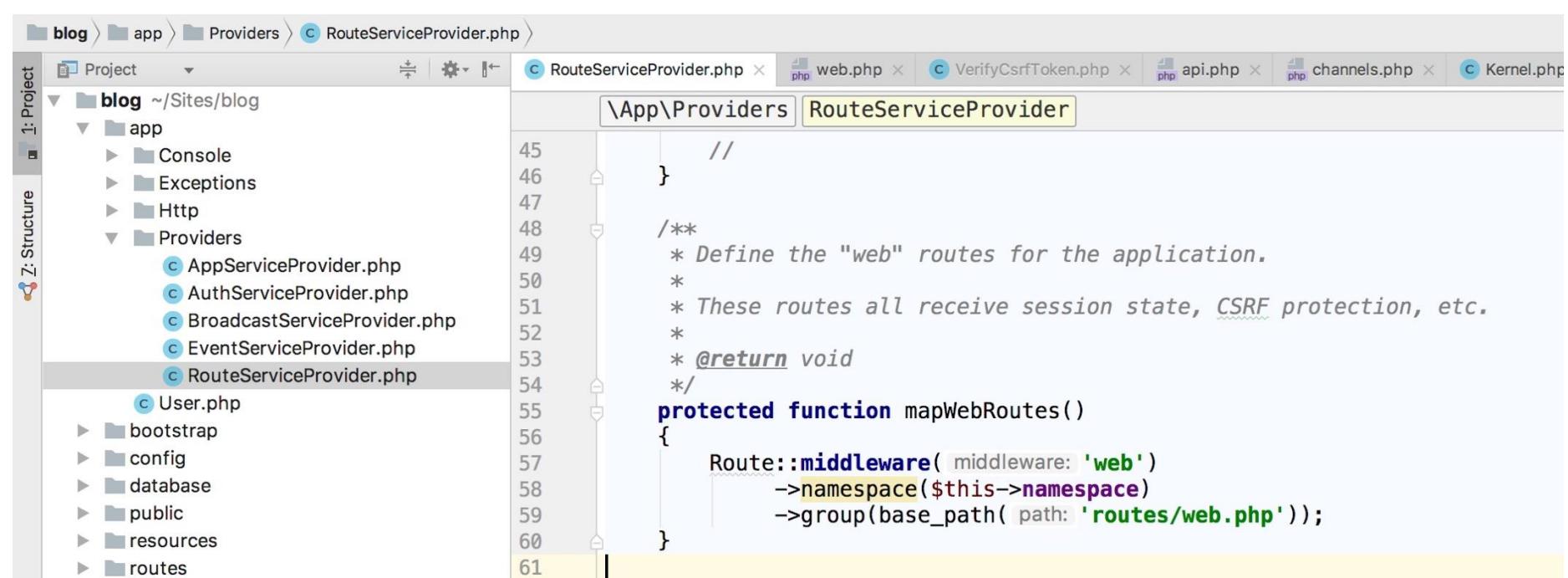
```
User {#231 ▾
  #fillable: array:3 [▶]
  #hidden: array:2 [▶]
  #connection: "mysql"
  #table: null
  #primaryKey: "id"
  #keyType: "int"
  +incrementing: true
  #with: []
  #withCount: []
  #perPage: 15
  +exists: true
  +wasRecentlyCreated: false
  #attributes: array:7 [▶]
  #original: array:7 [▶]
  #changes: []
  #casts: []
  #dates: []
  #dateFormat: null
  #appends: []
  #dispatchesEvents: []
  #observables: []
  #relations: []
  #touches: []
  +timestamps: true
  #visible: []
  #guarded: array:1 [▶]
  #rememberTokenName: "remember_token"
}
```

注: 控制器并不是一定要继承自基类, 不过, 那样的话就不能使用一些基类提供的便利方法了, 比如 `middleware`、`validate` 和 `dispatch` 等, 后面我们慢慢接触和了解到这些方法的使用。

命名空间

你应该注意到我们在定义控制器路由的时候没有指定完整的控制器命名空间, 而只是定义了 `App\Http\Controllers` 之后的部分, 那为什么可以这么做呢? 这是因为默认情况下, `RouteServiceProvider` 将会在一个指定了控制器所在命名空间的路由分组中载入路由文件, 故而我们只需指定后面相对命名空间即可:

[obj]



```

45      // 
46    }
47
48  /**
49   * Define the "web" routes for the application.
50   *
51   * These routes all receive session state, CSRF protection, etc.
52   *
53   * @return void
54  */
55  protected function mapWebRoutes()
56  {
57    Route::middleware('web')
58      ->namespace($this->namespace)
59      ->group(base_path('routes/web.php'));
60
61

```

这里的 `$this->namespace` 就是 `App\Http\Controllers`。

如果你在 `App\Http\Controllers` 目录下选择使用 PHP 命名空间嵌套或组织控制器，只需要使用相对于 `App\Http\Controllers` 命名空间的指定类名即可。因此，如果你的完整控制器类是 `App\Http\Controllers\Photos\AdminController`，则可以像这样注册路由：

```
Route::get('foo', 'Photos\AdminController@method');
```

单动作控制器

如果你想要定义一个只处理一个动作的控制器，可以在这个控制器中定义 `__invoke` 方法：

```

<?php

namespace App\Http\Controllers;

use App\User;
use App\Http\Controllers\Controller;

class ShowProfile extends Controller
{
    /**
     * 展示给定用户的个人主页
     *
     * @param int $id
     * @return Response
     */
    public function __invoke($id)
    {
        return view('user.profile', ['user' => User::findOrFail($id)]);
    }
}

```

当你为这个单动作控制器注册路由的时候，不需要指定方法：

```
Route::get('user/{id}', 'ShowProfile');
```

这背后的原理是在 PHP 中当尝试以调用函数的方式调用一个对象时，`__invoke()` 方法会被自动调用。

控制器中间件

中间件可以像这样分配给控制器路由：

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

不过，将中间件放在控制器构造函数中更方便，在控制器的构造函数中使用 `middleware` 方法你可以很轻松地分配中间件给该控制器（该方法继承自控制器基类），这样该中间件对所有控制器方法都生效：

```

<?php

namespace App\Http\Controllers;

use App\User;
use Illuminate\Http\Request;

class UserController extends Controller
{
    public function __construct()
    {

```

```

    $this->middleware('token');

}

/**
 * @param $id
 * @return \Illuminate\Contracts\View\Factory|\Illuminate\View\View
 * @author LaravelAcademy.org
 */
public function show($id)
{
    return view('user.profile', ['user' => User::findOrFail($id)]);
}
}

```

这里我们在构造函数中声明使用 `token` 中间件（关于该中间件定义参考[中间件](#)这篇文档），这样当我们访问 `http://blog.test/user/1` 的时候，就会跳转到 Laravel 学院，只有当访问 `http://blog.test/user/1?token=laravelacademy.org` 时，才能访问到正确的页面。除此之外，我们还可以指定中间件对指定方法生效或者排除指定方法的校验：

```

$this->middleware('auth')->only('show'); // 只对该方法生效
$this->middleware('auth')->except('show'); // 对该方法以外的方法生效

```

如果要指定多个控制器方法可以以数组的方式传参：

```

$this->middleware('auth')->only(['show', 'index']); // 只对指定方法生效
$this->middleware('auth')->except(['show', 'index']); // 对指定方法以外的方法生效

```

在控制器中还可以使用闭包注册中间件，这为我们定义只在某个控制器使用的中间件提供了方便，无需定义完整的中间件类：

```

$this->middleware(function ($request, $next) {
    // ...
    return $next($request);
});

```

还是以 `UserController` 为例，我们为其定义一个匿名中间件：

```

class UserController extends Controller
{
    public function __construct()
    {
        $this->middleware(middleware: 'token')->except(methods: 'show');
        $this->middleware(function ($request, $next) {
            if (!is_numeric($request->input('id'))) {
                throw new NotFoundHttpException();
            }
            return $next($request);
        });
    }
}

```

这样当我们访问 `http://blog.test/user/1` 会抛出 404 异常，只有当访问 `http://blog.test/user/1?id=1` 时才能正常展示。

注：你还可以将中间件分配给多个控制器动作，不过，这意味着你的控制器会变得越来越臃肿，这种情况下，需要考虑将控制器分割成多个更小的控制器。

资源控制器

Laravel 的资源控制器可以让我们很便捷地构建基于资源的 RESTful 控制器，例如，你可能想要在应用中创建一个控制器，用于处理关于文章存储的 HTTP 请求，使用 Artisan 命令 `make:controller`，我们可以快速创建这样的控制器：

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

该 Artisan 命令将会生成一个控制器文件 `app/Http/Controllers/PostController.php`，这个控制器包含了每一个资源操作对应的方法：

```

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class PostController extends Controller
{
    /**
     * Display a listing of the resource.

```

```
* @return \Illuminate\Http\Response
*/
public function index()
{
    //
}

/**
 * Show the form for creating a new resource.
 *
 * @return \Illuminate\Http\Response
*/
public function create()
{
    //
}

/**
 * Store a newly created resource in storage.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
*/
public function store(Request $request)
{
    //
}

/**
 * Display the specified resource.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
*/
public function show($id)
{
    //
}

/**
 * Show the form for editing the specified resource.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
*/
public function edit($id)
{
    //
}

/**
 * Update the specified resource in storage.
 *
 * @param \Illuminate\Http\Request $request
 * @param int $id
 * @return \Illuminate\Http\Response
*/
public function update(Request $request, $id)
{
    //
}

/**
 * Remove the specified resource from storage.
 *
 * @param int $id
 * @return \Illuminate\Http\Response
*/
public function destroy($id)
{
    //
}
```

```
}
```

接下来，可以通过 `resource` 方法为该控制器注册一个资源路由：

```
Route::resource('posts', 'PostController');
```

这个路由声明包含了处理文章资源对应动作的多个路由，相应地，Artisan 生成的控制器也已经为这些动作设置了对应的处理方法。

你可以通过传递数组到 `resources` 方法从而一次注册多个资源控制器：

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

资源控制器处理的动作

请求方式	URI 路径	控制器方法	路由名称
GET	/posts	index	posts.index
GET	/posts/create	create	posts.create
POST	/posts	store	posts.store
GET	/posts/{post}	show	posts.show
GET	/posts/{post}/edit	edit	posts.edit
PUT/PATCH	/posts/{post}	update	posts.update
DELETE	/posts/{post}	destroy	posts.destroy

指定资源模型

如果你使用了路由模型绑定，并且想要在资源控制器的方法中对模型实例进行依赖注入，可以在生成控制器的使用使用 `--model` 选项：

```
php artisan make:controller PostController --resource --model=Post
```

不过学院君个人不推荐使用这种模型绑定，因为这里会涉及到对模型数据的缓存逻辑，为性能考虑，我们不想总是从数据库取数据，所以，尽量保持单个功能的简单和单一职责，让开发者自己去组装需要的功能，这是 Unix 奉行的设计哲学，也是我们在系统设计的时候需要考量的重要因素。

伪造表单方法

由于 HTML 表单不支持发起 `PUT`、`PATCH` 和 `DELETE` 请求，需要添加一个隐藏的 `_method` 字段来伪造 HTTP 请求方式，Blade 指令 `@method` 可以帮我们做这件事：

```
<form action="/foo/bar" method="POST">
    @method('PUT')
</form>
```

部分资源路由

声明资源路由时可以指定该路由处理的动作子集：

```
Route::resource('post', 'PostController', ['only' =>
    ['index', 'show']
]);

Route::resource('post', 'PostController', ['except' =>
    ['create', 'store', 'update', 'destroy']
]);
```

API 资源路由

声明被 API 消费的资源路由时，你可能需要排除展示 HTML 模板的路由，如 `create` 和 `edit`，为了方便起见，Laravel 提供了 `apiResource` 方法自动排除这两个路由：

```
Route::apiResource('post', 'PostController');
```

同样，你可以传递数组到 `apiResources` 方法从而一次注册多个 API 资源控制器：

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

要想快速生成不包含 `create` 或 `edit` 方法的 API 资源控制器，可以在执行 `make:controller` 命令时使用 `--api` 开关：

```
php artisan make:controller API/PostController --api
```

命名资源路由

默认情况下，所有资源控制器动作都有一个路由名称，不过，我们可以通过传入 `names` 数组来覆盖这些默认的名称：

```
Route::resource('post', 'PostController', ['names' =>
    ['create' => 'post.build']
]);
```

命名资源路由参数

默认情况下，`Route::resource` 将会基于资源名称的单数格式为资源路由创建路由参数，你可以通过在选项数组中传递 `parameters` 来覆盖这一默认设置。`parameters` 是资源名称和参数名称的关联数组：

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

上面的示例代码会为资源的 `show` 路由生成如下 URL：

```
/user/{admin_user}
```

本地化资源 URI

默认情况下，`Route::resource` 创建的资源 URI 是英文风格的，如果你需要本地化 `create` 和 `edit` 请求路由，可以使用 `Route::resourceVerbs` 方法。该功能可以在 `AppServiceProvider` 的 `boot` 方法中实现：

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

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Route::resourceVerbs([
        'create' => 'xinzeng',
        'edit' => 'bianji',
    ]);
}
```

定制化请求方式完成后，注册资源路由如 `Route::resource('wenzhang', 'PostController')` 将会生成如下 URI：

```
/wenzhang/xinzeng
/wenzhang/{wenzhang}/bianji
```

好吧，你可以看出来，我是用拼音的方式对资源路由进行了本地化设置。

补充资源控制器

如果需要在默认资源路由之外添加额外的路由到资源控制器，应该在调用 `Route::resource` 之前定义这些路由，否则，通过 `resource` 方法定义的路由可能无意中覆盖掉补充的额外路由：

```
Route::get('posts/popular', 'PostController@method');
Route::resource('posts', 'PostController');
```

注：注意保持控制器的单一职责，如果你发现指向控制器动作的路由超过默认提供的资源控制器动作集合了，考虑将你的控制器分割成多个更小的控制器。

依赖注入

构造函数注入

Laravel 使用 [服务容器](#) 解析所有的 Laravel 控制器，因此，可以在控制器的构造函数中注入任何依赖，这些依赖会被自动解析并注入到控制器实例中：

```
<?php

namespace App\Http\Controllers;

use App\Repositories\UserRepository;

class UserController extends Controller
{
    /**
     * The user repository instance.
     */
}
```

```

protected $users;

/**
 * 创建新的控制器实例
 *
 * @param UserRepository $users
 * @return void
 */
public function __construct(UserRepository $users)
{
    $this->users = $users;
}
}

```

当然，你还可以注入任何 [Laravel 契约](#)，如果容器可以解析，就可以进行依赖注入。注入依赖到控制器可以让应用更加易于测试，同时也更加方便使用。

方法注入

除了构造函数注入之外，还可以在控制器的动作方法中进行依赖注入，例如，我们可以在某个方法中注入 [Illuminate\Http\Request](#) 实例：

```

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * 存储新用户
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        $name = $request->name;

        //
    }
}

```

如果控制器方法期望输入路由参数，只需要将路由参数放到其他依赖之后，例如，如果你的路由定义如下：

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

则需要以下方式定义控制器方法来注入 [Illuminate\Http\Request](#) 依赖并访问路由参数 `id`：

```

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * 更新指定用户
     *
     * @param Request $request
     * @param int $id
     * @return Response
     * @translator http://laravelacademy.org
     */
    public function update(Request $request, $id)
    {
        //
    }
}

```

关于服务容器和依赖注入，我们后面在服务容器部分会详细阐述，这里我们只需要了解可以这么使用就可以了。

路由缓存

注：路由缓存不会作用于基于闭包的路由。要使用路由缓存，必须将闭包路由转化为控制器路由。

如果你的应用完全基于控制器路由，可以使用 Laravel 的路由缓存，使用路由缓存将会极大降低注册所有应用路由所花费的时间开销，在某些案例中，路由注册速度甚至能提高 100 倍！想要生成路由缓存，只需执行 Artisan 命令 `route:cache`：

```
php artisan route:cache
```

运行完成后，每次请求都会从缓存中读取路由，所以如果你添加了新的路由需要重新生成路由缓存。因此，只有在项目部署阶段才需要运行 `route:cache` 命令，本地开发环境完全无此必要。

想要移除缓存路由文件，使用 `route:clear` 命令即可：

```
php artisan route:clear
```

HTTP 请求

访问请求实例

在控制器中获取当前 HTTP 请求实例，需要在构造函数或方法中对 `Illuminate\Http\Request` 类进行依赖注入，这样当前请求实例会被服务容器自动注入：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * 存储新用户
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        $name = $request->input('name');

        //
    }
}
```

依赖注入 & 路由参数

如果还期望在控制器方法中获取路由参数，只需要将路由参数置于其它依赖之后即可，例如，如果你的路由定义如下：

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

你仍然可以对 `Illuminate\Http\Request` 进行依赖注入并通过如下方式定义控制器方法来访问路由参数 `id`：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * 更新指定用户
     *
     * @param Request $request
     * @param int $id
     * @return Response
     */
    public function update(Request $request, $id)
    {
        //
    }
}
```

通过路由闭包访问请求

还可以在路由闭包中注入 `Illuminate\Http\Request`, 在执行闭包函数的时候服务容器会自动注入输入请求:

```
use Illuminate\Http\Request;

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

请求路径 & 方法

`Illuminate\Http\Request` 继承自 `Symfony\Component\HttpFoundation\Request` 类, 提供了多个方法来检测应用的 HTTP 请求, 下面我们来演示其提供的一些获取请求路径和请求方式的方法:

获取请求路径

`path` 方法将会返回请求的路径信息, 因此, 如果请求 URL 是 `http://domain.com/user/1`, 则 `path` 方法将会返回 `user/1`:

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

`is` 方法允许你验证请求路径是否与给定模式匹配。该方法参数支持 `*` 通配符:

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

如果请求 URL 是 `http://domain.com/user/1`, 该方法会返回 `true`。

获取请求 URL

想要获取完整的 URL, 而不仅仅是路径信息, 可以使用请求实例提供的 `url` 或 `fullUrl` 方法, `url` 方法返回不带查询字符串的 URL, 而 `fullUrl` 方法返回结果则包含查询字符串:

```
// 不包含查询字符串
$url = $request->url();

// 包含查询字符串
$url_with_query = $request->fullUrl();
```

例如, 我们请求 `http://domain.com/user/1?token=laravelacademy.org`, 则上述 `$url` 的值是 `http://domain.com/user/1`, `$url_with_query` 的值是 `http://blog.test/user/1?token=laravelacademy.org`。

获取请求方法

`method` 方法将会返回 HTTP 请求方式。你还可以使用 `isMethod` 方法来验证 HTTP 请求方式是否匹配给定字符串:

```
$method = $request->method(); // GET/POST

if($request->isMethod('post')) {
    // true or false
}
```

PSR-7 请求

PSR-7 标准 指定了 HTTP 消息接口, 包括请求和响应。如果你想要获取遵循 PSR-7 标准的请求实例而不是 Laravel 请求实例, 首先需要安装一些库。Laravel 可以使用 *Symfony HTTP Message Bridge* 组件将典型的 Laravel 请求和响应转化为兼容 PSR-7 接口的实现:

```
composer require symfony/psr-http-message-bridge
composer require zendframework/zend-diactoros
```

安装完这些库之后, 只需要在路由或控制器中通过对请求示例进行类型提示就可以获取 PSR-7 请求:

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

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

对比下 `Request` 实例和 `ServerRequestInterface` 实例的数据结构, 可以看到 Laravel 的 `Request` 实例提供信息更丰富:

Request {#43 ▼}

```

#json: null
#convertedFiles: null
#userResolver: Closure {#168 ▶}
#routeResolver: Closure {#167 ▶}
+attributes: ParameterBag {#45 ▶}
+request: ParameterBag {#51 ▶}
+query: ParameterBag {#51 ▼
    #parameters: array:1 [▼
        "token" => "laravelacademy.org"
    ]
}
+server: ServerBag {#47 ▶}
+files: FileBag {#48 ▶}
+cookies: ParameterBag {#46 ▶}
+headers: HeaderBag {#49 ▶}
#content: null
#languages: null
#Charsets: null
#encodings: null
#acceptableContentTypes: null
#pathInfo: "/user/1"
#requestUri: "/user/1?token=laravelacademy.org"
#baseUrl: ""
#basePath: null
#method: "GET"
#format: null
#session: Store {#213 ▶}
#locale: null
#defaultLocale: "en"
-isHostValid: true
-isClientIpsValid: true
-isForwardedValid: true
 basePath: ""
 format: "html"
}

```

ServerRequest {#234 ▼}

```

-attributes: []
-cookieParams: []
-parsedBody: array:1 [▶]
-queryParams: array:1 [▼
    "token" => "laravelacademy.org"
]
-serverParams: array:32 [▶]
-uploadedFiles: []
-method: "GET"
-requestTarget: null
-uri: Uri {#233 ▶}
#headers: array:10 [▶]
#headerNames: array:10 [▶]
-protocol: "1.1"
-stream: Stream {#231 ▶}
}

```

注：如果从路由或控制器返回的是 PSR-7 响应实例，则其将会自动转化为 Laravel 响应实例并显示出来。

请求字符串处理

默认情况下，Laravel 在 `App\Http\Kernel` 的全局中间件堆栈中引入了 `TrimStrings` 和 `ConvertEmptyStringsToNull` 中间件。这些中间件会自动对请求中的字符串字段进行处理，前者将字符串两端的空格清除，后者将空字符串转化为 `null`。这样，在路由和控制器中我们就不必对字符串字段做额外的处理：



```
\App\Http\Kernel

use App\Http\Middleware\CheckToken;
use Illuminate\Foundation\Http\Kernel as HttpKernel;

class Kernel extends HttpKernel
{
    /**
     * The application's global HTTP middleware stack.
     *
     * These middleware are run during every request to your application.
     *
     * @var array
     */
    protected $middleware = [
        \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
        \Illuminate\Foundation\Http\Middleware\ValidatePostSize::class,
        \App\Http\Middleware\TrimStrings::class,
        \Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::class,
        \App\Http\Middleware\TrustProxies::class,
    ];
}
```

如果你想要禁止该行为，可以从 `App\Http\Kernel` 的中间件堆栈属性 `$middleware` 中移除这两个中间件。

获取请求输入

获取所有输入值

你可以使用 `a11` 方法以数组格式获取所有输入值：

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

如果请求 URL 是 `http://blog.test/user/1?token=laravelacademy.org&name=学院君`，则对应 `$input` 返回值是：

```
array:2 [▼  
    "token" => "laravelacademy.org"  
    "name" => "学院君"  
]
```

获取单个输入值

通过一些很简单的方法，就可以从 `Illuminate\Http\Request` 实例中访问用户输入。你不需要关心请求所使用的 HTTP 请求方式，因为对所有请求方式都是通过 `input` 方法获取用户输入：

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

还是以上面的请求 URL 为例，这里的 \$name 值是 学院君。

你还可以传递一个默认值作为第二个参数给 `input` 方法，如果请求输入值在当前请求 URL 中未出现时该值将会被返回：

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

比如我们访问 `http://blog.test/user/1?token=laravelacademy.org`，仍然可以获取到 `$name` 的值为 `学院君`。

处理表单数组输入时，可以使用“.”来访问数组输入：

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

比如我们访问 `http://blog.test/user/1?products[] [name]=学院君&products[] [name]=学院君小号`，则上述 `$input` 的值是 `学院君`，而 `$names` 的值是：

```
array:2 [▼  
 0 => "学院君"  
 1 => "学院君小号"  
 ]
```

从查询字符串中获取输入

`input` 方法会从整个请求负载（包括查询字符串）中获取数值，`query` 则只会从查询字符串中获取数值：

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

如果请求的查询字符串中没有提供给定的查询项，我们可以像 `input` 方法一样设置第二个参数为默认值：

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

你也可以调用一个不传任何参数的 `query` 方法以便以关联数组的方式获取整个查询字符串的值，类似 `all` 方法所做的：

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

写到这里，估计有些人要蒙圈了，那 `input` 和 `query` 到底有什么区别，官方文档还是有些含混不清，那么这里学院君一杆子打到底，跟你聊聊两者本质区别，回到上面打印 Request 实例那张图：

Request {#43 ▼}

```
#json: null
#convertedFiles: null
#userResolver: Closure {#168 ▶}
#routeResolver: Closure {#167 ▶}
+attributes: ParameterBag {#45 ▶}
+request: ParameterBag {#51 ▶}
+query: ParameterBag {#51 ▶} +query: ParameterBag {#51 ▶}
+server: ServerBag {#47 ▶}
+files: FileBag {#48 ▶}
+cookies: ParameterBag {#46 ▶}
+headers: HeaderBag {#49 ▶}
#content: null
#languages: null
#Charsets: null
#encodings: null
#acceptableContentTypes: null
#pathInfo: "/user/1"
#requestUri: "/user/1"
#baseUrl: ""
#basePath: null
#method: "GET"
#format: null
#session: Store {#213 ▶}
#locale: null
#defaultLocale: "en"
-isHostValid: true
-isClientIpsValid: true
-isForwardedValid: true
 basePath: ""
 format: "html"
}
```

注意到标红圈的部分，`query` 方法就是从 `query` 属性对象中获取参数值，`input` 方法会从 `query + request` 属性对象中获取参数值，请求实例上还有个 `post` 方法用于从 `request` 属性对象中获取参数值，讲到这里我们应该有点眉目了，`query` 方法用于获取 GET 请求查询字符串参数值，`input` 方法用于获取所有 HTTP 请求参数值，`post` 方法用于获取 POST 请求参数值。感兴趣的同学可以去底层查看下源码：

`vendor/laravel/framework/src/Illuminate/Http/Concerns/InteractsWithInput.php`。

通过动态属性获取输入

此外，你还可以通过使用 `Illuminate\Http\Request` 实例上的动态属性来访问用户输入。例如，如果你的应用表单包含 `name` 字段，那么可以像这样访问提交的值：

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

使用动态属性的时候，Laravel 首先会在请求中查找参数的值，如果不存在，还会到路由参数中查找。该功能的实现原理自然是魔术函数 `__get` 了：

```

/**
 * Get an input element from the request.
 *
 * @param string $key
 * @return mixed
 */
public function __get($key)
{
    if (array_key_exists($key, $this->all())) {
        return data_get($this->all(), $key);
    }

    return $this->route($key);
}

```

获取 JSON 输入值

发送 JSON 请求到应用的时候，只要 `Content-Type` 请求头被设置为 `application/json`，都可以通过 `input` 方法获取 JSON 数据，还可以通过“.”号解析数组：

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

获取输入的部分数据

如果你需要取出输入数据的子集，可以使用 `only` 或 `except` 方法，这两个方法都接收一个数组或动态列表作为唯一参数，这和我们在上一篇[控制器](#)中提到的控制器中间件使用语法类似：

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

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

注：`only` 方法返回所有你想要获取的参数键值对，不过，如果你想要获取的参数不存在，则对应参数会被过滤掉。

判断请求参数是否存在

判断参数在请求中是否存在，可以使用 `has` 方法，如果参数存在则返回 `true`：

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

该方法支持以数组形式查询多个参数，这种情况下，只有当参数都存在时，才会返回 `true`：

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

如果你想要判断参数存在且参数值不为空，可以使用 `filled` 方法：

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

上一次请求输入

Laravel 允许你在两次请求之间保存上一次输入数据，这个特性在检测校验数据失败后需要重新填充表单数据时很有用，不过如果你使用的是 Laravel 自带的校验功能，则不需要手动使用这些方法，因为一些 Laravel 自带的校验设置会自动调用它们。

将输入存储到 Session

`Illuminate\Http\Request` 实例的 `flash` 方法会将当前输入放到一次性 Session（所谓的一次性指的是从 Session 中取出数据后，对应数据会从 Session 中销毁）中，这样在下一次请求时上一次输入的数据依然有效：

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

你还可以使用 `flashOnly` 和 `flashExcept` 方法将输入数据子集放到 Session 中，这些方法在 Session 之外保存敏感信息时很有用，该功能适用于登录密码填写错误的场景：

```
$request->flashOnly('username', 'email');
$request->flashExcept('password');
```

将输入存储到 Session 然后重定向

如果你经常需要一次性存储输入请求输入并返回到表单填写页，可以在 `redirect` 之后调用 `withInput` 方法实现这样的功能：

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

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

取出上次请求数据

要从 Session 中取出上次请求的输入数据，可以使用 Request 实例提供的 `old` 方法。`old` 方法可以很方便地从 Session 中取出一次性数据：

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

Laravel 还提供了一个全局的辅助函数 `old()`，如果你是在 Blade 模板中显示上次输入数据，使用辅助函数 `old()` 更方便，如果给定参数没有对应输入，返回 `null`：

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

Cookie

从请求中取出 Cookie

为了安全起见，Laravel 框架创建的所有 Cookie 都经过加密并使用一个验证码进行签名，这意味着如果客户端修改了它们则需要对其进行有效性验证。我们使用 `Illuminate\Http\Request` 实例的 `cookie` 方法从请求中获取 Cookie 的值：

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

此外，还可以使用 `Cookie` 门面获取 Cookie 值：

```
$value = Cookie::get('name');
```

添加 Cookie 到响应

你可以使用 `cookie` 方法将一个 Cookie 添加到返回的 `Illuminate\Http\Response` 实例，你需要传递 Cookie 名称、值、以及有效期（分钟）到这个方法：

```
return response('欢迎来到 Laravel 学院')->cookie(
    'name', '学院君', $minutes
);
```

`cookie` 方法可以接收一些使用频率较低的参数，一般来说，这些参数和 PHP 原生函数 `setcookie` 作用和意义一致：

```
return response('欢迎来到 Laravel 学院')->cookie(
    'name', '学院君', $minutes, $path, $domain, $secure, $httpOnly
);
```

我们简单演示下该功能的使用，在 `routes/web.php` 定义两个路由如下：

```
Route::get('cookie/add', function () {
    $minutes = 24 * 60;

    return response('欢迎来到 Laravel 学院')->cookie('name', '学院君', $minutes);
});

Route::get('cookie/get', function (\Illuminate\Http\Request $request) {
    $cookie = $request->cookie('name');
    dd($cookie);
});
```

先访问 `http://blog.test/cookie/add` 设置 Cookie，然后再通过 `http://blog.test/cookie/get` 获取 Cookie 值，如果在页面看到输出 `学院君`，则表示 Cookie 设置成功。当然我们也可以通过 Chrome 浏览器的 F12 模式快速查看 Cookie 信息：

"学院君"

Name	Value	Domain	Path	Expires / Max-Age	Size	HTTP	Secure	SameSite
XSRF-TOKEN	eyJpdii6IkIBbkhvZ0VjUE9mQmNVZG9wTDRhTkE9PSIsInZh...	blog.dev	/	2017-09-26T07:...	288			
laravel_session	eyJpdii6IIF2U2FcLzhKMzNJWmhGZnRMZUVzV0N3PT0ILCJ...	blog.dev	/	2017-09-26T07:...	291	✓		
name	eyJpdii6IjZNOWpCNWJ0bVY0eGRBbHdBanRYK3c9PSIsInZ...	blog.dev	/	2017-09-27T05:...	220	✓		

被加密过的 `name` 就是我们刚刚设置的 Cookie 了。

此外，你还可以使用 `Cookie` 门面将应用于附件的 Cookie 推送到输出响应队列。`queue` 方法接收一个 `Cookie` 实例或者创建一个 `Cookie` 实例的必要参数。这些 Cookie 会在响应发送到浏览器之前添加上：

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

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

生成 Cookie 实例

如果你想要生成一个 `Symfony\Component\HttpFoundation\Cookie` 实例以便后续添加到响应实例，可以使用全局辅助函数 `cookie`，该 `Cookie` 只有在添加到响应实例上才会发送到客户端：

```
$cookie = cookie('name', '学院君', $minutes);

return response('欢迎来到 Laravel 学院')->cookie($cookie);
```

我们改写下之前的 `cookie/add` 路由实现逻辑：

```
Route::get('cookie/add', function () {
    $minutes = 24 * 60;

    //return response('欢迎来到 Laravel 学院')->cookie('name', '学院君', $minutes);

    $cookie = cookie('name', '学院君 X', $minutes);

    return response('欢迎来到 Laravel 学院')->cookie($cookie);
});
```

效果和之前一致，这次访问 `cookie/get` 路由，页面打印结果是 `学院君 X`。

文件上传

获取上传的文件

可以使用 `Illuminate\Http\Request` 实例提供的 `file` 方法或者动态属性来访问上传文件，`file` 方法返回 `Illuminate\Http\UploadedFile` 类的一个实例，该类继承自 PHP 标准库中提供与文件交互方法的 `SplFileInfo` 类：

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

你可以使用 `hasFile` 方法判断文件在请求中是否存在：

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

验证文件是否上传成功

使用 `isValid` 方法判断文件在上传过程中是否出错：

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

文件路径 & 扩展名

`UploadedFile` 类还提供了访问上传文件绝对路径和扩展名的方法。`extension` 方法可以基于文件内容判断文件扩展名，该扩展名可能会和客户端提供的扩展名不一致：

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

其他文件方法

`UploadedFile` 实例上还有很多其他可用方法，查看[该类的 API 文档](#)了解更多信息。

保存上传的文件

要保存上传的文件，需要使用你所配置的某个[文件系统](#)，对应配置位于 `config/filesystems.php`：

```

'disks' => [
    'local' => [
        'driver' => 'local',
        'root' => storage_path( path: 'app' ),
    ],
    'public' => [
        'driver' => 'local',
        'root' => storage_path( path: 'app/public' ),
        'url' => env( key: 'APP_URL' ).'/storage',
        'visibility' => 'public',
    ],
    's3' => [
        'driver' => 's3',
        'key' => env( key: 'AWS_KEY' ),
        'secret' => env( key: 'AWS_SECRET' ),
        'region' => env( key: 'AWS_REGION' ),
        'bucket' => env( key: 'AWS_BUCKET' ),
    ],
],
]

```

[obj]

Laravel 默认使用 `local` 配置存放上传文件，即本地文件系统，默认根目录是 `storage/app`，`public` 也是本地文件系统，只不过存放在这里的文件可以被公开访问，其对应的根目录是 `storage/app/public`，要让 Web 用户访问到该目录下存放文件的前提是在应用入口 `public` 目录下建一个软链接 `storage` 链接到 `storage/app/public`。

`UploadedFile` 类有一个 `store` 方法，该方法会将上传文件移动到相应的磁盘路径上，该路径可以是本地文件系统的某个位置，也可以是云存储（如 Amazon S3）上的路径。

`store` 方法接收一个文件保存的相对路径（相对于文件系统配置的根目录），该路径不需要包含文件名，因为系统会自动生成一个唯一 ID 作为文件名。

`store` 方法还接收一个可选的参数 —— 用于存储文件的磁盘名称作为第二个参数（对应文件系统配置 `disks` 的键名，默认值是 `local`），该方法会返回相对于根目录的文件路径：

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

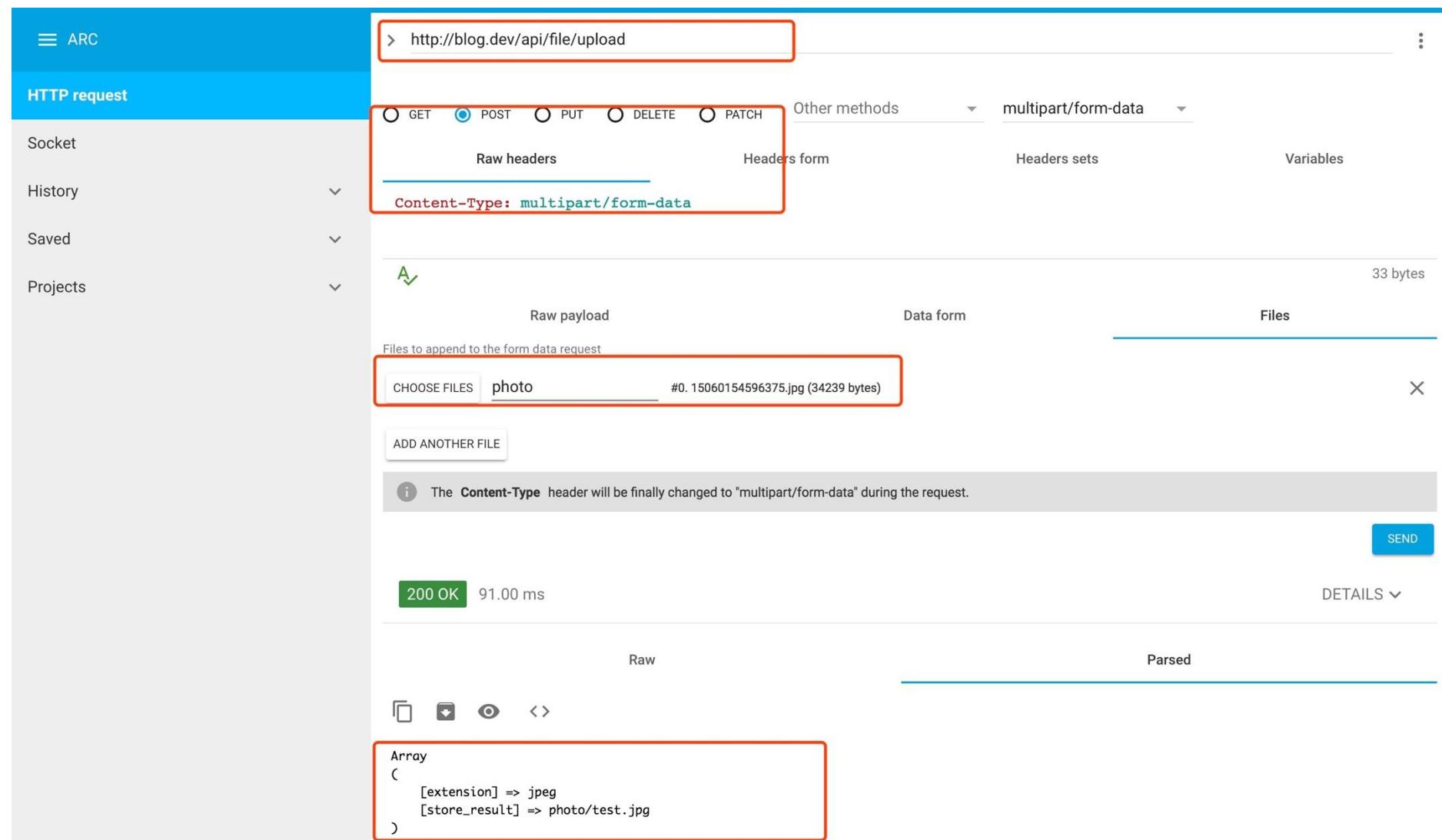
如果你不想自动生成文件名，可以使用 `storeAs` 方法，该方法接收保存路径、文件名和磁盘名作为参数：

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

下面我们来简单演示下文件上传功能，在 `routes/api.php` 中定义如下文件上传路由：

```
Route::post('file/upload', function(\Illuminate\Http\Request $request) {
    if ($request->hasFile('photo') && $request->file('photo')->isValid()) {
        $photo = $request->file('photo');
        $extension = $photo->extension();
        // $store_result = $photo->store('photo');
        $store_result = $photo->storeAs('photo', 'test.jpg');
        $output = [
            'extension' => $extension,
            'store_result' => $store_result
        ];
        print_r($output); exit();
    }
    exit('未获取到上传文件或上传过程出错');
});
```

我们还是使用 Advanced REST Client 工具来演示 POST 表单提交：



标记红圈的地方是需要重点关注的输入和输出。我分别测试了 `store` 方法和 `storeAs` 方法，上传文件成功后可以去 `storage/app` 目录下查看：



其他存储介质使用方式也差不多，无非是修改下 `store` 和 `storeAs` 对应的参数。在使用过程中遇到什么问题，欢迎在评论区反馈。

配置信任代理

如果你的应用运行在一个会中断 TLS/SSL 证书的负载均衡器之后，你会注意到有的时候应用不会生成 HTTPS 链接，通常这是因为应用是从负载均衡器从 80 端口转发过来的流量，所以不知道应该生成安全加密链接。

要解决这个问题可以使用 `App\Http\Middleware\TrustProxies` 中间件，该中间件允许你快速自定义需要被应用信任的负载均衡器或代理。被信任的代理位于这个中间件的 `$proxies` 属性列表，除了配置信任代理之外，还可以配置代理发送的带有请求来源信息的消息头：

```
<?php

namespace App\Http\Middleware;

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

class TrustProxies extends Middleware
{
    /**
     * The trusted proxies for this application.
     *
     * @var array
     */
    protected $proxies = [
        '192.168.1.1',
        '192.168.1.2',
    ];

    /**
     * The headers that should be used to detect proxies.
     *
     * @var string
     */
}
```

```
 */
protected $headers = Request::HEADER_X_FORWARDED_ALL;
}
```

注：如果你在使用 AWS Elastic Load Balancing，`headers` 值应该修改为 `Request::HEADER_X_FORWARDED_AWS_ELB`，关于 `headers` 属性可用的更多常量，请查看 [Symfony 信任代理文档](#)。

信任所有代理

如果你在使用 Amazon AWS 或者其他云服务提供的负载均衡，并不知道均衡器真实的 IP 地址，这种情况下，可以使用 `*` 通配符信任所有代理：

```
/**
 * The trusted proxies for this application.
 *
 * @var array
 */
protected $proxies = '*';
```

HTTP 响应

创建响应

字符串 & 数组

所有路由和控制器处理完业务逻辑之后都会返回一个发送到用户浏览器的响应，Laravel 提供了多种不同的方式来返回响应，最基本的响应就是从路由或控制器返回一个简单的字符串，框架会自动将这个字符串转化为一个完整的 HTTP 响应：

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

除了从路由或控制器返回字符串之外，还可以返回数组。框架会自动将数组转化为一个 JSON 响应：

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

注：你知道还可以从路由或控制器返回 [Eloquent 集合](#) 吗？这也会被自动转化为 JSON 响应。

Response 对象

通常，我们并不只是从路由动作简单返回字符串和数组，大多数情况下，都会返回一个完整的 `Illuminate\Http\Response` 实例或 [视图](#)。

返回一个完整的 `Response` 实例允许你自定义响应的 HTTP 状态码和头信息。`Response` 实例继承

自 `Symfony\Component\HttpFoundation\Response` 基类，该类提供了一系列方法用于创建 HTTP 响应：

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

添加响应头

大部分响应方法都可以以方法链的形式调用，从而可以流式构建响应（[流接口模式](#)）。例如，在发送响应给用户前可以使用 `header` 方法来添加一系列响应头：

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

或者你可以使用 `withHeaders` 方法来指定头信息数组添加到响应：

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

添加 Cookie 到响应

使用响应实例上的 `cookie` 方法可以轻松添加 Cookie 到响应。例如，你可以使用 `cookie` 方法生成 Cookie 并添加将其添加到响应实例：

```
return response($content)
    ->header('Content-Type', $type)
    ->cookie('name', 'value', $minutes);
```

`cookie` 方法还可以接收更多使用频率较低的额外可选参数，一般来说，这些参数和 PHP 原生提供的 `setcookie` 方法目的和意义差不多：

```
->cookie($name, $value, $minutes, $path, $domain, $secure, $httpOnly)
```

此外，还可以使用 `Cookie` 门面将应用于附件的 Cookie 推送到输出响应队列。`queue` 方法接收 `Cookie` 实例或创建 `Cookie` 所必要的参数，这些 `Cookie` 会在响应被发送到浏览器之前添加到响应：

```
Route::get('cookie/response', function() {
    Cookie::queue(Cookie::make('site', 'Laravel 学院', 1));
    Cookie::queue('author', '学院君', 1);
    return response('Hello Laravel', 200)
        ->header('Content-Type', 'text/plain');
});
```

我们在浏览器中访问 <http://blog.test/cookie/response> 就能看到这两个新增的 Cookie:

Name	Headers	Preview	Response	Cookies	Timing
response					
topFrame.js					
Clipper.js					
ContentPreview.js					
Coordinator.js					
GlobalUtils.js					
Promotion.js					
CustomTooltipEligibility.js					
checkSimSearch.js					
pageVisible.js					
Request Cookies					
XSRF-TOKEN				eyJpdil6ljdPbzRyR3FOdXVxekYyRjg3VzdURkE9PSIsInZhbHVljo...	N/A
laravel_session				eyJpdil6lmlZcStuVXNmY2VrMFhzZ3hITVFRTwC9PSIsInZhbHVljo...	N/A
Response Cookies					
XSRF-TOKEN				eyJpdil6llFock1RMWl5cm1vRUh6dFNoNzd3elE9PSIsInZhbHVljo...	
author				eyJpdil6lmZVeFl2Nk9EUFWveDB5ck1KUiszOVVRPT0iLCJ2YWx1...	
laravel_session				eyJpdil6ljtSHdkQW5IS2VXMUFMZXBVnRngzZ3c9PSIsInZhbHVljo...	
site				eyJpdil6lmVSSkFMc20yNmdlU0RLcjVmczZ2cnc9PSIsInZhbHVljo...	

Cookie & 加密

默认情况下，Laravel 框架生成的 Cookie 都经过了加密和签名，以免在客户端被篡改。如果你想要让特定的 Cookie 子集在生成时取消加密，可以通过 `app/Http/Middleware` 目录下的中间件 `App\Http\Middleware\EncryptCookies` 提供的 `$except` 属性来排除这些 Cookie:

```
/*
 * 不需要被加密的 cookies 名称
 *
 * @var array
 */
protected $except = [
    'cookie_name',
];
```

重定向

重定向响应是 `Illuminate\Http\RedirectResponse` 类的实例，包含了必要的头信息将用户重定向到另一个 URL，有很多方式来生成 `RedirectResponse` 实例，最简单的方法就是使用全局辅助函数 `redirect`:

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

有时候你想要将用户重定向到上一个请求的位置，比如，表单提交后，验证不通过，你就可以使用辅助函数 `back` 返回到前一个 URL（由于该功能使用了 `Session`，使用该方法之前确保相关路由位于 `web` 中间件组或者应用了 `Session` 中间件）：

```
Route::post('user/profile', function () {
    // 验证请求...
    return back()->withInput();
});
```

重定向到命名路由

如果调用不带参数的 `redirect` 方法，会返回一个 `Illuminate\Routing\Redirector` 实例，然后就可以使用 `Redirector` 实例上的所有方法。例如，要生成一个 `RedirectResponse` 到命名路由，可以使用 `route` 方法:

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

如果路由中有参数，可以将其作为第二个参数传递到 `route` 方法:

```
// For a route with the following URI: profile/{id}
return redirect()->route('profile', ['id'=>1]);
```

通过 Eloquent 模型填充参数

如果要重定向到带 ID 参数的路由（Eloquent 模型绑定），可以传递模型本身，ID 会被自动解析出来:

```
return redirect()->route('profile', [$user]);
```

如果你想要自定义这个路由参数中的默认参数名（默认是 `id`），需要重写模型实例上的 `getRouteKey` 方法:

```
/**
 * Get the value of the model's route key.
 *
 * @return mixed
```

```
 */
public function getRouteKey()
{
    return $this->slug;
}
```

重定向到控制器动作

你还可以生成重定向到控制器动作的响应，只需传递控制器和动作名到 `action` 方法即可。记住，你不需要指定控制器的完整命名空间，因为 Laravel 的 `RouteServiceProvider` 将会自动设置默认的控制器命名空间：

```
return redirect()->action('HomeController@index');
```

和 `route` 方法一样，如果控制器路由要求参数，你可以将参数作为第二个参数传递给 `action` 方法：

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

重定向到外部域名

有时候你可能需要重定向到应用之外的其他域名，这可以通过调用 `away` 方法来实现，该方法会创建一个 `RedirectResponse`，没有任何额外的 URL 编码或验证：

```
return redirect()->away('http://laravelacademy.org');
```

带一次性 Session 数据的重定向

重定向到一个新的 URL 并将数据存储到一次性 Session 中通常是同时完成的，为了方便，可以创建一个 `RedirectResponse` 实例然后在同一个方法链上将数据存储到 Session，这种方式在 `action` 之后存储状态信息时特别方便：

```
Route::post('user/profile', function () {
    // 更新用户属性...
    return redirect('dashboard')->with('status', 'Profile updated!');
});
```

用户重定向到新页面之后，你可以从 `Session` 中取出并显示这些一次性信息，使用 Blade 语法实现如下：

```
@if (session('status'))
    <div class="alert alert-success">
        {{ session('status') }}
    </div>
@endif
```

注：这个一次性体现在从 `Session` 取出数据之后，这些数据就会被销毁，不复存在。

其它响应类型

上面我们讲了 `Response` 和 `RedirectResponse` 两种响应类型，我们还可以通过辅助函数 `response` 很方便地生成其他类型的响应实例，当无参数调用 `response` 时会返回 `Illuminate\Contracts\Routing\ResponseFactory` 契约的一个实现，该契约提供了一些有用的方法来生成各种响应，如视图相应、JSON 响应，文件下载、流响应等等。

视图响应

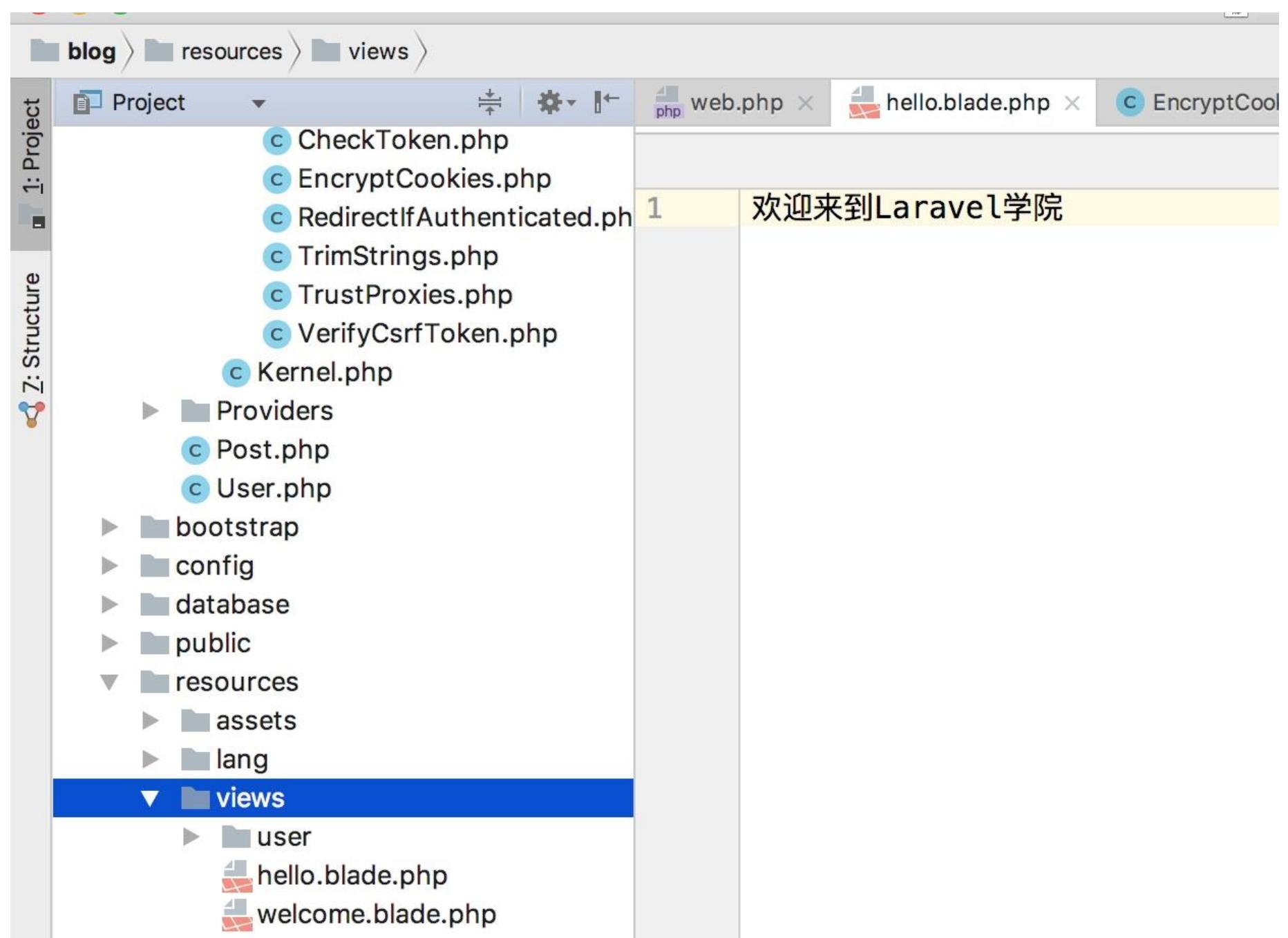
如果你需要控制响应状态和响应头，并且还需要返回一个视图作为响应内容，可以使用 `view` 方法：

```
return response()
    ->view('hello', $data, 200)
    ->header('Content-Type', $type);
```

当然，如果你不需要传递自定义的 HTTP 状态码和头信息，只需要简单使用全局辅助函数 `view` 即可：

```
Route::get('view/response', function() {
    return view('hello');
});
```

我们需要在这个路径下创建这个文件才能正常访问其内容：



JSON 响应

`json` 方法会自动将 `Content-Type` 头设置为 `application/json`, 并使用 PHP 函数 `json_encode` 方法将给定数组转化为 JSON 格式数据:

```
return response()->json([
    'name' => 'Abigail',
    'state' => 'CA'
]);
```

如果你想要创建一个 JSONP 响应, 可以在 `json` 方法之后调用 `withCallback` 方法:

```
return response()
    ->json(['name' => 'Abigail', 'state' => 'CA'])
    ->withCallback($request->input('callback'));
```

或者直接使用 `jsonp` 方法:

```
return response()
    ->jsonp($request->input('callback'), ['name' => 'Abigail', 'state' => 'CA']);
```

文件下载

`download` 方法用于生成强制用户浏览器下载给定路径文件的响应。`download` 方法接受文件名作为第二个参数, 该参数决定用户下载文件的显示名称, 你还可以将 HTTP 头信息作为第三个参数传递到该方法:

```
return response()->download($pathToFile);
return response()->download($pathToFile, $name, $headers);
return response()->download($pathToFile)->deleteFileAfterSend(true);
```

注: 管理文件下载的 Symfony HttpFoundation 类要求被下载文件有一个 ASCII 文件名, 这意味着被下载文件名不能是中文。

举个例子, 我们可以通过以下代码下载[上一篇文档](#)上传的图片:

```
Route::get('download/response', function() {
    return response()->download(storage_path('app/photo/test.jpg'), '测试图片.jpg');
});
```

流式下载

有时候你可能想要将给定操作的字符串响应转化为可下载的响应而不用将操作内容写入磁盘。这种场景下可以使用 `streamDownload` 方法, 该方法接收一个回调、文件名以及可选的响应头数组作为参数:

```
return response()->streamDownload(function () {
    echo GitHub::api('repo')
        ->contents()
```

```
->readme('laravel', 'laravel')['contents'];
}, 'laravel-readme.md');
```

文件响应

`file` 方法可用于直接在用户浏览器显示文件，例如图片或 PDF，而不需要下载，该方法接收文件路径作为第一个参数，头信息数组作为第二个参数：

```
return response()->file($pathToFile);
return response()->file($pathToFile, $headers);
```

响应宏

如果你想要定义一个自定义的可以在多个路由和控制器中复用的响应，可以使用 `Response` 门面上的 `macro` 方法。例如，在某个服务提供者的 `boot` 方法中编写代码如下：

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Response;
use Illuminate\Support\ServiceProvider;

class ResponseMacroServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Response::macro('caps', function ($value) {
            return Response::make(strtoupper($value));
        });
    }
}
```

`macro` 方法接收响应名称作为第一个参数，闭包函数作为第二个参数，响应宏的闭包在 `ResponseFactory` 实现类或辅助函数 `response` 中调用宏名称的时候被执行：

```
Route::get('macro/response', function() {
    return response()->caps('LaravelAcademy');
});
```

在浏览器中访问 `http://blog.test/macro/response`，输出如下：

```
LARAVELACADEMY
```

Blade 模板引擎

简介

Blade 是由 Laravel 提供的非常简单但功能强大的模板引擎，不同于其他流行的 PHP 模板引擎，Blade 在视图中并不约束你使用 PHP 原生代码。所有的 Blade 视图最终都会被编译成原生 PHP 代码并缓存起来直到被修改，这意味着对应用的性能而言 Blade 基本上是零开销。Blade 视图文件使用 `.blade.php` 文件扩展并存放在 `resources/views` 目录下。

模板继承

定义布局

使用 Blade 的两个最大优点是模板继承和片段组合，开始之前让我们先看一个例子。首先，我们测试“主”页面布局，由于大多数 Web 应用在不同页面中使用同一个布局，可以很方便的将这个布局定义为一个单独的 Blade 页面：

```
<!-- 存放在 resources/views/layouts/app.blade.php -->

<html>
    <head>
```

```
<title>应用名称 - @yield('title')</title>
</head>
<body>
    @section('sidebar')
        这里是侧边栏
    @show

    <div class="container">
        @yield('content')
    </div>
</body>
</html>
```

正如你所看到的，该文件包含典型的 HTML 标记，不过，注意 `@section` 和 `@yield` 指令，前者正如其名字所暗示的，定义了一个内容片段，而后者用于显示给定片段的内容。

现在我们已经为应用定义了一个布局，接下来让我们定义继承该布局的子页面吧。

继承布局

定义子页面的时候，可以使用 Blade 的 `@extends` 指令来指定子页面所继承的布局，继承一个 Blade 布局的视图可以使用 `@section` 指令注入内容到布局定义的内容片段中，记住，如上面例子所示，这些片段的内容将会显示在布局中使用 `@yield` 的地方：

```
<!-- 存放在 resources/views/child.blade.php -->

@extends('layouts.app')

@section('title', 'Laravel 学院')

@section('sidebar')
    @parent
    <p>Laravel 学院致力于提供优质 Laravel 中文学习资源</p>
@endsection

@section('content')
    <p>这里是主体内容，完善中...</p>
@endsection
```

在本例中，`sidebar` 片段使用 `@parent` 指令来追加（而非覆盖）内容到继承布局的侧边栏，`@parent` 指令在视图渲染时将会被布局中的内容替换。注：与之前的示例相反，`sidebar` 部分以 `@endsection` 结束而不是 `@show`，`@endsection` 指令只是定义一个 `section` 而 `@show` 指令定义并立即返回这个 `section`。

Blade 视图可以通过 `view` 方法直接从路由中返回：

```
Route::get('blade', function () {
    return view('child');
});
```

这样在浏览器中访问 <http://blog.test/blade>，就可以看到页面显示如下：

这里是侧边栏

Laravel学院致力于提供优质Laravel中文学习资源

这里是主体内容，完善中...

现在页面还很粗糙，没有任何样式，后面学习前端组件后可以回来完善。

组件 & 插槽

组件和插槽给内容片段（section）和布局（layout）带来了方便，不过，有些人可能会发现组件和插槽的模型更容易理解。首先，我们假设有一个可复用的“alert”组件，我们想要在整个应用中都可以复用它：

```
<!-- /resources/views/alert.blade.php -->

<div class="alert alert-danger">
    {{ $slot }}
</div>
```

`{{ $slot }}` 变量包含了我们想要注入组件的内容，现在，要构建这个组件，我们可以使用 Blade 指令 `@component`：

```
@component('alert')
    <strong>Whoops!</strong> Something went wrong!
@endcomponent
```

有时候为组件定义多个插槽很有用。下面我们来编辑 alert 组件以便可以注入“标题”，命名插槽可以通过“echoing”与它们的名字相匹配的变量来显示：

```
<!-- /resources/views/alert.blade.php -->

<div class="alert alert-danger">
    <div class="alert-title">{{ $title }}</div>
    {{ $slot }}
</div>
```

现在，我们可以使用指令 `@slot` 注入内容到命名的插槽。任何不在 `@slot` 指令中的内容都会被传递到组件的 `$slot` 变量中：

```
@component('alert')
    @slot('title')
        Forbidden
    @endslot

    You are not allowed to access this resource!
@endcomponent
```

当我们在浏览器中查看这个组件内容的话，对应输出如下：

Forbidden

You are not allowed to access this resource!

这段代码的意思是通过组件名 `alert` 去查找对应的视图文件，装载到当前视图，然后通过组件中 `@slot` 定义的插槽内容去渲染插槽视图中对应的插槽位，如果组件没有为某个插槽位定义对应的插槽内容片段，则组件中的其他不在 `@slot` 片段中的内容将会用于渲染该插槽位，如果没有其他多余内容则对应插槽位为空。

传递额外数据到组件

有时候你可能需要传递额外数据到组件，出于这个原因，你可以传递数组数据作为第二个参数到 `@component` 指令，所有数据都会在组件模板中以变量方式生效：

```
@component('alert', ['foo' => 'bar'])
    ...
@endcomponent
```

组件别名

如果 Blade 组件存储在子目录中，你可能想要给它们起别名以便访问。例如，假设有一个存放
在 `resources/views/components/alert.blade.php` 的 Blade 组件，你可以使用 `component` 方法将这个组件设置别名为 `alert`（原名
是 `components.alert`）。通常，这个操作在 `AppServiceProvider` 的 `boot` 方法中完成：

```
use Illuminate\Support\Facades\Blade;

Blade::component('components.alert', 'alert');
```

组件设置别名后，就可以使用如下指令来渲染：

```
@alert(['type' => 'danger'])
    You are not allowed to access this resource!
@endalert
```

如果没有额外插槽的话也可以省略组件参数：

```
@alert
    You are not allowed to access this resource!
@endalert
```

数据显示

可以通过两个花括号包裹变量来显示传递到视图的数据，比如，如果给出如下路由：

```
Route::get('greeting', function () {
    return view('welcome', ['name' => '学院君']);
});
```

那么可以通过如下方式显示 `name` 变量的内容：

```
你好, {{ $name }}。
```

当然，不限制显示到视图中的变量内容，你还可以输出任何 PHP 函数的结果，实际上，可以将任何 PHP 代码放到 Blade 模板语句中：

```
The current UNIX timestamp is {{ time() }}.
```

注：Blade 的 `{{}}` 语句已经经过 PHP 的 `htmlentities` 函数处理以避免 XSS 攻击。

输出存在的数据

有时候你想要输出一个变量，但是不确定该变量是否被设置，我们可以通过如下 PHP 代码：

```
{ isset($name) ? $name : 'Default' } }
```

除了使用三元运算符，Blade 还提供了更简单的方式：

```
{{ $name or 'Default' }}
```

在本例中，如果 `$name` 变量存在，其值将会显示，否则将会显示 `Default`。

显示原生数据

默认情况下，Blade 的 `{{}}` 语句已经通过 PHP 的 `htmlentities` 函数处理以避免 XSS 攻击，如果你不想要数据被处理，比如要输出带 HTML 元素的富文本，可以使用如下语法：

```
Hello, {!! $name !!}. 
```

注：输出用户提供的内容时要当心，对用户提供的内容总是要使用双花括号包裹以避免直接输出 HTML 代码。

渲染 JSON 内容

有时候你可能会将数据以数组方式传递到视图再将其转化为 JSON 格式以便初始化某个 JavaScript 变量，例如：

```
<script>
    var app = <?php echo json_encode($array); ?>;
</script>
```

这样显得很麻烦，有更简便的方式来实现这个功能，那就是 Blade 的 `@json` 指令：

```
<script>
    var app = @json($array);
</script>
```

HTML 实体编码

默认情况下，Blade（以及辅助函数 `e`）会对 HTML 实体进行双重编码。如果你想要禁止双重编码，可以在 `AppServiceProvider` 的 `boot` 方法中调用 `Blade::withoutDoubleEncoding` 方法：

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Blade::withoutDoubleEncoding();
    }
}
```

Blade & JavaScript 框架

由于很多 JavaScript 框架也是用花括号来表示要显示在浏览器中的表达式，如 Vue，我们可以使用 `@` 符号来告诉 Blade 渲染引擎该表达式应该保持原生格式不作改动。比如：

```
<h1>Laravel</h1>
Hello, @{{ name }}.
```

在本例中，`@` 符在编译阶段会被 Blade 移除，但是，`{{ name }}` 表达式将会保持不变，从而可以被 JavaScript 框架正常渲染。

@verbatim 指令

如果你在模板中有很大一部分篇幅显示 JavaScript 变量，那么可以将这部分 HTML 封装在 `@verbatim` 指令中，这样就不需要在每个 Blade 输出表达式前加上 `@` 前缀：

```
@verbatim
<div class="container">
    Hello, {{ name }}.
</div>
@endverbatim
```

流程控制

除了模板继承和数据显示之外，Blade 还为常用的 PHP 流程控制提供了便利操作，例如条件语句和循环，这些快捷操作提供了一个干净、简单的方式来处理 PHP 的流程控制，同时保持和 PHP 相应语句的相似性。

If 语句

可以使用 `@if`, `@elseif`, `@else` 和 `@endif` 来构造 `if` 语句，这些指令的功能和 PHP 相同：

```
@if (count($records) === 1)
    I have one record!
@elseif (count($records) > 1)
    I have multiple records!
@else
    I don't have any records!
@endif
```

为方便起见，Blade 还提供了 `@unless` 指令，表示除非：

```
@unless (Auth::check())
    You are not signed in.
@endunless
```

此外，Blade 还提供了 `@isset` 和 `@empty` 指令，分别对应 PHP 的 `isset` 和 `empty` 方法：

```
@isset($records)
    // $records is defined and is not null...
@endisset

@empty($records)
    // $records is "empty"...
@endempty
```

认证指令

`@auth` 和 `@guest` 指令可用于快速判断当前用户是否登录：

```
@auth
    // 用户已登录...
@endauth

@guest
    // 用户未登录...
@endguest
```

如果需要的话，你也可以在使用 `@auth` 和 `@guest` 的时候指定 `认证 guard`：

```
@auth('admin')
    // The user is authenticated...
@endauth

@guest('admin')
    // The user is not authenticated...
@endguest
```

Section 指令

你可以使用 `@hasSection` 指令判断某个 `section` 中是否有内容：

```
@hasSection('navigation')
    <div class="pull-right">
        @yield('navigation')
    </div>

    <div class="clearfix"></div>
@endif
```

Switch 语句

`switch` 语句可以通过 `@switch`, `@case`, `@break`, `@default` 和 `@enswitch` 指令构建：

```
@switch($i)
    @case(1)
        First case...
        @break

    @case(2)
        Second case...
        @break
```

```
@default
Default case...
@endswitch
```

循环

除了条件语句，Blade 还提供了简单的指令用于处理 PHP 的循环结构，同样，这些指令的功能和 PHP 对应功能完全一样：

```
@for ($i = 0; $i < 10; $i++)
    The current value is {{ $i }}
@endfor

@foreach ($users as $user)
    <p>This is user {{ $user->id }}</p>
@endforeach

@forelse ($users as $user)
    <li>{{ $user->name }}</li>
@empty
    <p>No users</p>
@endforelse

@while (true)
    <p>I'm looping forever.</p>
@endwhile
```

注：在循环的时候可以使用 `$loop` 变量获取循环信息，例如是否是循环的第一个或最后一个迭代。

使用循环的时候还可以结束循环或跳出当前迭代：

```
@foreach ($users as $user)
@if ($user->type == 1)
    @continue
@endif

<li>{{ $user->name }}</li>

@if ($user->number == 5)
    @break
@endif
@endforeach
```

还可以使用指令声明来引入条件：

```
@foreach ($users as $user)
@continue($user->type == 1)
<li>{{ $user->name }}</li>
@break($user->number == 5)
@endforeach
```

`$loop` 变量

在循环的时候，可以在循环体中使用 `$loop` 变量，该变量提供了一些有用的信息，比如当前循环索引，以及当前循环是不是第一个或最后一个迭代：

```
@foreach ($users as $user)
@if ($loop->first)
    This is the first iteration.
@endif

@if ($loop->last)
    This is the last iteration.
@endif

<p>This is user {{ $user->id }}</p>
@endforeach
```

如果你身处嵌套循环，可以通过 `$loop` 变量的 `parent` 属性访问父级循环：

```
@foreach ($users as $user)
@foreach ($user->posts as $post)
@if ($loop->parent->first)
    This is first iteration of the parent loop.
@endif
@endforeach
```

```
@endforeach
```

`$loop` 变量还提供了其他一些有用的属性:

属性	描述
<code>\$loop->index</code>	当前循环迭代索引 (从 0 开始)
<code>\$loop->iteration</code>	当前循环迭代 (从 1 开始)
<code>\$loop->remaining</code>	当前循环剩余的迭代
<code>\$loop->count</code>	迭代数组元素的总数量
<code>\$loop->first</code>	是否是当前循环的第一个迭代
<code>\$loop->last</code>	是否是当前循环的最后一个迭代
<code>\$loop->depth</code>	当前循环的嵌套层级
<code>\$loop->parent</code>	嵌套循环中的父级循环变量

注释

Blade 还允许你在视图中定义注释，然而，不同于 HTML 注释，Blade 注释并不会包含到 HTML 中被返回:

```
{ {-- This comment will not be present in the rendered HTML --} }
```

PHP

在一些场景中，嵌入 PHP 代码到视图中很有用，你可以使用 `@php` 指令在模板中执行一段原生 PHP 代码:

```
@php
//
@endphp
```

注：尽管 Blade 提供了这个特性，如果过于频繁地使用它意味着你在视图模板中嵌入了过多的业务逻辑，需要注意。

包含子视图

Blade 的 `@include` 指令允许你很轻松地在一个视图中包含另一个 Blade 视图，所有父级视图中变量在被包含的子视图中依然有效:

```
<div>
    @include('shared.errors')

    <form>
        <!-- Form Contents -->
    </form>
</div>
```

上述指令会在当前目录下的 `shared` 子目录中寻找 `errors.blade.php` 文件并将其内容引入当前视图。

尽管被包含的视图可以继承所有父视图中的数据，你还可以传递额外参数到被包含的视图:

```
@include('view.name', ['some' => 'data'])
```

当然，如果你尝试包含一个不存在的视图，Laravel 会抛出错误，如果你想要包含一个有可能不存在的视图，可以使用 `@includeIf` 指令:

```
@includeIf('view.name', ['some' => 'data'])
```

如果包含的视图取决于一个给定的布尔条件，可以使用 `@includeWhen` 指令:

```
@includeWhen($boolean, 'view.name', ['some' => 'data'])
```

要包含给定数组中的第一个视图，可以使用 `@includeFirst` 指令:

```
@includeFirst(['custom.admin', 'admin'], ['some' => 'data'])
```

注：不要在 Blade 视图中使用 `__DIR__` 和 `__FILE__` 常量，因为它们会指向缓存视图的路径。

曾经有人问我 `@include` 和 `@component` 有什么区别，两者有共同之处，都用于将其他内容引入当前视图，我理解的区别在于 `@include` 用于粗粒度的视图包含，`@component` 用于细粒度的组件引入，`@component` 通过插槽机制对引入视图内容可以进行更加细粒度的控制，如果你只是引入一块视图内容片段，用 `@include` 即可，如果想要在当前视图对引入视图内容片段进行调整和控制，则可以考虑使用 `@component`。

渲染集合视图

你可以使用 Blade 的 `@each` 指令通过一行代码循环引入多个局部视图:

```
@each('view.name', $jobs, 'job')
```

该指令的第一个参数是数组或集合中每个元素要渲染的局部视图，第二个参数是你希望迭代的数组或集合，第三个参数是要分配给当前视图的变量名。举个例子，如果你要迭代一个 `jobs` 数组，通常你需要在局部视图中访问 `$job` 变量。在局部视图中可以通过 `key` 变量访问当前迭代的键。你还可以传递第四个参数到 `@each` 指令，该参数用于指定给定数组为空时渲染的视图：

```
@each('view.name', $jobs, 'job', 'view.empty')
```

注：通过 `@each` 渲染的视图不会从父视图中继承变量，如果子视图需要这个变量，可以使用 `@foreach` 和 `@include` 指令来替代。

堆栈

Blade 允许你推送内容到命名堆栈，以便在其他视图或布局中渲染。这在子视图中引入指定 JavaScript 库时很有用：

```
@push('scripts')
    <script src="/example.js"></script>
@endpush
```

推送次数不限，要渲染完整的堆栈内容，传递堆栈名称到 `@stack` 指令即可：

```
<head>
    <!-- Head Contents -->

    @stack('scripts')
</head>
```

服务注入

`@inject` 指令可以用于从服务容器中获取服务，传递给 `@inject` 的第一个参数是服务对应的变量名，第二个参数是要解析的服务类名或接口名：

```
@inject('metrics', 'App\Services\MetricsService')

<div>
    Monthly Revenue: {{ $metrics->monthlyRevenue() }}.
</div>
```

扩展 Blade

Blade 甚至还允许你自定义指令，可以使用 `directive` 方法来注册一个指令。当 Blade 编译器遇到该指令，将会传入参数并调用提供的回调。下面的例子创建了一个 `@datetime($var)` 指令格式化给定的 `DateTime` 的实例 `$var`：

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        \Blade::directive('datetime', function($expression) {
            return "<?php echo ($expression)->format('m/d/Y H:i'); ?>";
        });
    }

    /**
     * 在容器中注册绑定.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

正如你所看到的，我们可以将 `format` 方法应用到任何传入指令的表达式上，所以，在本例中，该指令最终生成的 PHP 代码如下：

```
<?php echo ($var)->format('m/d/Y H:i'); ?>
```

注：更新完 Blade 指令逻辑后，必须删除所有的 Blade 缓存视图。缓存的 Blade 视图可以通过 Artisan 命令 `view:clear` 移除。

自定义 If 语句

在定义一些简单、自定义的条件语句时，编写自定义指令往往复杂性大于必要性，因为这个原因，Blade 提供了一个 `Blade::if` 方法通过闭包的方式快速定义自定义的条件指令，例如，我们来自定义一个条件来检查当前应用的环境，我们可以在 `AppServiceProvider` 的 `boot` 方法中定义这段逻辑：

```
use Illuminate\Support\Facades\Blade;

/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    \Blade::if('env', function ($environment) {
        return app() -> environment($environment);
    });
}
```

定义好自定义条件后，就可以在模板中使用了：

```
@env('local')
    // The application is in the local environment...
@elseenv('testing')
    // The application is in the testing environment...
@else
    // The application is not in the local or testing environment...
@endenv
```

视图

创建视图

注：想要了解如何编写 Blade 模板？请先查看 [Blade 文档](#)。

视图包含应用的 HTML 代码，并将应用的控制器逻辑和表现逻辑进行分离。视图文件存放在 `resources/views` 目录。下面是一个简单视图示例：

```
<!-- 该视图存放 resources/views/greeting.php -->

<html>
    <body>
        <h1>Hello, {{ $name }}</h1>
    </body>
</html>
```

由于这个视图存放在 `resources/views/greeting.php`，我们可以通过辅助函数 `view` 像这样返回它：

```
Route::get('/', function () {
    return view('greeting', ['name' => '学院君']);
});
```

正如你所看到的，传递给 `view` 方法的第一个参数是 `resources/views` 目录下相应的视图文件的名字，第二个参数是一个数组，该数组包含了在该视图中所有有效的数据。在这个例子中，我们传递了一个 `name` 变量，在视图中通过使用 [Blade 语法](#) 将其显示出来。

当然，视图还可以存放在 `resources/views` 的子目录中，用“.”号来引用嵌套视图，例如，如果视图存放路径是 `resources/views/admin/profile.blade.php`，那我们可以这样引用它：

```
return view('admin.profile', $data);
```

判断视图是否存在

如果需要判断视图是否存在，可调用 `View` 门面上的 `exists` 方法，如果视图在磁盘存在则返回 `true`：

```
use Illuminate\Support\Facades\View;

if (View::exists('emails.customer')) {
    //
}
```

创建第一个有效视图

使用视图上的 `first` 方法可以创建给定视图数组中存在的第一个视图。这在你的应用或扩展包允许自定义或覆盖视图时很有用：

```
return view()->first(['custom.admin', 'admin'], $data);
```

当然，也可以调用 `View` 门面上的 `first` 方法来创建：

```
use Illuminate\Support\Facades\View;

return View::first(['custom.admin', 'admin'], $data);
```

传递数据到视图

在上述例子中可以看到，我们可以简单通过数组方式将数据传递到视图：

```
return view('greetings', ['name' => '学院君']);
```

以这种方式传递数据的话，`$data` 应该是一个键值对数组，在视图中，就可以使用相应的键来访问数据值，比如`<?php echo $key; ?>`。除此之外，还可以通过 `with` 方法添加独立的数据片段到视图：

```
$view = view('greeting')->with('name', '学院君');
```

在视图间共享数据

有时候，我们需要在所有视图之间共享数据片段，这时候可以使用视图门面的 `share` 方法，通常，需要在某个服务提供者的 `boot` 方法中调用 `share` 方法，你可以将其添加到 `AppServiceProvider` 或生成独立的服务提供者来存放这段代码逻辑：

```
<?php

namespace App\Providers;

use View;

class AppServiceProvider extends ServiceProvider
{
    /**
     * 启动所有应用服务
     *
     * @return void
     */
    public function boot()
    {
        View::share('key', 'value');
    }

    /**
     * 注册服务提供者
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

视图 Composer

视图 Composer 是当视图被渲染时的回调函数或类方法。如果你有一些数据要在视图每次渲染时都做绑定，可以使用视图 Composer 将逻辑组织到一个单独的地方。

在本例中，首先要在某个服务提供者中注册视图 Composer，我们将会使用 `View` 门面来访问 `Illuminate\Contracts\View\Factory` 的底层实现，记住，Laravel 不会包含默认的视图 Composer 目录，我们可以按照自己的喜好组织其路径，例如可以创建一个 `app/Http/ViewComposers` 目录：

```
<?php

namespace App\Providers;

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

class ComposerServiceProvider extends ServiceProvider
{
    /**
     * 在容器中注册绑定.
     *
     * @return void
     */
}
```

```

 * @author http://laravelacademy.org
 */
public function boot()
{
    // 使用基于类方法的 composers...
    View::composer(
        'profile', 'App\Http\ViewComposers\ProfileComposer'
    );

    // 使用基于回调函数的 composers...
    View::composer('dashboard', function ($view) {});
}

/**
 * 注册服务提供者.
 *
 * @return void
 */
public function register()
{
    //
}
}

```

注：如果创建一个新的服务提供者来包含视图 Composer 注册，需要添加该服务提供者到配置文件 config/app.php 的 providers 数组中。现在我们已经注册了视图 Composer，每次 profile 视图被渲染时都会执行 ProfileComposer@compose 方法，接下来我们来定义该 Composer 类：

```

<?php

namespace App\Http\ViewComposers;

use Illuminate\View\View;
use Illuminate\Repositories\UserRepository;

class ProfileComposer
{
    /**
     * 用户仓库实现.
     *
     * @var UserRepository
     */
    protected $users;

    /**
     * 创建一个新的属性 composer.
     *
     * @param UserRepository $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        // 依赖注入通过服务容器自动解析...
        $this->users = $users;
    }

    /**
     * 绑定数据到视图.
     *
     * @param View $view
     * @return void
     */
    public function compose(View $view)
    {
        $view->with('count', $this->users->count());
    }
}

```

```
}
```

视图被渲染前，Composer 类的 `compose` 方法被调用，同时 `Illuminate\View\View` 实例被注入该方法，从而可以使用其 `with` 方法来绑定数据到视图。

注：所有视图 Composer 都通过 [服务容器](#) 被解析，所以你可以在 Composer 类的构造函数中声明任何你需要的依赖。

添加 Composer 到多个视图

你可以传递视图数组作为 `composer` 方法的第一个参数来一次性将视图 Composer 添加到多个视图：

```
View::composer(
    ['profile', 'dashboard'],
    'App\Http\ViewComposers\MyViewComposer',
);
```

`composer` 方法还支持 `*` 通配符，从而允许将一个 Composer 添加到所有视图：

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

视图创建器

视图创建器和视图 Composer 非常类似，不同之处在于前者在视图实例化之后立即失效而不是等到视图即将渲染。使用 `View` 门面的 `creator` 方法即可注册一个视图创建器：

```
View::creator('profile', 'App\Http\ViewCreators\ProfileCreator');
```

URL 生成

简介

Laravel 提供了多个辅助函数来帮助我们在应用中生成 URL。这些函数主要用于在视图模板和 API 响应中构建链接，或者生成重定向响应。

快速入门

生成 URL

`url` 辅助函数可用于为应用生成任意 URL，并且生成的 URL 会自动使用当前请求的 `scheme`（HTTP or HTTPS）和 `host` 属性：

```
$post = App\Post::find(1);

echo url("/posts/{$post->id}");

// 输出 http://example.com/posts/1
```

访问当前 URL

如果没有传递路径信息给 `url` 辅助函数，则会返回一个 `Illuminate\Routing\UrlGenerator` 实例，从而允许你访问当前 URL 的信息：

```
// 获取不带请求字符串的当前 URL...
echo url()->current();

// 获取包含请求字符串的当前 URL...
echo url()->full();

// 获取上一个请求的完整 URL...
echo url()->previous();
```

上述每一个方法都可以通过 URL 门面进行访问，例如：

```
use Illuminate\Support\Facades\URL;

echo URL::current();
```

命名路由 URL

`route` 可用于生成指向命名路由的 URL。命名路由允许你生成不与路由中定义的实际 URL 耦合的 URL，因此，当路由的 URL 改变了，`route` 函数调用不需要做任何更改。例如，假设你的应用包含一个定义如下的路由：

```
Route::get('/post/{post}', function () {
    //
```

```
) ->name('post.show');
```

要生成指向该路由的 URL，可以这样使用 `route` 辅助函数：

```
echo route('post.show', ['post' => 1]);
```

```
// 输出 http://example.com/post/1
```

通常我们会使用 Eloquent 模型的主键来生成 URL，因此，可以传递 Eloquent 模型作为参数值，`route` 辅助函数会自动解析模型主键值，所以，上述方法还可以这么调用：

```
echo route('post.show', ['post' => $post]);
```

控制器动作 URL

`action` 辅助函数用于为控制器动作生成 URL，和路由中的定义一样，你不需要传递完整的控制器命名空间，却而代之地，传递相对于 `App\Http\Controllers` 命名空间的控制器类名即可：

```
$url = action('HomeController@index');
```

如果控制器方法接收路由参数，你可以将其作为第二个参数传递给该方法：

```
$url = action('UserController@profile', ['id' => 1]);
```

参数默认值

对某些应用而言，你可能希望为特定 URL 参数指定请求默认值，例如，假设多个路由都定义了一个 `{locale}` 变量：

```
Route::get('/{locale}/posts', function () {
    //
})->name('post.index');
```

每次调用 `route` 辅助函数都要传递 `locale` 变量显得很笨拙，所以，我们可以在当前请求中使用 `URL::defaults` 方法为这个参数定义一个默认值，我们可以在某个 `路由中间件` 中调用该方法以便可以访问当前请求：

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Support\Facades\URL;

class SetDefaultLocaleForUrls
{
    public function handle($request, Closure $next)
    {
        URL::defaults(['locale' => $request->user()->locale]);

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

一旦设置好 `locale` 参数的默认值之后，就不必在通过 `route` 辅助函数生成 URL 时每次指定传递的值了。

Session

简介

由于 HTTP 协议本身是无状态的，上一个请求与下一个请求无任何关联，为此我们引入 Session 来存储用户请求信息以解决特定场景下无状态导致的问题（比如登录、购物）。Laravel 通过简洁的 API 统一处理后端各种 Session 驱动，目前开箱支持的流行后端驱动包括 Memcached、Redis 和数据库。

学院君注： Laravel 并没有使用 PHP 内置的 Session 功能，而是自己实现了一套更加灵活更加强大的 Session 机制，核心逻辑请参考 `Illuminate\Session\Middleware\StartSession` 这个中间件，因此在 Laravel 应用中不要试图通过 `$_SESSION` 方式去获取应用的 Session 值，这是徒劳的。另外，还有一个大家都感到困惑的问题，就是在 Laravel 的控制器构造函数中是无法获取应用 Session 数据的，这是因为 Laravel 的 Session 通过 `StartSession` 中间件启动，既然是中间件就会在服务容器注册所有服务之后执行，而控制器们的构造函数都是在容器注册服务的时候执行的，所以这个时候 Session 尚未启动，有何来的获取数据呢？解决办法是将获取 Session 数据逻辑后置或者在构造函数中引入在 `StartSession` 之后执行的中间件。

配置

Session 配置文件位于 `config/session.php`。默认情况下，Laravel 使用的 Session 驱动为 `file` 驱动，这对许多应用而言是没有什么问题的。在生产环境中，你可能考虑使用 `memcached` 或者 `redis` 驱动以便获取更佳的 Session 性能，尤其是线上同一个应用部署到多台机器的时候，这是最佳实践。

Session 驱动用于定义请求的 Session 数据存放在哪里, Laravel 可以处理多种类型的驱动:

- `file` - Session 数据存储在 `storage/framework/sessions` 目录下;
- `cookie` - Session 数据存储在经过安全加密的 Cookie 中;
- `database` - Session 数据存储在数据库中;
- `memcached / redis` - Session 数据存储在 Memcached/Redis 缓存中, 访问速度最快;
- `array` - Session 数据存储在简单 PHP 数组中, 在多个请求之间是非持久化的。

注: 数组驱动通常用于运行测试以避免 Session 数据持久化。

驱动预备知识

数据库

当使用 `database` 作为 Session 驱动时, 需要设置表包含 `Session` 字段, 下面是该数据表的表结构声明:

```
Schema::create('sessions', function ($table) {
    $table->string('id')->unique();
    $table->unsignedInteger('user_id')->nullable();
    $table->string('ip_address', 45)->nullable();
    $table->text('user_agent')->nullable();
    $table->text('payload');
    $table->integer('last_activity');
});
```

你可以使用 Artisan 命令 `session:table` 在数据库中创建这张表:

```
php artisan session:table
php artisan migrate
```

Redis

在 Laravel 中使用 Redis 作为 Session 驱动前, 需要通过 Composer 安装 `predis/predis` 包。可以在 `database` 配置文件中配置 Redis 连接, 在 Session 配置文件中, `connection` 选项用于指定 Session 使用哪一个 Redis 连接。

比如我在 `config/database.php` 中为 Redis 配置了一个 Session 连接:

```
'redis' => [
    'client' => 'predis',
    'default' => [
        'host' => env('REDIS_HOST', '127.0.0.1'),
        'password' => env('REDIS_PASSWORD', null),
        'port' => env('REDIS_PORT', 6379),
        'database' => 0,
    ],
    'session' => [
        'host' => env('SESS_REDIS_HOST', '127.0.0.1'),
        'password' => env('SESS_REDIS_PASSWORD', null),
        'port' => env('SESS_REDIS_PORT', 6379),
        'database' => 0,
    ],
],
```

然后在 `config/session.php` 中配置 Session 驱动为 `redis`, 对应的 `connection` 项指向 `database` 中的 `redis.session` 配置:

```
'driver' => env('SESSION_DRIVER', 'file'),
'connection' => 'session',
```

注: `SESSION_DRIVER=redis` 在 `.env` 中设置。

这样我们就完成了 Session 驱动配置为 `redis`。

使用 Session

获取数据

在 Laravel 中主要有两种方式处理 Session 数据: 全局的辅助函数 `session`, 或者通过 `Request` 实例 (启动过程中会将 Session 数据设置到请求实例的 `session` 属性中)。

Request 实例

首先, 我们通过 `Request` 实例来访问 Session 数据, 我们可以在控制器方法中对请求实例进行依赖注入 (控制器方法依赖通过 Laravel 服务容器自动注入):

```
<?php
```

```

namespace App\Http\Controllers;

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

class UserController extends Controller{
    /**
     * 显示指定用户的属性
     *
     * @param Request $request
     * @param int $id
     * @return Response
     */
    public function show(Request $request, $id)
    {
        $value = $request->session()->get('key');

        //
    }
}

```

从 Session 中获取数据的时候，还可以传递默认值作为第二个参数到 `get` 方法，默认值在指定键在 Session 中不存在时返回。如果你传递一个闭包作为默认值到 `get` 方法，该闭包会执行并返回执行结果：

```

$value = $request->session()->get('key', 'default');

$value = $request->session()->get('key', function() {
    return 'default';
});

```

全局 Session 辅助函数

还可以使用全局的 PHP 函数 `session` 来获取和存储 Session 数据，如果只传递一个字符串参数到 `session` 方法，则返回该 Session 键对应的值；如果传递的参数是 key/value 键值对数组，则将这些数据保存到 Session：

```

Route::get('home', function () {
    // 从 session 中获取数据...
    $value = session('key');

    // 指定默认值...
    $value = session('key', 'default');

    // 存储数据到 session...
    session(['key' => 'value']);
});

```

注：通过 HTTP 请求实例和辅助函数 `session` 处理数据并无实质性差别，这两个方法在测试用例中都可以通过 `assertSessionHas` 方法进行测试。

获取所有 Session 数据

如果你想要从 Session 中获取所有数据，可以使用 `all` 方法：

```
$data = $request->session()->all();
```

判断 Session 中是否存在指定项

`has` 方法可用于检查数据项在 Session 中是否存在。如果存在并且不为 `null` 的话返回 `true`：

```

if ($request->session()->has('users')) {
    //
}

```

要判断某个值在 Session 中是否存在，即使是 `null` 的话也无所谓，则可以使用 `exists` 方法。如果值存在的话 `exists` 返回 `true`：

```

if ($request->session()->exists('users')) {
    //
}

```

存储数据

要在 Session 中存储数据，通常可以通过 `put` 方法或 `session` 辅助函数：

```

// 通过调用请求实例的 put 方法
$request->session()->put('key', 'value');

// 通过全局辅助函数 session

```

```
session(['key' => 'value']);
```

推送数据到数组 Session

`push` 方法可用于推送数据到值为数组的 Session，例如，如果 `user.teams` 键包含团队名数组，可以像这样推送新值到该数组：

```
$request->session()->push('user.teams', 'developers');
```

获取&删除数据

`pull` 方法将会通过一条语句从 Session 获取并删除数据：

```
$value = $request->session()->pull('key', 'default');
```

一次性数据

有时候你可能想要在 Session 中存储只在下个请求中有效的数据，这可以通过 `flash` 方法来实现。使用该方法存储的 Session 数据只在随后的 HTTP 请求中有效，然后将会被删除：

```
$request->session()->flash('status', '登录 Laravel 学院成功！');
```

如果你需要在更多请求中保持该一次性数据，可以使用 `reflash` 方法，该方法将所有一次性数据保留到下一个请求，如果你只是想要保存特定一次性数据，可以使用 `keep` 方法：

```
$request->session()->reflash();
$request->session()->keep(['username', 'email']);
```

删除数据

`forget` 方法从 Session 中移除指定数据，如果你想要从 Session 中移除所有数据，可以使用 `flush` 方法：

```
$request->session()->forget('key');
$request->session()->flush();
```

重新生成 Session ID

重新生成 Session ID 经常用于阻止恶意用户对应用进行 `session fixation` 攻击（关于 `session fixation` 攻击可参考这篇文章：

http://www.360doc.com/content/11/1028/16/1542811_159889635.shtml）。

如果你使用内置的 `LoginController` 的话，Laravel 会在认证期间自动重新生成 session ID，如果你需要手动重新生成 session ID，可以使用 `regenerate` 方法：

```
$request->session()->regenerate();
```

添加自定义 Session 驱动

实现驱动

自定义 Session 驱动需要实现 `SessionHandlerInterface` 接口，该接口包含少许我们需要实现的方法，比如一个基于 MongoDB 的 Session 驱动实现如下：

```
<?php

namespace App\Extensions;

class MongoHandler implements SessionHandlerInterface
{
    public function open($savePath, $sessionId) {}
    public function close() {}
    public function read($sessionId) {}
    public function write($sessionId, $data) {}
    public function destroy($sessionId) {}
    public function gc($lifetime) {}
}
```

注： Laravel 默认并没有附带一个用于包含扩展的目录，你可以将扩展放置在任何地方，这里我们创建一个 `Extensions` 目录用于存放 `MongoHandler`。

由于这些方法并不是很容易理解，所以我们接下来快速过一遍每一个方法：

- `open` 方法用于基于文件的 Session 存储系统，由于 Laravel 已经有了一个 `file` Session 驱动，所以在该方法中不需要放置任何代码，可以将其置为空方法。
- `close` 方法和 `open` 方法一样，也可以被忽略，对大多数驱动而言都用不到该方法。
- `read` 方法应该返回与给定 `$sessionId` 相匹配的 Session 数据的字符串版本，从驱动中获取或存储 Session 数据不需要做任何序列化或其它编码，因为 Laravel 已经为我们做了序列化。
- `write` 方法应该将给定 `$data` 写到持久化存储系统相应的 `$sessionId`，例如 MongoDB, Dynamo 等等。再次重申，不要做任何序列化操作，Laravel 已经为我们处理好了。
- `destroy` 方法从持久化存储中移除 `$sessionId` 对应的数据。
- `gc` 方法销毁大于给定 `$lifetime` 的所有 Session 数据，对本身拥有过期机制的系统如 Memcached 和 Redis 而言，该方法可以留空。

注册驱动

Session 驱动被实现后，需要将其注册到框架，要添加额外驱动到 Laravel Session 后端，可以使用 `Session` 门面上的 `extend` 方法。我们在某个服务提供者 如 `AppServiceProvider` 的 `boot` 方法中调用该方法（也可以自己重新创建一个新的服务提供者）：

```
<?php

namespace App\Providers;

use App\Extensions\MongoSessionStore;
use Illuminate\Support\Facades\Session;
use Illuminate\Support\ServiceProvider;

class SessionServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Session::extend('mongo', function($app) {
            // Return implementation of SessionHandlerInterface...
            return new MongoSessionStore;
        });
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

Session 驱动被注册之后，就可以在配置文件 `config/session.php` 中使用 `mongo` 驱动了。去试试吧。

表单验证

简介

Laravel 提供了多种方法来验证请求输入数据。默认情况下，Laravel 的控制器基类使用 `ValidatesRequests` trait，该 trait 提供了便捷方法通过各种功能强大的验证规则来验证输入的 HTTP 请求。

快速入门

要掌握 Laravel 强大的验证特性，让我们先看一个完整的验证表单并返回错误信息给用户的示例。

定义路由

首先，我们假定在 `routes/web.php` 文件中包含如下路由：

```
// 显示创建博客文章表单...
Route::get('post/create', 'PostController@create');

// 存储新的博客文章...
Route::post('post', 'PostController@store');
```

显然，`GET` 路由为用户显示了一个创建新的博客文章的表单，`POST` 路由将新的博客文章存储到数据库。

创建控制器

接下来，让我们看一个处理这些路由的简单控制器示例。我们先将 `store` 方法留空：

```
<?php

namespace App\Http\Controllers;
```

```

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

class PostController extends Controller
{
    /**
     * 显示创建新的博客文章的表单
     *
     * @return Response
     */
    public function create()
    {
        return view('post.create');
    }

    /**
     * 存储新的博客文章
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        // 验证并存储博客文章...
    }
}

```

编写验证逻辑

现在我们准备用验证新博客文章输入的逻辑填充 `store` 方法。我们使用 `Illuminate\Http\Request` 对象提供的 `validate` 方法来实现这一功能，如果验证规则通过，代码将会继续往下执行；反之，如果验证失败，将会抛出一个异常，相应的错误响应也会自动发送给用户。在这个传统的 HTTP 请求案例中，将会生成一个重定向响应，如果是 AJAX 请求则会返回一个 JSON 响应。

要更好地理解 `validate` 方法，让我们回顾下 `store` 方法：

```

/**
 * 存储博客文章
 *
 * @param Request $request
 * @return Response
 */
public function store(Request $request) {
    $validatedData = $request->validate([
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ]);

    // 验证通过，存储到数据库...
}

```

正如你所看到的，我们只是传入期望的验证规则到 `validate` 方法。再强调一次，如果验证失败，相应的响应会自动生成。如果验证通过，控制器将会继续往下执行。

注：实际执行代码之前，需要在数据库中创建 `posts` 数据表，因为这里用到了 `unique:posts` 这个验证规则，该规则会去数据库中查询传入标题是否已存在以保证唯一性。

首次验证失败后中止后续规则验证

有时候你可能想要在首次验证失败后停止检查该属性的其它验证规则，要实现这个功能，可以在规则属性中分配 `bail` 作为首规则：

```

$request->validate([
    'title' => 'bail|required|unique:posts|max:255',
    'body' => 'required',
]);

```

在这个例子中，如果 `title` 属性上的 `required` 规则验证失败，则不会检查 `unique` 规则，规则会按照分配顺序依次进行验证。

嵌套属性注意事项

如果 HTTP 请求中包含“嵌套”参数，可以使用“.”在验证规则中指定它们：

```

$request->validate([
    'title' => 'required|unique:posts|max:255',
    'author.name' => 'required',
    'author.desc' => 'required',
]);

```

```
]);
```

这样的验证规则适用于验证如下标签请求：

```
<form method="POST" action="{{ route('posts.store') }}">
    {{ csrf_field() }}
    <input type="text" name="title"/>
    <input type="text" name="author[name]" />
    <input type="text" name="author[desc]" />
    <textarea cols="20" rows="5" name="body"></textarea>
    <button type="submit">submit</button>
</form>
```

显示验证错误信息

那么，如果请求输入参数没有通过给定验证规则怎么办？正如前面所提到的，Laravel 将会自动将用户重定向回上一个位置。此外，所有验证错误信息会自动存放到一次性 Session，我们可以看下这个 Session 数据的数据结构：

```

array:5 [
    "_token" => "pk9a3rdEXvvuN2b0bTJ4PJQHDqsdmCjudFAChYpQ"
    "_previous" => array:1 [
        "url" => "http://blog.dev/posts/create"
    ]
    "_flash" => array:2 [
        "old" => array:2 [
            0 => "_old_input"
            1 => "errors"
        ]
    ]
    "new" => []
]
"_old_input" => array:4 [
    "_token" => "pk9a3rdEXvvuN2b0bTJ4PJQHDqsdmCjudFAChYpQ"
    "title" => "11"
    "author" => array:1 [
        "name" => null
    ]
    "body" => "33"
]
"errors" => Illuminate\Support\ViewErrorBag {#345
    #bags: array:1 [
        "default" => Illuminate\Support\MessageBag {#336
            #messages: array:1 [
                "author.name" => array:1 [
                    0 => "The author.name field is required."
                ]
            ]
            #format: ":message"
        }
    ]
}
]

```

注意我们并没有在 `GET` 路由中显式绑定错误信息到视图。这是因为 Laravel 总是从 Session 数据中检查错误信息，而且如果有的话会自动将其绑定到视图。所以，值得注意的是每次请求的所有视图中总是存在一个 `$errors` 变量，从而允许你在视图中方便而又安全地使用。`$errors` 变量是一个 `Illuminate\Support\MessageBag` 实例。想要了解更多关于该对象的信息，查看其[对应文档](#)。

注：`$errors` 变量会通过 `web` 中间件组中的 `Illuminate\View\Middleware\ShareErrorsFromSession` 中间件绑定到视图，如果使用了该中间件，那么 `$errors` 变量在视图中总是有效，从而方便你随时使用。

所以，在我们的例子中，验证失败的话用户将会被重定向到控制器的 `create` 方法，从而允许我们在视图中显示错误信息：

```

<!-- /resources/views/post/create.blade.php -->

<h1>Create Post</h1>

```

```

@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

<!-- Create Post Form -->

```

如果验证失败的话，跳转回表单页面的同时会显示错误信息：

Create Post

- The body field is required.
- The author.desc field is required.

可选字段注意事项

默认情况下，Laravel 自带了 `TrimStrings` 和 `ConvertEmptyStringsToNull` 中间件，这两个中间件位于 `App\Http\Kernel` 类的全局中间件堆栈中，因为这个原因，你需要经常将“可选”的请求字段标记为 `nullable` —— 如果你不希望验证器将 `null` 判定为无效的话。例如：

```

$this->validate($request, [
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
    'publish_at' => 'nullable|date',
]);

```

在这个例子中，我们指定 `publish_at` 字段可以为 `null` 或者有效的日期格式。如果 `nullable` 没有被添加到验证规则，验证器会将 `null` 判定为无效日期。

AJAX 请求 & 验证

在上面的例子中，我们使用了传统的表单来发送数据到应用。不过，现实场景中，很多应用使用 AJAX 请求。在 AJAX 请求中使用 `validate` 方法时，Laravel 不会生成重定向响应。取而代之的，Laravel 生成一个包含验证错误信息的 JSON 响应。该 JSON 响应会带上一个 HTTP 状态码 `422`。

表单请求验证

创建表单请求

对于更复杂的验证场景，你可能想要创建一个“表单请求”。表单请求是包含验证逻辑的自定义请求类，要创建表单验证类，可以使用 Artisan 命令 `make:request`：

```
php artisan make:request StoreBlogPost
```

生成的类位于 `app/Http/Requests` 目录下，如果该目录不存在，运行 `make:request` 命令时会替我们生成。接下来我们添加少许验证规则到该类的 `rules` 方法：

```

/**
 * 获取应用到请求的验证规则
 *
 * @return array
 */
public function rules() {
    return [
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ];
}

```

那么，验证规则如何生效呢？你所要做的就是在控制器方法中类型提示该请求类。这样表单输入请求会在控制器方法被调用之前被验证，这就是说你不需要将控制器方法和验证逻辑杂糅在一起：

```
/**
```

```

 * 存储输入的博客文章
 *
 * @param StoreBlogPostRequest $request
 * @return Response
 */
public function store(StoreBlogPost $request) {
    // The incoming request is valid...
}

```

如果验证失败，重定向响应会被生成并将用户退回上一个位置，错误信息也会被存储到一次性 Session 以便在视图中显示。如果是 AJAX 请求，带 422 状态码的 HTTP 响应将会返回给用户，该响应数据中还包含了 JSON 格式的验证错误信息。

添加验证后钩子到表单请求

如果你想要添加“验证后”钩子到表单请求，可以使用 `withValidator` 方法。该方法接收完整的构造验证器，从而允许你在验证规则执行前调用任何验证器方法：

```

/**
 * 配置验证器实例.
 *
 * @param \Illuminate\Validation\Validator $validator
 * @return void
 */
public function withValidator($validator)
{
    $validator->after(function ($validator) {
        if ($this->somethingElseIsInvalid()) {
            $validator->errors()->add('field', 'Something is wrong with this field!');
        }
    });
}

```

授权表单请求

表单请求类还包含了一个 `authorize` 方法，你可以通过该方法检查认证用户是否有权限更新指定资源。例如，如果用户尝试更新一条博客评论，那么他必须是该评论的所有者。举个例子：

```

/**
 * 判定用户是否有权限发起请求.
 *
 * @return bool
 * @translator laravelacademy.org
 */
public function authorize()
{
    $comment = Comment::find($this->route('comment'));
    return $comment && $this->user()->can('update', $comment);
}

```

由于所有请求都继承自 Laravel 请求基类，我们可以使用 `user` 方法获取当前认证用户，还要注意上面这个例子中对 `route` 方法的调用。该方法赋予用户访问被调用路由 URI 参数的权限，比如下面这个例子中的 `{comment}` 参数：

```
Route::post('comment/{comment}');
```

如果 `authorize` 方法返回 `false`，一个包含 403 状态码的 HTTP 响应会自动返回而且控制器方法将不会被执行。

如果你计划在应用的其他部分调用授权逻辑，只需在 `authorize` 方法中简单返回 `true` 即可：

```

/**
 * 判断请求用户是否经过授权
 *
 * @return bool
 */
public function authorize() {
    return true;
}

```

自定义错误消息

你可以通过重写 `messages` 方法自定义表单请求使用的错误消息，该方法应该返回属性/规则对数组及其对应错误消息：

```

/**
 * 获取被定义验证规则的错误消息
 *
 */

```

```
* @return array
* @translator laravelacademy.org
*/
public function messages() {
    return [
        'title.required' => 'A title is required',
        'body.required'  => 'A message is required',
    ];
}
```

手动创建验证器

如果你不想使用请求实例上的 `validate` 方法，可以使用 `Validator` 门面手动创建一个验证器实例，该门面提供的 `make` 方法可用于生成一个新的验证器实例：

```
<?php

namespace App\Http\Controllers;

use Validator;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PostController extends Controller{
    /**
     * 存储新的博客文章
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        $validator = Validator::make($request->all(), [
            'title' => 'required|unique:posts|max:255',
            'body'  => 'required',
        ]);

        if ($validator->fails()) {
            return redirect('post/create')
                ->withErrors($validator)
                ->withInput();
        }

        // 存储博客文章...
    }
}
```

传递给 `make` 方法的第一个参数是需要验证的数据，第二个参数是要应用到数据上的验证规则。

检查请求没有通过验证后，可以使用 `withErrors` 方法将错误数据存放到一次性 Session，使用该方法时，`$errors` 变量重定向后自动在视图间共享，从而允许你轻松将其显示给用户，`withErrors` 方法接收一个验证器、或者一个 `MessageBag`，又或者一个 PHP 数组。

自动重定向

如果你想要手动创建一个验证器实例，但仍然使用请求实例的 `validate` 方法提供的自动重定向，可以调用已存在验证器实例上的 `validate` 方法，如果验证失败，用户将会被自动重定向，或者，如果是 AJAX 请求的话，返回 JSON 响应：

```
Validator::make($request->all(), [
    'title' => 'required|unique:posts|max:255',
    'body'  => 'required',
])->validate();
```

命名错误包

如果你在单个页面上有多个表单，可能需要命名错误的 `MessageBag`，从而允许你为指定表单获取错误信息。只需要传递名称作为第二个参数给 `withErrors` 即可：

```
return redirect('register')
    ->withErrors($validator, 'login');
```

然后你就可以从 `$errors` 变量中访问命名的 `MessageBag` 实例：

```
{{ $errors->login->first('email') }}
```

验证钩子之后

验证器允许你在验证完成后添加回调，这种机制允许你轻松执行更多验证，甚至添加更多错误信息到消息集合。使用验证器实例上的 `after` 方法即可：

```
$validator = Validator::make(...);

$validator->after(function($validator) {
    if ($this->somethingElseIsInvalid()) {
        $validator->errors()->add('field', 'Something is wrong with this field!');
    }
});

if ($validator->fails()) {
    //
}
```

处理错误信息

调用 `Validator` 实例上的 `errors` 方法之后，将会获取一个 `Illuminate\Support\MessageBag` 实例，该实例中包含了多种处理错误信息的便利方法。在所有视图中默认有效的 `$errors` 变量也是一个 `MessageBag` 实例。

获取某字段的第一条错误信息

要获取指定字段的第一条错误信息，可以使用 `first` 方法：

```
$errors = $validator->errors();
echo $errors->first('email');
```

获取指定字段的所有错误信息

如果你想要简单获取指定字段的所有错误信息数组，使用 `get` 方法：

```
foreach ($errors->get('email') as $message) {
    //
}
```

如果是一个数组表单字段，可以使用 `*` 获取所有数组元素错误信息：

```
foreach ($errors->get('attachments.*') as $message) {
    //
}
```

获取所有字段的所有错误信息

要获取所有字段的所有错误信息，可以使用 `all` 方法：

```
foreach ($errors->all() as $message) {
    //
}
```

判断消息中是否存在某字段的错误信息

`has` 方法可用于判断错误信息中是否包含给定字段：

```
if ($errors->has('email')) {
    //
}
```

自定义错误信息

如果需要的话，你可以使用自定义错误信息替代默认的，有多种方法来指定自定义信息。首先，你可以传递自定义信息作为第三个参数给 `Validator::make` 方法：

```
$messages = [
    'required' => 'The :attribute field is required.',
];

$validator = Validator::make($input, $rules, $messages);
```

在本例中，`:attribute` 占位符将会被验证时实际的字段名替换，你还可以在验证消息中使用其他占位符，例如：

```
$messages = [
    'same'      => 'The :attribute and :other must match.',
    'size'      => 'The :attribute must be exactly :size.',
    'between'   => 'The :attribute must be between :min - :max.',
    'in'        => 'The :attribute must be one of the following types: :values',
];
```

为给定属性指定自定义信息

有时候你可能只想为特定字段指定自定义错误信息，可以通过“.”来实现，首先指定属性名，然后是规则：

```
$messages = [
```

```
'email.required' => '邮箱地址不能为空!',  
];
```

在语言文件中指定自定义消息

在很多案例中，你可能想要在语言文件中指定自定义消息而不是将它们直接传递给 `Validator`。要实现这个，添加消息到 `resources/lang/xx/validation.php` 语言文件的 `custom` 数组：

```
'custom' => [  
    'email' => [  
        'required' => '邮箱地址不能为空!',  
    ],  
],
```

在语言文件中指定自定义属性

如果你想要将验证消息的 `:attribute` 部分替换成自定义属性名称，可以在语言文件 `resources/lang/xx/validation.php` 的 `attributes` 数组中指定自定义名称：

```
'attributes' => [  
    'email' => '邮箱地址',  
],
```

验证规则大全

下面是有效规则及其函数列表：

- Accepted
- Active URL
- After (Date)
- After Or Equal (Date)
- Alpha
- Alpha Dash
- Alpha Numeric
- Array
- Before (Date)
- Before Or Equal (Date)
- Between
- Boolean
- Confirmed
- Date
- Date Equals
- Date Format
- Different
- Digits
- Digits Between
- Dimensions (图片文件)
- Distinct
- E-Mail
- Exists (Database)
- File
- Filled
- Image (File)
- In
- In Array
- Integer
- IP Address
- JSON
- Max
- MIME Types (File)
- MIME Type By File Extension
- Min
- Nullable
- Not In
- Numeric
- Present
- Regular Expression
- Required
- Required If
- Required Unless
- Required With
- Required With All
- Required Without
- Required Without All

- Same
- Size
- String
- Timezone
- Unique (Database)
- URL

accepted

验证字段的值必须是 `yes`、`on`、`1` 或 `true`，这在“同意服务协议”时很有用。

active_url

验证字段必须是基于 PHP 函数 `dns_get_record` 的，有 A 或 AAAA 记录的值。

after:date

验证字段必须是给定日期之后的一个值，日期将会通过 PHP 函数 `strtotime` 传递：

```
'start_date' => 'required|date|after:tomorrow'
```

你可以指定另外一个与日期进行比较的字段，而不是传递一个日期字符串给 `strtotime` 执行：

```
'finish_date' => 'required|date|after:start_date'
```

after_or_equal:date

验证字段必须是大于等于给定日期的值，更多信息，请参考 `after:date` 规则。

alpha

验证字段必须是字母。

alpha_dash

验证字段可以包含字母和数字，以及破折号和下划线。

alpha_num

验证字段必须是字母或数字。

array

验证字段必须是 PHP 数组。

before:date

和 `after:date` 相对，验证字段必须是指定日期之前的一个数值，日期将会传递给 PHP `strtotime` 函数。

before_or_equal:date

验证字段必须小于等于给定日期。日期将会传递给 PHP 的 `strtotime` 函数。

between:min,max

验证字段大小在给定的最小值和最大值之间，字符串、数字、数组和文件都可以像使用 `size` 规则一样使用该规则：

```
'name' => 'required|between:1,20'
```

boolean

验证字段必须可以被转化为布尔值，接收 `true`, `false`, `1`, `0`, `"1"` 和 `"0"` 等输入。

confirmed

验证字段必须有一个匹配字段 `foo_confirmation`，例如，如果验证字段是 `password`，必须输入一个与之匹配的 `password_confirmation` 字段。

date

验证字段必须是一个基于 PHP `strtotime` 函数的有效日期

date_equals:date

验证字段必须等于给定日期，日期会被传递到 PHP `strtotime` 函数。

date_format:format

验证字段必须匹配指定格式，可以使用 PHP 函数 `date` 或 `date_format` 验证该字段。

different:field

验证字段必须是一个和指定字段不同的值。

digits:value

验证字段必须是数字且长度为 `value` 指定的值。

digits_between:min,max

验证字段数值长度必须介于最小值和最大值之间。

dimensions

验证的图片尺寸必须满足该规定参数指定的约束条件：

```
'avatar' => 'dimensions:min_width=100,min_height=200'
```

有效的约束条件包括：`min_width`, `max_width`, `min_height`, `max_height`, `width`, `height`, `ratio`。

`ratio` 约束宽度/高度的比率，这可以通过表达式 `3/2` 或浮点数 `1.5` 来表示：

```
'avatar' => 'dimensions:ratio=3/2'
```

由于该规则要求多个参数，可以使用 `Rule::dimensions` 方法来构造该规则：

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'avatar' => [
        'required',
        Rule::dimensions()->maxWidth(1000)->maxHeight(500)->ratio(3 / 2),
    ],
]);
```

distinct

处理数组时，验证字段不能包含重复值：

```
'foo.*.id' => 'distinct'
```

email

验证字段必须是格式正确的电子邮件地址

exists:table,column

验证字段必须存在于指定数据表

基本使用：

```
'state' => 'exists:states'
```

指定自定义列名：

```
'state' => 'exists:states,abbreviation'
```

有时，你可能需要为 `exists` 查询指定要使用的数据库连接，这可以在表名前通过`.前置数据库连接`来实现：

```
'email' => 'exists:connection.staff,email'
```

如果你想要自定义验证规则执行的查询，可以使用 `Rule` 类来定义规则。在这个例子中，我们还以数组形式指定了验证规则，而不是使用 `|` 字符来限定它们：

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'email' => [
        'required',
        Rule::exists('staff')->where(function ($query) {
            $query->where('account_id', 1);
        }),
    ],
]);
```

file

验证字段必须是上传成功的文件。

filled

验证字段如果存在则不能为空。

image

验证文件必须是图片（jpeg、png、bmp、gif 或者 svg）

in:foo,bar...

验证字段值必须在给定的列表中，由于该规则经常需要我们对数组进行 `implode`，我们可以使用 `Rule::in` 来构造这个规则：

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'zones' => [
        'required',
        Rule::in(['first-zone', 'second-zone']),
    ],
]);
```

in_array:另一个字段

验证字段必须在另一个字段值中存在。

integer

验证字段必须是整型。

ip

验证字段必须是 IP 地址。

ipv4

验证字段必须是 IPv4 地址。

ipv6

验证字段必须是 IPv6 地址。

json

验证字段必须是有效的 JSON 字符串

max:value

验证字段必须小于等于最大值，和字符串、数值、数组、文件字段的 `size` 规则使用方式一样。

mimetypes: text/plain...

验证文件必须匹配给定的 MIME 文件类型之一：

```
'video' => 'mimetypes:video/avi,video/mpeg,video/quicktime'
```

为了判断上传文件的 MIME 类型，框架将会读取文件内容来猜测 MIME 类型，这可能会和客户端 MIME 类型不同。

mimes:foo,bar,...

验证文件的 MIME 类型必须是该规则列出的扩展类型中的一个

MIME 规则的基本使用：

```
'photo' => 'mimes:jpeg,bmp,png'
```

尽管你只是指定了扩展名，该规则实际上验证的是通过读取文件内容获取到的文件 MIME 类型。

完整的 MIME 类型列表及其相应的扩展可以在这里找到：<http://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types>

min:value

与 `max:value` 相对，验证字段必须大于等于最小值，对字符串、数值、数组、文件字段而言，和 `size` 规则使用方式一致。

nullable

验证字段可以是 `null`，这在验证一些可以为 `null` 的原始数据如整型或字符串时很有用。

not_in:foo,bar,...

验证字段值不能在给定列表中，和 `in` 规则类似，我们可以使用 `Rule::notIn` 方法来构建规则：

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'toppings' => [
        'required',
        Rule::notIn(['sprinkles', 'cherries']),
    ],
]);
```

numeric

验证字段必须是数值

present

验证字段必须出现在输入数据中但可以为空。

regex:pattern

验证字段必须匹配给定正则表达式。

注：使用 `regex` 模式时，规则必须放在数组中，而不能使用管道分隔符，尤其是正则表达式中已经使用了管道符号时。

required

验证字段值不能为空，以下情况字段值都为空：

- 值为 `null`
- 值是空字符串
- 值是空数组或者空的 `Coutable` 对象
- 值是上传文件但路径为空

required_if:anotherfield,value,...

验证字段在 `anotherfield` 等于指定值 `value` 时必须存在且不能为空

required_unless:anotherfield,value,...

除非 `anotherfield` 字段等于 `value`，否则验证字段不能为空

required_with:foo,bar,...

验证字段只有在任一其它指定字段存在的情况下才是必须的

required_with_all:foo,bar,...

验证字段只有在所有指定字段存在的情况下才是必须的

required_without:foo,bar,...

验证字段只有当任一指定字段不存在的情况下才是必须的

required_without_all:foo,bar,...

验证字段只有当所有指定字段不存在的情况下才是必须的

same:field

给定字段和验证字段必须匹配

size:value

验证字段必须有和给定值 `value` 相匹配的尺寸/大小，对字符串而言，`value` 是相应的字符数目；对数值而言，`value` 是给定整型值；对数组而言，`value` 是数组长度；对文件而言，`value` 是相应的文件千字节数（KB）

string

验证字段必须是字符串，如果允许字段为空，需要分配 `nullable` 规则到该字段。

timezone

验证字符必须是基于 PHP 函数 `timezone_identifiers_list` 的有效时区标识

unique:table,column,except,idColumn

验证字段在给定数据表上必须是唯一的，如果不指定 `column` 选项，字段名将作为默认 `column`。

指定自定义列名：

```
'email' => 'unique:users,email_address'
```

自定义数据库连接

有时候，你可能需要自定义验证器生成的数据库连接，正如上面所看到的，设置 `unique:users` 作为验证规则将会使用默认数据库连接来查询数据库。要覆盖默认连接，在数据表名后使用“.”指定连接：

```
'email' => 'unique:connection.users,email_address'
```

强制一个忽略给定 ID 的唯一规则：

有时候，你可能希望在唯一检查时忽略给定 ID，例如，考虑一个包含用户名、邮箱地址和位置的“更新属性”界面，你将要验证邮箱地址是唯一的，然而，如果用户只改变用户名字段而并没有改变邮箱字段，你不想要因为用户已经拥有该邮箱地址而抛出验证错误，你只想要在用户提供的邮箱已经被别人使用的情况下才抛出验证错误。

要告诉验证器忽略用户 ID，可以使用 `Rule` 类来定义这个规则，我们还要以数组方式指定验证规则，而不是使用 | 来界定规则：

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'email' => [
        'required',
        Rule::unique('users')->ignore($user->id),
    ],
]);
```

如果你的数据表使用主键字段不是 `id`，可以在调用 `ignore` 方法的时候指定字段名称：

```
'email' => Rule::unique('users')->ignore($user->id, 'user_id')
```

添加额外的 `where` 子句：

使用 `where` 方法自定义查询的时候还可以指定额外查询约束，例如，下面我们来添加一个验证 `account_id` 为 1 的约束：

```
'email' => Rule::unique('users')->where(function ($query) {
    $query->where('account_id', 1);
})
```

url

验证字段必须是有效的 URL。

添加条件规则

存在时验证

在某些场景下，你可能想要只有某个字段存在的情况下进行验证检查，要快速实现这个，添加 `sometimes` 规则到规则列表：

```
$v = Validator::make($data, [
    'email' => 'sometimes|required|email',
]);
```

在上例中，`email` 字段只有存在于 `$data` 数组时才会被验证。

注：如果你尝试验证一个总是存在但可能为空的字段时，参考[可选字段注意事项](#)。

复杂条件验证

有时候你可能想要基于更复杂的条件逻辑添加验证规则。例如，你可能想要只有在另一个字段值大于 100 时才要求一个给定字段是必须的，或者，你可能需要只有当另一个字段存在时两个字段才都有给定值。添加这个验证规则并不是一件头疼的事。首先，创建一个永远不会改变的静态规则到 `Validator` 实例：

```
$v = Validator::make($data, [
    'email' => 'required|email',
    'games' => 'required|numeric',
]);
```

让我们假定我们的 Web 应用服务于游戏收藏者。如果一个游戏收藏者注册了我们的应用并拥有超过 100 个游戏，我们想要他们解释为什么他们会有很多游戏，例如，也许他们在运营一个游戏二手店，又或者他们只是喜欢收藏。要添加这种条件，我们可以使用 `Validator` 实例上的 `sometimes` 方法：

```
$v->sometimes('reason', 'required|max:500', function($input) {
    return $input->games >= 100;
});
```

传递给 `sometimes` 方法的第一个参数是我们需要有条件验证的名称字段，第二个参数是我们想要添加的规则，如果作为第三个参数的闭包返回 `true`，规则被添加。该方法让构建复杂条件验证变得简单，你甚至可以一次为多个字段添加条件验证：

```
$v->sometimes(['reason', 'cost'], 'required', function($input) {
    return $input->games >= 100;
});
```

注：传递给闭包的 `$input` 参数是 `Illuminate\Support\Fluent` 的一个实例，可用于访问输入和文件。

验证数组输入

验证表单数组输入字段不再是件痛苦的事情，例如，如果进入的 HTTP 请求包含 `photos[profile]` 字段，可以这么验证：

```
$validator = Validator::make($request->all(), [
    'photos.profile' => 'required|image',
]);
```

我们还可以验证数组的每个元素，例如，要验证给定数组输入中每个 `email` 是否是唯一的，可以这么做（这种针对提交的数组字段是二维数组，如 `person[] [email]` 或 `person[test] [email]`）：

```
$validator = Validator::make($request->all(), [
    'person.*.email' => 'email|unique:users',
    'person.*.first_name' => 'required_with:person.*.last_name',
]);
```

```
]);
```

类似地，在语言文件中你也可以使用 `*` 字符指定验证消息，从而可以使用单个验证消息定义基于数组字段的验证规则：

```
'custom' => [
    'person.*.email' => [
        'unique' => '每个人的邮箱地址必须是唯一的',
    ],
],
```

自定义验证规则

使用 Rule 对象

如上所述，Laravel 提供了多种有用的验证规则；不过，你可能还是需要指定一些自己的验证规则。注册自定义验证规则的一种方法是使用规则对象，要生成一个新的规则对象，可以使用 Artisan 命令 `make:rule`。下面我们使用这个命令来生成一个用于验证字符串是否是大写的规则，生成的新规则对象类位于 `app/Rules` 目录：

```
php artisan make:rule Uppercase
```

规则创建之后，就可以定义行为方法，一个规则对象包含两个方法：`passes` 和 `message`，`passes` 方法接收属性值和名称，并且基于属性值是否有效返回 `true` 或 `false`。`message` 方法用于在验证失败时返回验证错误消息：

```
<?php

namespace App\Rules;

use Illuminate\Contracts\Validation\Rule;

class Uppercase implements Rule
{
    /**
     * Determine if the validation rule passes.
     *
     * @param string $attribute
     * @param mixed $value
     * @return bool
     */
    public function passes($attribute, $value)
    {
        return strtoupper($value) === $value;
    }

    /**
     * Get the validation error message.
     *
     * @return string
     */
    public function message()
    {
        return 'The :attribute must be uppercase.';
    }
}
```

当然，你可以在 `message` 方法中调用辅助函数 `trans` 来返回一个在语言文件中定义的错误消息：

```
/**
 * Get the validation error message.
 *
 * @return string
 */
public function message()
{
    return trans('validation.uppercase');
}
```

规则定义好之后，就可以将其以规则对象实例的方式和其他验证规则一起提供给验证器：

```
use App\Rules\Uppercase;

$request->validate([
    'name' => ['required', new Uppercase],
]);
```

使用闭包

如果在整个应用只需要一次自定义规则的功能，可以使用闭包替代规则对象。该闭包接收属性名、属性值以及验证失败时调用的 `$fail` 回调：

```
$validator = Validator::make($request->all(), [
    'title' => [
        'required',
        'max:255',
        function($attribute, $value, $fail) {
            if ($value === 'foo') {
                return $fail($attribute.' is invalid.');
            }
        },
    ],
]);
```

使用扩展

另一个注册自定义验证规则的方式是使用 `Validator` 门面上的 `extend` 方法。我们在某个服务提供者（如 `AppServiceProvider`）中使用该方法注册一个自定义验证规则：

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\Validator;

class AppServiceProvider extends ServiceProvider
{
    /**
     * 启动应用服务
     *
     * @return void
     */
    public function boot()
    {
        Validator::extend('foo', function($attribute, $value, $parameters, $validator) {
            return $value == 'foo';
        });
    }

    /**
     * 注册服务提供者
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

自定义验证器闭包接收四个参数：要验证的属性名称、属性值、传递给规则的参数数组以及 `Validator` 实例。

你还可以传递类和方法到 `extend` 方法而不是闭包：

```
Validator::extend('foo', 'FooValidator@validate');
```

定义错误信息

你还需要为自定义规则定义错误信息。你可以使用内联自定义消息数组或者在验证语言文件中添加条目来实现这一功能。消息应该被放到数组的第一维，而不是在只用于存放属性指定错误信息的 `custom` 数组内：

```
"foo" => "Your input was invalid!",
"accepted" => "The :attribute must be accepted.",
// 验证错误信息其它部分...
```

当创建一个自定义验证规则时，你可能有时候需要为错误信息定义自定义占位符，可以通过创建自定义验证器然后调用 `Validator` 门面上的 `replacer` 方法来实现。在服务提供者的 `boot` 方法中编写如下代码：

```
/**
 * 启动应用服务
 *
```

```
* @return void
* @translator laravelacademy.org
*/
public function boot() {
    Validator::extend(...);
    Validator::replacer('foo', function($message, $attribute, $rule, $parameters) {
        return str_replace(...);
    });
}
```

隐式扩展

默认情况下，被验证的属性如果没有提供或者验证规则为 `required` 而值为空，那么正常的验证规则，包括自定义扩展将不会执行。例如，`unique` 规则将不会检验 `null` 值：

```
$rules = ['name' => 'unique'];
$input = ['name' => null];
Validator::make($input, $rules)->passes(); // true
```

如果要求即使为空时也要验证属性，则必须要暗示属性是必须的，要创建一个隐式扩展，可以使用 `Validator::extendImplicit()` 方法：

```
Validator::extendImplicit('foo', function($attribute, $value, $parameters, $validator) {
    return $value == 'foo';
});
```

注：一个隐式扩展仅仅暗示属性是必须的，至于它到底是缺失的还是空值这取决于你。

异常处理

简介

Laravel 默认已经为我们配置好了错误和异常处理，我们在 `App\Exceptions\Handler` 类中触发异常并将响应返回给用户。在本文档中我们将深入探讨这个类。

注：关于 Laravel 异常处理底层原理和实现可以参考学院的这篇教程了解更多 —— [《深入探讨 PHP 错误异常处理机制及 Laravel 框架底层的相应实现》](#)

配置

配置文件 `config/app.php` 中的 `debug` 配置项控制浏览器显示的错误信息数量。默认情况下，该配置项通过 `.env` 文件中的环境变量 `APP_DEBUG` 进行设置。

对本地开发而言，你应该设置环境变量 `APP_DEBUG` 值为 `true`。在生产环境，该值应该被设置为 `false`。如果在生产环境被设置为 `true`，就有可能将一些敏感的配置值暴露给终端用户。

异常处理器

所有异常都由类 `App\Exceptions\Handler` 处理，该类包含两个方法：`report` 和 `render`。下面我们详细阐述这两个方法。

report 方法

`report` 方法用于记录异常并将其发送给外部服务如 `Bugsnag` 或 `Sentry`，默认情况下，`report` 方法只是将异常传递给异常被记录的基类，当然你也可以按自己的需要记录异常并进行相关处理。

例如，如果你需要以不同方式报告不同类型的异常，可使用 PHP 的 `instanceof` 比较操作符：

```
/**
 * 报告或记录异常
 *
 * This is a great spot to send exceptions to Sentry, Bugsnag, etc.
 *
 * @param \Exception $e
 * @return void
 * @translator laravelacademy.org
 */
public function report(Exception $e) {
    if ($e instanceof CustomException) {
        //
    }

    return parent::report($e);
}
```

注：可以考虑使用 `可报告的异常` 来取代在 `reports` 方法中进行大量的 `instanceof` 检查。

report 辅助函数

有时候你可能需要报告一个异常并继续处理当前请求。辅助函数 `report` 允许你使用异常处理器的 `report` 方法快速报告一个异常而不会渲染错误页：

```
public function isValid($value)
{
    try {
        // Validate the value...
    } catch (Exception $e) {
        report($e);

        return false;
    }
}
```

通过类型忽略异常

异常处理器的 `$dontReport` 属性包含一个不会被记录的异常类型数组，默认情况下，404 错误异常不会被写到日志文件，如果需要的话你可以添加其他异常类型到这个数组：

```
/**
 * 不应该被报告的异常类型列表。
 *
 * @var array
 */
protected $dontReport = [
    \Illuminate\Auth\AuthenticationException::class,
    \Illuminate\Auth\Access\AuthorizationException::class,
    \Symfony\Component\HttpKernel\Exception\HttpException::class,
    \Illuminate\Database\Eloquent\ModelNotFoundException::class,
    \Illuminate\Validation\ValidationException::class,
];
```

render 方法

`render` 方法负责将给定异常转化为发送给浏览器的 HTTP 响应，默认情况下，异常被传递给为你生成响应的基类。当然，你也可以按照自己的需要检查异常类型或者返回自定义响应：

```
/**
 * 将异常渲染到 HTTP 响应中
 *
 * @param \Illuminate\Http\Request $request
 * @param \Exception $exception
 * @return \Illuminate\Http\Response
 */
public function render($request, Exception $exception){
    if ($e instanceof CustomException) {
        return response()->view('errors.custom', [], 500);
    }

    return parent::render($request, $exception);
}
```

可报告 & 可渲染的异常

除了在异常处理器的 `report` 和 `render` 方法中进行异常类型检查外，还可以在自定义异常中直接定义 `report` 和 `render` 方法。当异常中存在这些方法时，框架会自动调用它们：

```
<?php

namespace App\Exceptions;

use Exception;

class RenderException extends Exception
{
    /**
     * Report the exception.
     *
     * @return void
     */
    public function report()
    {
        //
    }
}
```

```

 * Render the exception into an HTTP response.
 *
 * @param \Illuminate\Http\Request
 * @return \Illuminate\Http\Response
 */
public function render($request)
{
    return response(...);
}

```

HTTP 异常

有些异常描述来自服务器的 HTTP 错误码，例如，这可能是一个“页面未找到”错误（404），“认证失败错误”（401）亦或是程序出错造成的 500 错误，为了在应用中生成这样的响应，可以使用 `abort` 辅助函数：

```
abort(404);
```

`abort` 辅助函数会立即引发一个会被异常处理器渲染的异常，此外，你还可以像这样提供响应描述：

```
abort(403, '未授权操作');
```

该方法可在请求生命周期的任何时间点使用。

自定义 HTTP 错误页面

在 Laravel 中，返回不同 HTTP 状态码的错误页面很简单，例如，如果你想要自定义 404 错误页面，创建一个 `resources/views/errors/404.blade.php` 文件，该视图文件用于渲染程序返回的所有 404 错误。需要注意的是，该目录下的视图命名应该和相应的 HTTP 状态码相匹配。`abort` 函数触发的 `HttpException` 异常会以 `$exception` 变量的方式传递给视图：

```
<h2>{{ $exception->getMessage() }}</h2>
```

日志

简介

为了帮助你了解更多关于应用中所发生的事情，Laravel 提供了强大的日志服务来记录日志信息到文件、系统错误日志、甚至是 Slack 以便通知整个团队。

在日志引擎之下，Laravel 集成了 `Monolog` 日志库以便提供各种功能强大的日志处理器，从而允许你通过它们来定制自己应用的日志处理。

配置

应用日志系统的所有配置都存放在配置文件 `config/logging.php` 中，该文件允许你配置应用的日志通道，因此请务必查看每个可用通道及其配置项。下面我们就来看看其中某些配置项。

默认情况下，Laravel 使用 `stack` 通道来记录日志信息，`stack` 通道被用于聚合多个日志通道到单个通道，更多关于构建 `stack` 的信息，请查看[下面的文档](#)。

配置通道名称

默认情况下，`Monolog` 通过与当前环境匹配的「通道名」实例化，例如 `production` 或 `local`，要改变这个值，添加 `name` 项到通道配置：

```
'stack' => [
    'driver' => 'stack',
    'name' => 'channel-name',
    'channels' => ['single', 'slack'],
],
```

配置 Slack 通道

`slack` 通道需要一个 `url` 配置项，这个 URL 需要和你配置的 Slack 团队请求 URL 相匹配。

构建日志堆栈

如上所述，`stack` 驱动允许你将多个通道合并到单个日志通道，为了说明如何实现，让我们看一个你可能在生产环境中看到的示例配置：

```
'channels' => [
    'stack' => [
        'driver' => 'stack',
        'channels' => ['syslog', 'slack'],
    ],
    'syslog' => [
        'driver' => 'syslog',
        'level' => 'debug',
    ],
]
```

```
],
'slack' => [
    'driver' => 'slack',
    'url' => env('LOG_SLACK_WEBHOOK_URL'),
    'username' => 'Laravel Log',
    'emoji' => ':boom:',
    'level' => 'critical',
],
],
```

我们来剖析这个配置。首先，注意 `stack` 通道通过 `channels` 项将聚合了其他两个通道：`syslog` 和 `slack`。因此，记录日志信息时，这两个通道都有机会记录信息。

日志级别

注意上述示例中 `syslog` 和 `slack` 通道配置中出现的 `level` 配置项，这个配置项决定了日志信息被通道记录所必须达到的最低「级别」。为 Laravel 提供日志服务的 Monolog，支持定义在 [RFC 5424 规范](#) 中的所有日志级别：`emergency`、`alert`、`critical`、`error`、`warning`、`notice`、`info` 和 `debug`。

因此，假设我们使用 `debug` 方法来记录日志信息：

```
Log::debug('An informational message.');
```

鉴于我们的配置，`syslog` 通道将会将信息记录到系统日志；不过，由于错误消息不是 `critical` 或更高级别，将不会发送到 Slack。但是，如果我们记录的是 `emergency` 级别的信息，就会被发送到系统日志和 Slack，因为 `emergency` 级别高于两个通道的最低级别门槛：

```
Log::emergency('The system is down!');
```

写入日志信息

你可以使用 `Log` 门面记录日志信息，如上所述，日志系统提供了定义在 [RFC 5424 规范](#) 中的八种日志级别：`emergency`、`alert`、`critical`、`error`、`warning`、`notice`、`info` 和 `debug`：

```
Log::emergency($error);
Log::alert($error);
Log::critical($error);
Log::error($error);
Log::warning($error);
Log::notice($error);
Log::info($error);
Log::debug($error);
```

因此，你可以调用其中的任意一个方法来记录相应级别的日志信息，默认情况下，信息会被写入到通过配置文件 `config/logging.php` 所配置的默认通道：

```
<?php

namespace App\Http\Controllers;

use App\User;
use Illuminate\Support\Facades\Log;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * 显示指定用户的属性
     *
     * @param int $id
     * @return Response
     */
    public function showProfile($id)
    {
        Log::info('Showing user profile for user: '.$id);
        return view('user.profile', ['user' => User::findOrFail($id)]);
    }
}
```

上下文信息

上下文数据也会以数组形式传递给日志方法，然后和日志信息一起被格式化和显示：

```
Log::info('User failed to login.', ['id' => $user->id]);
```

写入指定通道

有时候你可能希望将日志信息记录到某个通道而不是应用的默认通道。要实现这个目的，你可以使用 `Log` 门面上的 `channel` 方法来获取配置文件定义的通道并将日志写入进去：

```
Log::channel('slack')->info('Something happened!');
```

如果你想要创建一个由多个通道组成的按需日志堆栈，可以使用 `stack` 方法：

```
Log::stack(['single', 'slack'])->info('Something happened!');
```

为通道自定义 Monolog

有时候你可能需要在某个通道中完全控制 Monolog 的配置，例如，你可能想要为给定通道的处理器配置一个自定义的 Monolog `FormatterInterface` 实现。

作为开始，我们在频道的配置上定义一个 `tap` 数组，这个 `tap` 数组需要包含可以自定义创建后的 Monolog 实例的类列表：

```
'single' => [
    'driver' => 'single',
    'tap' => [App\Logging\CustomizeFormatter::class],
    'path' => storage_path('logs/laravel.log'),
    'level' => 'debug',
],
```

在通道上配置好 `tap` 项后，就可以定义自定义 Monolog 实例的类了。这个类只需要一个获取 `\Illuminate\Log\Logger` 实例的方法：`__invoke`，`\Illuminate\Log\Logger` 实例会代理所有调用底层 Monolog 实例的方法：

```
<?php

namespace App\Logging;

class CustomizeFormatter
{
    /**
     * Customize the given logger instance.
     *
     * @param \Illuminate\Log\Logger $logger
     * @return void
     */
    public function __invoke($logger)
    {
        foreach ($logger->getHandlers() as $handler) {
            $handler->setFormatter(...);
        }
    }
}
```

注：所有「`tap`」类都通过服务容器解析，所以他们需要的所有构造函数依赖都会被自动注入。

创建自定义频通道

如果你想要定义一个完整的自定义通道从而可以完全控制 Monolog 的实例化和配置，可以在配置文件 `config/logging.php` 中指定一个 `custom` 驱动类型。此外，你的配置中还要包含一个 `via` 项来指定被调用来创建 Monolog 实例的类：

```
'channels' => [
    'custom' => [
        'driver' => 'custom',
        'via' => App\Logging\CreateCustomLogger::class,
    ],
],
```

配置好 `custom` 通道后，就可以定义创建 Monolog 实例的类了，这个类只需要一个返回 Monolog 实例的方法：`__invoke`：

```
<?php

namespace App\Logging;

use Monolog\Logger;

class CreateCustomLogger
{
    /**
     * Create a custom Monolog instance.
     *
     * @param array $config
     * @return \Monolog\Logger
     */
    public function __invoke(array $config)
    {
        return new Logger(...);
    }
}
```

```
}
```

五、前端开发

本地化

简介

Laravel 的本地化特性允许你在应用中轻松实现多语言支持。语言字符串默认存放在 `resources/lang` 目录中，该目录包含了应用支持的每种语言的子目录：

```
/resources
  /lang
    /en
      messages.php
    /es
      messages.php
```

所有语言文件都返回一个键值对数组，例如：

```
<?php

return [
  'welcome' => 'Welcome to LaravelAcademy.org'
];
```

配置 Locale 选项

应用默认语言存放在配置文件 `config/app.php` 中，当然，你可以修改该值来满足应用需要。你还可以在运行时使用 `App` 门面上的 `setLocale` 方法改变当前语言：

```
Route::get('welcome/{locale}', function ($locale) {
  App::setLocale($locale);
  //
});
```

你还可以配置一个“备用语言”，当当前语言不包含给定语言行时备用语言被返回。和默认语言一样，备用语言也在配置文件 `config/app.php` 中配置：

```
'fallback_locale' => 'en',
```

判断当前的本地语言

你可以使用 `App` 门面上的 `getLocale` 和 `isLocale` 方法来获取当前的本地语言或者检查是否与给定本地语言匹配：

```
$locale = App::getLocale();

if (App::isLocale('en')) {
  //
}
```

定义翻译字符串

使用缩写键

通常，翻译字符串存放在 `resources/lang` 目录下的文件中，这个目录包含了应用所支持的每种语言所对应的子目录：

```
/resources
  /lang
    /en
      messages.php
    /es
      messages.php
```

所有语言文件都返回有对应缩写键的字符串数组，例如：

```
<?php

// resources/lang/en/messages.php

return [
  'welcome' => 'Welcome to LaravelAcademy.org'
```

```
];
```

使用翻译字符串作为键

对于那些对翻译有重度要求的应用来说，为每个字符串定义一个“short key”在视图中引用的时候会变得越来越难以理解，甚至引起困惑。因为这个原因， Laravel 还支持使用“默认”翻译字符串作为键来定义翻译字符串。

使用翻译字符串作为键的翻译文件以 JSON 文件的方式存放在 `resources/lang` 目录下。例如，如果你的应用有一个西班牙版翻译，需要创建一个 `resources/lang/es.json` 文件：

```
{
    "I love programming.": "Me encanta la programación."
}
```

获取翻译字符串

你可以使用辅助函数 `__` 从语言文件中获取行，该方法接收文件和翻译字符串的键作为第一个参数，举个例子，我们从语言文件 `resources/lang/messages.php` 中获取 `welcome` 对应的翻译字符串：

```
echo __('messages.welcome');
echo __('I love programming.');
```

当然如果你使用了 Blade 模板引擎，可以使用 `{{ }}` 语法打印翻译字符串或者使用 `@lang` 指令：

```
{{ __('messages.welcome') }}
@lang('messages.welcome')
```

如果指定的翻译字符串不存在，`__` 函数将返回翻译字符串的键，所以，使用上面的例子，如果翻译字符串不存在的话，`__` 函数将返回 `messages.welcome`。

替换翻译字符串中的参数

如果需要的话，你可以在翻译字符串中定义占位符，所有的占位符都有一个`:`前缀，例如，你可以用占位符 `:name` 定义一个 `welcome` 消息：

```
'welcome' => 'Welcome, :name',
```

要在获取翻译字符串的时候替换占位符，传递一个替换数组作为 `__` 函数的第二个参数：

```
echo __('messages.welcome', ['name' => 'laravel']);
```

如果占位符都是大写的，或者首字母是大写的，那么相应的，传入的值也会保持和占位符格式一致：

```
'welcome' => 'Welcome, :NAME', // Welcome, LARAVEL
'goodbye' => 'Goodbye, :Name', // Goodbye, Laravel
```

复数

复数是一个复杂的问题，因为不同语言对复数有不同的规则，通过使用管道符“|”，你可以区分一个字符串的单数和复数形式：

```
'apples' => 'There is one apple|There are many apples',
```

你还可以创建为多个数字区间指定翻译字符串的、更复杂的复数规则：

```
'apples' => '{0} There are none|[1,19] There are some|[20,*] There are many',
```

之后，你可以使用 `trans_choice` 函数获取给定行数的语言行，在本例中，由于行数大于 1，将会返回翻译字符串的复数形式：

```
echo trans_choice('messages.apples', 10);
```

还可以在复数字符串中定义占位符属性，这些占位符会被传递给 `trans_choice` 函数的第三个数组参数替换：

```
'minutes_ago' => '{1} :value minute ago|[2,*] :value minutes ago',
```

```
echo trans_choice('time.minutes_ago', 5, ['value' => 5]);
```

覆盖 Vendor 包的语言文件

有些扩展包可能会自己处理语言文件。你可以通过将自己的文件放在 `resources/lang/vendor/{package}/{locale}` 目录下来覆盖它们而不是破坏这些包的核心文件来调整这些句子。

所以，举个例子，如果你需要覆盖名为 `skyrim/hearthfire` 扩展包中的 `messages.php` 文件里的英文句子，可以创建一个 `resources/lang/vendor/hearthfire/en/messages.php` 文件。在这个文件中只需要定义你想要覆盖的句子，没有覆盖的句子仍然从该扩展包原来的语言文件中加载。

快速入门：JavaScript & CSS 脚手架

简介

Laravel 并不强制你使用什么 JavaScript 框架或者 CSS 预处理器，不过也确实提供了对很多应用而言都很有用的基本脚手架。默认情况下，Laravel 使用 NPM 来安装这些前端包。

CSS

Laravel Mix 提供了干净、优雅的 API 用于编译 SASS 或 Less，SASS 和 Less 都是在原生 CSS 的基础上新增了变量、混合（MixIn）以及其它强大的功能特性，从而让我们在使用 CSS 的时候更加享受。

在本文档中，我们会简要讨论 CSS 的编译，不过，你最好参考完整的 [Laravel Mix 文档](#) 了解更多 SASS 或 Less 的编译细节。

JavaScript

Laravel 并不强制你使用指定的 JavaScript 框架或库来构建应用，事实上，你也可以完全不使用 JavaScript，不过，Laravel 还是引入了一些基本的脚手架：使用 Vue 库让我们更轻松地编写现代 JavaScript。Vue 提供了优雅的 API 让我们可以通过组件构建强大的 JavaScript 应用。和 CSS 一样，我们可以使用 Laravel Mix 轻松将多个 JavaScript 组件编译到单个 JavaScript 文件。

移除前端脚手架代码

如果你想要从应用中移除前端脚手架代码，可以使用 `preset` 命令，该命令和 `none` 选项一起使用的时候，会从应用中移除 Bootstrap 和 Vue 脚手架代码，只留下空的 SASS 文件和一些通用的 JavaScript 实用库：

```
php artisan preset none
```

编写 CSS

Laravel 应用根目录下的 `package.json` 文件包含了 `bootstrap` 扩展包以便我们使用 Bootstrap 构建前端原型，不过，你也可以按照自己应用的需要来增删 `package.json` 文件中的扩展包。此外，并不是必须要使用 Bootstrap 框架来构建 Laravel 应用——这只是为选择使用 Bootstrap 的开发者提供一个良好的起点。

编译 CSS 之前，使用 NPM 安装应用的前端依赖（在此之前确保系统已经安装过 Node.js）：

```
npm install
```

使用 `npm install` 安装好前端依赖之后，可以使用 Laravel Mix 将 SASS 文件编译为纯 CSS，`npm run dev` 命令会处理 `webpack.mix.js` 文件中的声明。通常，编译好的 CSS 文件会放到 `public/css` 目录下：

```
npm run dev
```

Laravel 自带的默认 `webpack.mix.js` 文件会编译 SASS 文件 `resources/assets/sass/app.scss`，这个 `app.scss` 文件将会导入一个包含 SASS 变量的文件并加载 Bootstrap，从而助力我们快速在应用中引入 Bootstrap 资源。你也可以按照自己的需要自定义 `app.scss` 文件，甚至可以通过配置 Laravel Mix 使用一个完全不同的预处理器。

编写 JavaScript

应用所需的所有 JavaScript 依赖都可以在应用根目录下的 `package.json` 中找到，这个文件和 `composer.json` 类似，只不过它指定的是 JavaScript 依赖而不是 PHP 依赖。你可以使用 NPM 来安装这些依赖：

```
npm install
```

注：默认情况下，Laravel 自带的 `package.json` 文件引入了一些扩展包，比如 `vue` 和 `axios`，以便快速构建 JavaScript 应用，同样，你可以按照应用的需要增删 `package.json` 中的扩展包。

扩展包安装好之后，可以使用 `npm run dev` 命令来编译前端资源，Webpack 是为现代 JavaScript 应用提供的模块捆绑器，当你执行 `npm run dev` 命令的时候，Webpack 将会执行 `webpack.mix.js` 中的指令：

```
npm run dev
```

默认情况下，Laravel 自带的 `webpack.mix.js` 将会编译 SASS 和 `resources/assets/js/app.js` 文件，在 `app.js` 文件中你可以注册 Vue 组件，或者如果你倾向于其它 JavaScript 框架，则可以配置你自己的 JavaScript 应用。编译好的 JavaScript 文件通常会存放到 `public/js` 目录下。

注：`app.js` 文件会加载 `resources/assets/js/bootstrap.js` 以便启动和配置 Vue，Axios，jQuery 以及所有其它 JavaScript 依赖，如果你有额外的 JavaScript 依赖需要配置，请在这里操作。

编写 Vue 组件

默认情况下，新安装的 Laravel 应用将会在 `resources/assets/js/components` 目录下包含一个 Vue 组件 `ExampleComponent.vue`，这个 Vue 组件是一个单文件 Vue 组件示例，其中定义了相关的 JavaScript 和 HTML 模板，单文件组件为构建 JavaScript 驱动的应用提供了便利。这个示例组件在 `app.js` 中注册：

```
Vue.component(
  'example',
  require('./components/ExampleComponent.vue')
);
```

要在应用中使用这个组件，只需要将其丢到某个 HTML 模板中。例如，在运行完 Artisan 命令 `make:auth` 创建登录和注册视图之后，就可以将这个组件丢到 Blade 模板 `home.blade.php` 中：

```
@extends('layouts.app')

@section('content')
<example-component></example-component>
```

```
@endsection
```

注：记住，每次修改 Vue 组件后都要运行一次 `npm run dev` 命令，或者，你也可以运行 `npm run watch` 命令进行监听，一旦组件被修改后可以自动进行重新编译。

如果你对编写 Vue 组件感兴趣，可以去阅读 [Vue 文档](#)，从而对 Vue 框架有更加全面的认识。

使用 React

如果你更喜欢使用 React 来构建 JavaScript 应用，在 Laravel 中从 Vue 脚手架切换到 React 脚手架也很简单，在所有新安装的 Laravel 应用中，使用带 `react` 选项的 `preset` 命令即可：

```
php artisan preset react
```

这个命令将会移除 Vue 脚手架代码并将其替换为 React 脚手架代码，同时包含一个示例组件。

使用进阶：通过 Laravel Mix 编译前端资源

简介

Laravel Mix 提供了一套流式 API，使用一些通用的 CSS 和 JavaScript 预处理器为 Laravel 应用定义 Webpack 构建步骤。通过简单的方法链，你可以流式定义资源管道。例如：

```
mix.js('resources/assets/js/app.js', 'public/js')
    .sass('resources/assets/sass/app.scss', 'public/css');
```

如果你对如何开始使用 Webpack 和前端资源编译感到困扰，那么你会爱上 Laravel Mix。不过，并不是强制要求在开发期间使用它。你可以自由选择使用任何前端资源管道工具，或者压根不使用。

安装 & 设置

安装 Node

在开始接触 Mix 之前，必须首先确保 Node.js 和 NPM 在机器上已经安装：

```
node -v
npm -v
```

默认情况下，Laravel Homestead 已经包含了你所需要的一切；不过，如果你没有使用 Homestead，你也可以从 [Node 的下载页面](#) 轻松的下载安装最新版本的 Node 和 NPM。

Laravel Mix

接下来，需要安装 Laravel Mix，在新安装的 Laravel 根目录下，你会发现有一个 `package.json` 文件。该文件包含你所需要的一切，和 `composer.json` 类似，只不过是用来定义 Node 依赖而非 PHP 依赖，你可以通过运行如下命令来安装需要的依赖：

```
npm install
```

如果你正在 Windows 系统上开发，需要在运行 `npm install` 命令时带上 `--no-bin-links`：

```
npm install --no-bin-links
```

运行 Mix

Mix 是位于 Webpack 顶层的配置层，所以要运行 Mix 任务你只需要在运行包含在默认 `package.json` 文件中的其中某个 NPM 脚本即可：

```
// 运行所有 Mix 任务...
npm run dev
```

```
// 运行所有 Mix 任务并减少输出...
npm run production
```

监控前端资源改变

`npm run watch` 命令将会持续在终端运行并监听所有相关文件的修改，Webpack 将会在发现修改后自动重新编译资源文件：

```
npm run watch
```

你可能会发现文件变更的时候特定环境的 Webpack 不会更新，如果你遇到了这样的问题，可以考虑使用 `watch-poll` 命令：

```
npm run watch-poll
```

处理样式表

`webpack.mix.js` 是所有资源编译的入口，可以将其看作 Webpack 的轻量级配置封装层。Mix 任务可以以方法链的方式被链在一起定义前端资源如何被编译。

Less

要将 Less 编译成 CSS，可以使用 `less` 方法。下面让我们来编译 `app.less` 文件到 `public/css/app.css`:

```
mix.less('resources/assets/less/app.less', 'public/css');
```

多次调用 `less` 方法可用于编译多个文件:

```
mix.less('resources/assets/less/app.less', 'public/css')
    .less('resources/assets/less/admin.less', 'public/css');
```

如果你想要自定义编译后文件的输出位置，可以将完整的路径信息作为第二个参数传递到 `less` 方法:

```
mix.less('resources/assets/less/app.less', 'public/stylesheets/styles.css');
```

如果你需要覆盖底层 Less 插件选项，可以传递一个对象作为 `mix.less()` 的第三个参数:

```
mix.less('resources/assets/less/app.less', 'public/css', {
    strictMath: true
});
```

Sass

`sass` 方法允许你将 Sass 编译成 CSS。你可以像这样使用该方法:

```
mix.sass('resources/assets/sass/app.scss', 'public/css');
```

同样，和 `less` 方法一样，你可以将多个 Sass 文件编译成单个 CSS 文件，甚至自定义结果 CSS 的输出路径:

```
mix.sass('resources/assets/sass/app.sass', 'public/css')
    .sass('resources/assets/sass/admin.sass', 'public/css/admin');
```

额外的 Node-Sass 插件选项可以以第三个参数的形式提供:

```
mix.sass('resources/assets/sass/app.sass', 'public/css', {
    precision: 5
});
```

Stylus

和 Less 和 Sass 类似，`stylus` 方法允许你将 Stylus 编译成 CSS:

```
mix.stylus('resources/assets/stylus/app.styl', 'public/css');
```

你还可以安装额外的 Stylus 插件，例如 `Rupture`，首先，通过 NPM 安装这个插件 (`npm install rupture`) 然后在调用 `mix.stylus()` 时引入它:

```
mix.stylus('resources/assets/stylus/app.styl', 'public/css', {
    use: [
        require('rupture')()
    ]
});
```

PostCSS

`PostCSS`，是一个转化 CSS 的强大工具，在 Laravel Mix 中开箱可用。默认情况下，Mix 使用了流行的 `Autoprefixer` 插件来自动添加所需要的 CSS3 浏览器引擎前缀。不过，你也可以添加与应用适配的其他额外插件。首先，通过 NPM 安装需要的插件，然后在 `webpack.mix.js` 文件中引用:

```
mix.sass('resources/assets/sass/app.scss', 'public/css')
    .options({
        postCss: [
            require('postcss-css-variables')()
        ]
    });
});
```

原生 CSS

如果你只想要将多个原生 CSS 样式文件合并到一个文件，可以使用 `styles` 方法:

```
mix.styles([
    'public/css/vendor/normalize.css',
    'public/css/vendor/videojs.css'
], 'public/css/all.css');
```

URL 处理

因为 Laravel Mix 是基于 Webpack 开发的，所以了解一点关于 Webpack 的概念很重要。对 CSS 编译而言，Webpack 会在样式表中重写并优化所有 `url()` 调用，虽然这可能最初听上去很奇怪，但这确实个不折不扣的强大功能。假设我们想要编译包含图片相对 URL 的 Sass:

```
.example {
    background: url('../images/example.png');
}
```

注: 任意给定 `url()` 的绝对路径都会从 URL 重写中排除, 例如, `url('/images/thing.png')` 或 `url('http://example.com/images/thing.png')` 将不会被修改。默认情况下, Laravel Mix 和 Webpack 会找到 `example.png`, 将其拷贝到 `public/images` 目录下, 然后在生成的样式表中重写 `url()`, 因此, 编译后的 CSS 如下所示:

```
.example {
    background: url(/images/example.png?d41d8cd98f00b204e9800998ecf8427e);
```

和这个功能一样有用的是, 可能已存在的目录结构已经配置成你想要的方式, 这种情况下, 你可以禁用 `url()` 重写:

```
mix.sass('resources/assets/app/app.scss', 'public/css')
.options({
    processCssUrls: false
});
```

如果添加了这项配置到 `webpack.mix.js` 文件, Mix 将不再匹配 `url()` 或拷贝资源到 `public` 目录。换句话说, 编译过的 CSS 和编译前输入的一样:

```
.example {
    background: url("../images/thing.png");
}
```

Source Map

虽然 Source Map 默认被禁用, 但是可以通过在 `webpack.mix.js` 文件中调用 `mix.sourceMaps()` 来激活。尽管这会带来编译/性能开销, 不过在编译资源的时候可以提供额外的调试信息给浏览器的开发者工具:

```
mix.js('resources/assets/js/app.js', 'public/js')
    .sourceMaps();
```

处理 JavaScript

Mix 还提供了多个特性帮助你处理 JavaScript 文件, 例如编译 ECMAScript 2015, 模块捆绑, 最小化以及合并原生 JavaScript 文件。更妙的是, 这些都是无缝集成的, 不需要额外的自定义配置:

```
mix.js('resources/assets/js/app.js', 'public/js');
```

通过这一行代码, 你可以使用如下功能:

- ES2015 语法
- 模块
- 编译 `.vue` 文件
- 最小化生产环境

提取 Vendor 库

捆绑所有应用特定 JavaScript 和 vendor 库的一个潜在缺点是进行长期缓存将变得更加困难, 例如, 单次更新应用代码将会强制浏览器下载所有 vendor 库, 即使它们并没有更新。

如果你想要频繁更新应用的 JavaScript, 需要考虑对 vendor 库进行提取和拆分, 这样的话, 对应用代码的一个修改不会影响 `vendor.js` 文件的缓存。Mix 的 `extract` 方法可以实现这样的功能:

```
mix.js('resources/assets/js/app.js', 'public/js')
    .extract(['vue']);
```

`extract` 方法接收包含所有库的数组或你想要提取到 `vendor.js` 文件的模块, 使用上述代码作为示例, Mix 将会生成如下文件:

- `public/js/manifest.js` : Webpack manifest runtime
- `public/js/vendor.js` : vendor 库
- `public/js/app.js` : 应用代码

要避免 JavaScript 错误, 确保以正确顺序加载这些文件:

```
<script src="/js/manifest.js"></script>
<script src="/js/vendor.js"></script>
<script src="/js/app.js"></script>
```

React

Mix 可以自动为安装 Babel 插件以便支持 React, 我们可以将 `mix.js()` 调用替换为 `mix.react()` 来实现:

```
mix.react('resources/assets/js/app.jsx', 'public/js');
```

在这个场景背后, Mix 会下载并引入合适的 Babel 插件 `babel-preset-react`。

Vanilla JS

和使用 `mix.styles()` 合并样式表类似，你可以通过 `scripts()` 方法合并并最小化任意数量的 JavaScript 文件：

```
mix.scripts([
    'public/js/admin.js',
    'public/js/dashboard.js'
], 'public/js/all.js');
```

这一功能对那些不需要 Webpack 对 Javascript 进行编译的传统应用来说很有用。

注：`mix.scripts()` 的一个轻微调整是 `mix.babel()`，它的方法签名和 `scripts` 一样，不同之处是合并的文件会经过 Babel 编译，从而将所有 ES2015 代码转化成所有浏览器都支持的原生 JavaScript。

自定义 Webpack 配置

在场景背后，Laravel Mix 引用了预配置的 `webpack.config.js` 文件来尽可能快的启动和运行。个别情况下，你需要手动编辑这个文件。你可能有一个被引用的特定的加载器或插件，或者可能倾向于使用 Stylus 而不是 Sass，在这些情况下，你有两个选择：

合并自定义配置

Mix 提供了一个有用的 `webpackConfig` 方法，从而允许你合并任意简短的 Webpack 配置覆盖。这是一个很吸引人的选择，因为不需要你拷贝或维护自己的 `webpack.config.js` 文件副本，`webpackConfig` 方法接收一个对象，该对象包含了任意你想要应用的 [Webpack 指定配置](#)：

```
mix.webpackConfig({
    resolve: {
        modules: [
            path.resolve(__dirname, 'vendor/laravel/spark/resources/assets/js')
        ]
    }
});
```

自定义配置文件

第二个选择是拷贝 Mix 的 `webpack.config.js` 到自己的项目根目录：

```
cp node_modules/laravel-mix/setup/webpack.config.js ./
```

接下来，将 `package.json` 文件中的所有 `--config` 引用指向拷贝后的新配置文件。如果你选择使用这种自定义方式，以后只要 Mix 的 `webpack.config.js` 有升级变更都要手动将变更合并到自定义的新文件。

拷贝文件/目录

你可以使用 `copy` 方法拷贝文件/目录到新路径，这在将 `node_modules` 目录下的特定资源文件重新放置到 `public` 目录下时很有用：

```
mix.copy('node_modules/foo/bar.css', 'public/css/bar.css');
```

拷贝目录的时候，`copy` 方法将会铺平目录结构，要维持目录的原始结构，需要使用 `copyDirectory` 方法：

```
mix.copyDirectory('assets/img', 'public/img');
```

版本号/缓存刷新

很多开发者会给编译的前端资源添加时间戳或者唯一令牌后缀以强制浏览器加载最新版本而不是代码的缓存副本。Mix 可以使用 `version` 方法为你处理这种场景。

`version` 方法会自动附加唯一哈希到已编译文件名，从而方便实现缓存刷新：

```
mix.js('resources/assets/js/app.js', 'public/js')
    .version();
```

生成版本文件后，还不知道提取的文件名，所以，你需要在 [视图](#) 中使用 Laravel 全局的 `mix` 函数来加载相应的带哈希值的前端资源。`mix` 函数会自动判当前的已哈希文件名：

```
<link rel="stylesheet" href="{{ mix('css/app.css') }}>
```

由于版本文件在本地开发中没有什么用，你可以只在运行 `npm run production` 期间进行版本处理操作：

```
mix.js('resources/assets/js/app.js', 'public/js');

if (mix.config.inProduction) {
    mix.version();
}
```

BrowserSync 重新加载

`BrowserSync` 会自动监控文件修改，并将修改注入浏览器而不需要手动刷新，你可以通过调用 `mix.browserSync()` 方法启用该支持：

```
mix.browserSync('my-domain.test');

// Or...

// https://browsersync.io/docs/options
```

```
mix.browserSync({
  proxy: 'my-domain.test'
});
```

你可以传递一个字符串（代理）或对象（BrowserSync 设置）到该方法。接下来，使用 `npm run watch` 命令来启动 Webpack 的开发服务器，现在，当你编辑一个 JavaScript 脚本或 PHP 文件时，会看到浏览器会立即刷新以响应你的修改。

环境变量

你可以通过在 `.env` 文文件添加 `MIX_` 前缀将环境变量注入 Mix:

```
MIX_SENTRY_DSN_PUBLIC=http://example.com
```

在 `.env` 文件中定义好变量之后，可以通过 `process.env` 对象进行访问（如果在运行 `watch` 任务期间变量值有变动，需要重启任务）：

```
process.env.MIX_SENTRY_DSN_PUBLIC
```

通知

在有效的情况下，Mix 会自动为每个捆绑显示操作系统通知，这可以给你一个及时的反馈：编译成功还是失败。不过，某些场景下你可能希望禁止这些通知，一个典型的例子就是在生产境服务器触发 Mix。通知可以通过 `disableNotifications` 方法被停用：

```
mix.disableNotifications();
```

六、数据库操作

快速入门

简介

Laravel 让连接不同数据库以及对数据库进行增删改查操作变得非常简单，不论使用原生 SQL、还是 [查询构建器](#)，还是 [Eloquent ORM](#)。目前，Laravel 支持四种类型的数据库系统：

- MySQL
- Postgres
- SQLite
- SQL Server

配置

应用的数据库配置位于 `config/database.php`（但是数据库用户及密码等敏感信息位于 `.env` 文件，如果你还不知道 `.env` 是何方神圣，那么你可能需要到这里补补课：http://laravelacademy.org/post/8650.html#toc_5）。在该文件中你可以定义所有的数据库连接，并指定哪个连接是默认连接。该文件中提供了所有支持数据库系统的配置示例。

默认情况下，Laravel 使用 MySQL 作为数据库引擎，并且示例配置已经为 Laravel Homestead 环境做好了设置（意思是说，如果你是用 Homestead 作为开发环境的话，就可以实现零配置使用），当然，你也可以按照需要为本地的数据库修改该配置（在 `.env` 中修改数据库配置）：

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret
```

SQLite 配置

在项目根目录下使用 `touch database/database.sqlite` 命令创建好新的 SQLite 数据库之后，就可以使用数据库绝对路径配置环境变量指向这个新创建的数据库：

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

注：正如其名字所标识的，SQLite 是一个轻量级的、遵守 ACID 标准的关系型数据库，它包含在一个相对小的 C 程序库中。与许多其它数据库管理系统不同，SQLite 不是一个客户端/服务器结构的数据库引擎，而是被集成在用户程序中。作为嵌入式数据库，是应用程序（如网页浏览器）在本地 / 客户端存储数据的常见选择。更多关于 SQLite 的信息请查看其官网：<http://www.sqlite.org/index.html>

读写分离

有时候你希望使用一个数据库连接做查询，另一个数据库连接做插入、更新和删除，Laravel 中实现这种读写分离非常简单，不管你用的是原生 SQL，还是查询构建器，还是 Eloquent ORM，只要配置正确，合适的连接总是会被使用。

想要知道如何配置读/写连接，可以参考下面这个例子：

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

注意我们在配置数组中新增了三个键：`read`、`write` 和 `sticky`，`read` 和 `write` 这两个键对应值都有一个包含单个键“host”的数组，而其所映射的 IP 值分别就是读连接和写连接，读/写连接的其它数据库配置项都共用 `mysql` 的主数组配置。

如果我们想要覆盖主数组中的配置，只需要将相应配置项放到 `read` 和 `write` 数组中即可。在本例中，`192.168.1.1` 将被用作“读”连接，而 `196.168.1.2` 将被用作“写”连接。两个数据库连接的凭证（用户名/密码）、前缀、字符集以及其它配置将会共享 `mysql` 数组中的设置，同理，如果不一样的话，分别在 `read` 或 `write` 数组中单独配置即可。

对于大部分应用来说都是读多写少，所以面对这种情况，如何配置多个读连接，一个写连接？可以这么做：

```
'mysql' => [
    'driver' => 'mysql',
    'read' => [
        'host' => ['193.168.1.1', '194.168.1.1']
    ],
    'write' => [
        'host' => '196.168.1.2'
    ],
    //
]
```

Laravel 在读数据时会从提供的 IP 中随机选一个进行连接。实现原理感兴趣的同学可以查看 [Illuminate\Database\Connectors\ConnectionFactory](#) 底层源码。

注：目前读写分离仅支持单个写连接。

sticky 项

`sticky` 项是一个可选的配置值，可用于在当前请求生命周期内允许立即读取写入数据库的记录。如果 `sticky` 选项被启用并且一个“写”操作在当前生命周期内发生，则后续所有“读”操作都会使用这个“写”连接（前提是同一个请求生命周期内），这样就可以确保同一个请求生命周期内写入的数据都可以立即被读取到，从而避免主从延迟导致的数据不一致，是否启用这一功能取决于你。

学院君注：当然，这只是一个针对分布式数据库系统中主从数据同步延迟的一个非常初级的解决方案，访问量不高的中小网站可以这么做，大流量高并发网站肯定不能这么干，主从读写分离本来就是为了解决单点性能问题，这样其实是把问题又引回去了，造成所有读写都集中到写数据库，对于高并发频繁写的场景下，后果可能是不堪设想的，但是话说回来，对于并发量不那么高，写操作不那么频繁的中小型站点来说，`sticky` 这种方式不失为一个初级的解决方案。

使用不同数据库连接

使用多个数据库连接的时候，可以通过 `DB` 门面上的 `connection` 方法访问不同连接。传递给 `connection` 方法的 `name` 对应配置文件 `config/database.php` 中设置的某个连接：

```
$users = DB::connection('read')->select(...);
```

甚至还可以指定数据库和连接名，使用 `::` 分隔：

```
$users = DB::connection('mysql::read')->select(...);
```

你还可以使用连接实例上的 `getPdo` 方法访问底层原生的 PDO 实例：

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

运行原生 SQL 查询

配置好数据库连接后，就可以使用 `DB` 门面来运行查询。`DB` 门面为每种操作提供了相应方法：`select`、`update`、`insert`、`delete` 和 `statement`。

运行 Select 查询

运行一个最基本的查询，可以使用 `DB` 门面的 `select` 方法：

```
<?php
```

```

namespace App\Http\Controllers;

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

class UserController extends Controller
{
    /**
     * 展示应用的用户列表.
     *
     * @return Response
     */
    public function index()
    {
        $users = DB::select('select * from users where active = ?', [1]);

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

```

传递给 `select` 方法的第一个参数是原生的 SQL 语句，第二个参数需要绑定到查询的参数绑定，通常，这些都是 `where` 子句约束中的值。参数绑定可以避免 SQL 注入攻击（输入参数校验由实现方控制，用户无法传递任意查询参数）。

`select` 方法以数组的形式返回结果集，数组中的每一个结果都是一个 PHP `stdClass` 对象：

```

array:1 [▼
  0 => {#350 ▼
    +"id": 1
    +"name": "jroJoGP71W"
    +"email": "Syin7oiogH@laravelacademy.org"
    +"password": "$2y$10$WX14KWPwQHrUI2Y.8ZMeKeROjyZV0GNJ0kDM7ATjNJI/dWxmy1A9e"
    +"remember_token": null
    +"created_at": null
    +"updated_at": null
  }
]

```

你可以像下面这样访问结果值：

```

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

```

使用命名绑定

除了使用 `?` 占位符来代表参数绑定外，还可以使用命名绑定来执行查询：

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

运行插入语句

使用 `DB` 门面的 `insert` 方法执行插入语句。和 `select` 一样，该方法将原生 SQL 语句作为第一个参数，将参数绑定作为第二个参数：

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

运行更新语句

`update` 方法用于更新数据库中已存在的记录，该方法返回受更新语句影响的行数：

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

运行删除语句

`delete` 方法用于删除数据库中已存在的记录，和 `update` 一样，该语句返回被删除的行数：

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

学院君注：使用 `delete` 和 `update` 语句时，需要非常小心，因为条件设置不慎，导致的后果有可能是无法挽回的，比如不带条件的 `delete` 语句删除的将是数据表的所有记录！这些都是有血淋淋的教训的。

运行一个通用语句

有些数据库语句不返回任何值，比如新增表，修改表，删除表等，对于这种类型的操作，可以使用 `DB` 门面的 `statement` 方法：

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

监听查询事件

如果你想要获取应用中每次 SQL 语句的执行，可以使用 `listen` 方法，该方法对查询日志和调试非常有用，你可以在 [服务提供者](#) 中注册查询监听器：

```
<?php
```

```

namespace App\Providers;

use Illuminate\Support\Facades\DB;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        DB::listen(function ($query) {
            // $query->sql
            // $query->bindings
            // $query->time
        });
    }

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}

```

数据库事务

想要在一个数据库事务中运行一连串操作，可以使用 `DB` 门面的 `transaction` 方法，使用 `transaction` 方法时不需要手动回滚或提交：如果事务闭包中抛出异常，事务将会自动回滚；如果闭包执行成功，事务将会自动提交：

```

DB::transaction(function () {
    DB::table('users')->update(['votes' => 1]);
    DB::table('posts')->delete();
});

```

处理死锁

数据库死锁指的是有两个或两个以上数据库操作相互依赖，一方需要等待另一方退出才能获取资源，但是没有一方提前退出，就会造成死锁，数据库事务容易造成的一个副作用就是死锁。为此 `transaction` 方法接收一个可选参数作为第二个参数，用于定义死锁发生时事务的最大重试次数。如果尝试次数超出指定值，会抛出异常：

```

DB::transaction(function () {
    DB::table('users')->update(['votes' => 1]);
    DB::table('posts')->delete();
}, 5);

```

手动使用事务

如果你想要手动开启事务从而对回滚和提交有更好的控制，可以使用 `DB` 门面的 `beginTransaction` 方法：

```
DB::beginTransaction();
```

你可以通过 `rollBack` 方法回滚事务：

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

最后，你可以通过 `commit` 方法提交事务：

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

注：使用 `DB` 门面的事务方法还可以用于控制 `查询构建器` 和 `Eloquent ORM` 的事务。关于这两块内容，我们马上就会看到。

查询构建器

简介

数据库查询构建器提供了一个方便的流接口用于创建和执行数据库查询。查询构建器可以用于执行应用中绝大部分数据库操作，并且能够在 Laravel 支持的所有数据库系统上工作。

注：流接口是一种设计模式，更多关于流接口模式的设计和使用方式，可查看这篇教程：[PHP 设计模式系列 —— 流接口模式](#)。

Laravel 查询构建器使用 PDO 参数绑定来避免 SQL 注入攻击，不再需要过滤以绑定方式传递的字符串。

获取结果集

从一张表中取出所有行

我们可以从 `DB` 门面的 `table` 方法开始，`table` 方法为给定表返回一个流式查询构建器实例，该实例允许你在查询上链接多个约束条件并返回最终查询结果。在本例中，我们使用 `get` 方法获取表中所有记录：

```
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller
{
    /**
     * 显示用户列表
     *
     * @return Response
     */
    public function index()
    {
        $users = DB::table('users')->get();
        return view('user.index', ['users' => $users]);
    }
}
```

`get` 方法返回包含结果集的 `Illuminate\Support\Collection`，其中每一个结果都是 PHP 的 `StdClass` 对象实例：

```
Collection {#350 ▼
#items: array:1 [▼
  0 => #351 ▼
    +"id": 1
    +"name": "jroJoGP71W"
    +"email": "Syin7oiogH@laravelacademy.org"
    +"password": "$2y$10$WX14KWPwQHrUI2Y.8ZMeKeROjyZV0GNJ0kDM7ATjNJI/dWxmy1A9e"
    +"remember_token": null
    +"created_at": null
    +"updated_at": null
  ]
}
```

你可以像访问对象的属性一样访问字段的值：

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

从一张表中获取一行/一列

如果你只是想要从数据表中获取一行数据，可以使用 `first` 方法，该方法将会返回单个 `StdClass` 对象：

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

如果你不需要完整的一行，可以使用 `value` 方法从结果中获取单个值，该方法会直接返回指定列的值：

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

获取数据列值列表

如果想要获取包含单个列值的数组，可以使用 `pluck` 方法，在本例中，我们获取角色标题数组：

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

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

还可以在返回数组中为列值指定自定义键（该自定义键必须是该表的其它字段列名，否则会报错）：

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

foreach ($roles as $name => $title) {
```

```
echo $title;
}
```

组块结果集

如果你需要处理成千上百条数据库记录，可以考虑使用 `chunk` 方法，该方法一次获取结果集的一小块，然后传递每一小块数据到闭包函数进行处理，该方法在编写处理大量数据库记录的 `Artisan 命令` 的时候非常有用。例如，我们可以将处理全部 `users` 表数据分割成一次处理 100 条记录的小组块：

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

你可以通过从闭包函数中返回 `false` 来终止组块的运行：

```
DB::table('users')->orderBy('id')->chunk(100, function($users) {
    // 处理结果集...
    return false;
});
```

聚合函数

查询构建器还提供了多个聚合方法，如 `count`, `max`, `min`, `avg` 和 `sum`，你可以在构造查询之后调用这些方法：

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

当然，你可以联合其它查询子句和聚合函数来构建查询：

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

判断记录是否存在

除了通过 `count` 方法来判断匹配查询条件的结果是否存在外，还可以使用 `exists` 或 `doesntExist` 方法：

```
return DB::table('orders')->where('finalized', 1)->exists();
return DB::table('orders')->where('finalized', 1)->doesntExist();
```

查询 (Select)

指定查询子句

当然，我们并不总是想要获取数据表的所有列，使用 `select` 方法，你可以为查询指定自定义的 `select` 子句：

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

`distinct` 方法允许你强制查询返回不重复的结果集：

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

如果你已经有了一个查询构建器实例并且希望添加一个查询列到已存在的 `select` 子句，可以使用 `addSelect` 方法：

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

原生表达式

有时候你希望在查询中使用原生表达式，这些表达式将会以字符串的形式注入到查询中，所以要格外小心避免 SQL 注入。想要创建一个原生表达式，可以使用 `DB::raw` 方法：

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

注：原生语句会以字符串的形式注入查询，所以这里尤其要注意避免 SQL 注入攻击。

原生方法

除了使用 `DB::raw` 外，你还可以使用以下方法来插入原生表达式到查询的不同部分。

`selectRaw`

`selectRaw` 方法可用于替代 `select(DB::raw(...))`，该方法接收一个可选的绑定数组作为第二个参数：

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

`whereRaw / orWhereRaw`

`whereRaw` 和 `orWhereRaw` 方法可用于注入原生 `where` 子句到查询，这两个方法接收一个可选的绑定数组作为第二个参数：

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

`havingRaw / orHavingRaw`

`havingRaw` 和 `orHavingRaw` 方法可用于设置原生字符串作为 `having` 子句的值：

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

`orderByRaw`

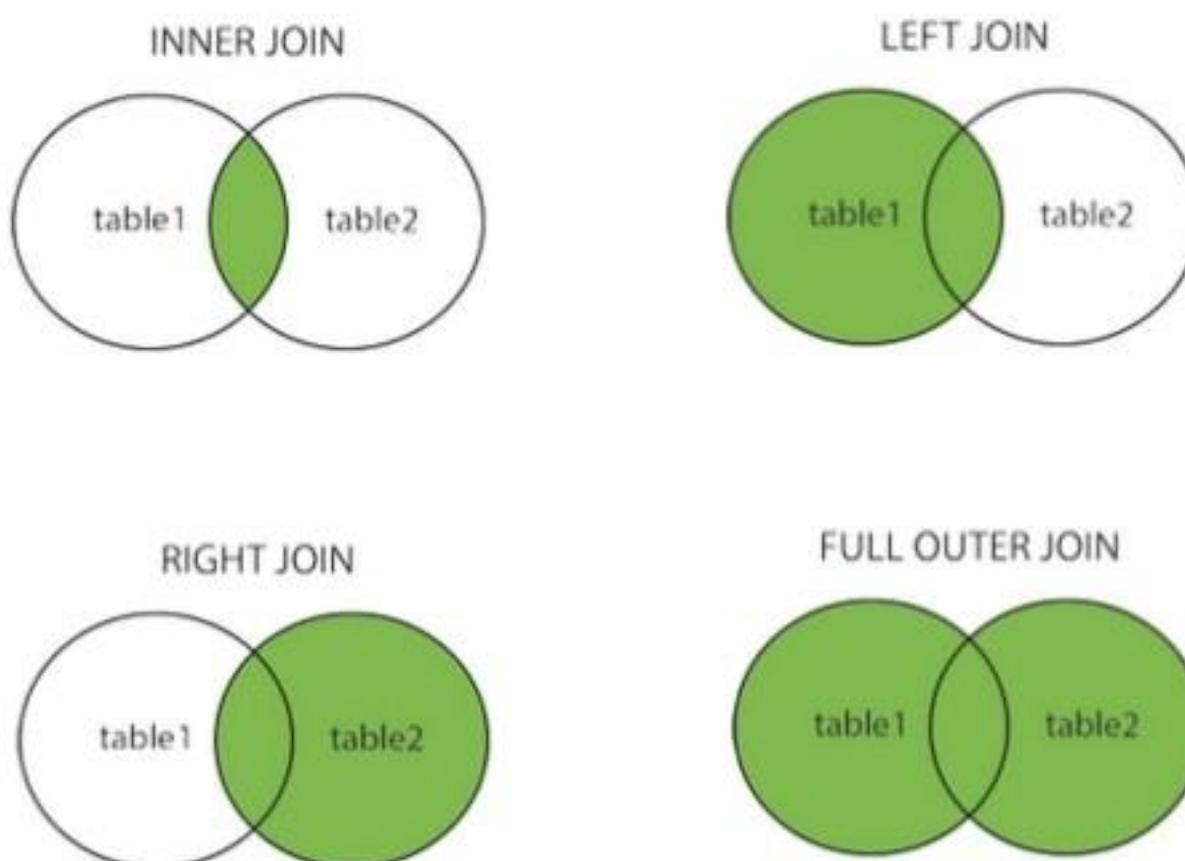
`orderByRaw` 方法可用于设置原生字符串作为 `order by` 子句的值：

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

连接 (Join)

查询构建器还可以用于编写连接语句，关于 SQL 的几种连接类型，通过下图可以一目了然：

连接



[obj]

内连接（等值连接）

要实现一个简单的“内连接”，你可以使用查询构建器实例上的 `join` 方法，传递给 `join` 方法的第一个参数是你需要连接到的表名，剩余的其它参数则是为连接指定的列约束，当然，正如你所看到的，你可以在单个查询中连接多张表：

```
$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();
```

左连接

如果你是想要执行“左连接”而不是“内连接”，可以使用 `leftJoin` 方法。该方法和 `join` 方法的用法一样：

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

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

交叉连接

要执行“交叉连接”可以使用 `crossJoin` 方法，传递你想要交叉连接的表名到该方法即可。交叉连接在第一张表和被连接表之间生成一个笛卡尔积：

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

高级连接语句

你还可以指定更多的高级连接子句，传递一个闭包到 `join` 方法作为第二个参数，该闭包将会接收一个 `JoinClause` 对象用于指定 `join` 子句约束：

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

如果你想要在连接中使用“where”风格的子句，可以在查询中使用 `where` 和 `orWhere` 方法。这些方法会将列和值进行比较而不是列和列进行比较：

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

联合 (Union)

查询构建器还提供了“联合”两个查询的快捷方式，比如，你可以先创建一个查询，然后使用 `union` 方法将其和第二个查询进行联合：

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

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

注：`unionAll` 方法也是有效的，并且和 `union` 使用方式相同。

Where 子句

简单 Where 子句

使用查询构建器上的 `where` 方法可以添加 `where` 子句到查询中，调用 `where` 最基本的方式需要传递三个参数，第一个参数是列名，第二个参数是任意一个数据库系统支持的操作符，第三个参数是该列要比较的值。

例如，下面是一个验证“votes”列的值是否等于 100 的查询：

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

为了方便，如果你只是简单比较列值和给定数值是否相等，可以将数值直接作为 `where` 方法的第二个参数：

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

当然，你还可以使用其它操作符来编写 `where` 子句：

```
$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();
```

还可以传递条件数组到 `where` 函数：

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

or 语句

你可以通过方法链将多个 `where` 约束链接到一起，也可以添加 `or` 子句到查询，`orWhere` 方法和 `where` 方法接收参数一样：

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

更多 Where 子句

`whereBetween`

`whereBetween` 方法验证列值是否在给定值之间：

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

`whereNotBetween`

`whereNotBetween` 方法验证列值不在给定值之间：

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

`whereIn`/`whereNotIn`

`whereIn` 方法验证给定列的值是否在给定数组中：

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

`whereNotIn` 方法验证给定列的值不在给定数组中：

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

`whereNull`/`whereNotNull`

`whereNull` 方法验证给定列的值为 `NULL`：

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

`whereNotNull` 方法验证给定列的值不是 `NULL`：

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

`whereDate` / `whereMonth` / `whereDay` / `whereYear` / `whereTime`

`whereDate` 方法用于比较字段值和日期：

```
$users = DB::table('users')
    ->whereDate('created_at', '2016-10-10')
    ->get();
```

`whereMonth` 方法用于比较字段值和一年中的指定月份：

```
$users = DB::table('users')
    ->whereMonth('created_at', '10')
    ->get();
```

`whereDay` 方法用于比较字段值和一月中的指定日期：

```
$users = DB::table('users')
    ->whereDay('created_at', '10')
    ->get();
```

`whereYear` 方法用于比较字段值和指定年：

```
$users = DB::table('users')
    ->whereYear('created_at', '2017')
    ->get();
```

`whereTime` 方法用于比较字段值和指定时间：

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

`whereColumn`

`whereColumn` 方法用于验证两个字段是否相等：

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

```
->whereColumn('first_name', 'last_name')
->get();
```

还可以传递一个比较运算符到该方法:

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

还可以传递多条件数组到 `whereColumn` 方法, 这些条件通过 `and` 操作符进行连接:

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

参数分组

有时候你需要创建更加高级的 `where` 子句, 比如“`where exists`”或者嵌套的参数分组。Laravel 查询构建器也可以处理这些。作为开始, 让我们看一个在括号中进行分组约束的例子:

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

正如你所看到的, 传递闭包到 `orWhere` 方法构造查询构建器来开始一个约束分组, 该闭包将会获取一个用于设置括号中包含的约束的查询构建器实例。上述语句等价于下面的 SQL:

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

where exists 子句

`whereExists` 方法允许你编写 `where exists` SQL 子句, `whereExists` 方法接收一个闭包参数, 该闭包获取一个查询构建器实例从而允许你定义放置在“exists”子句中的查询:

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

上述查询等价于下面的 SQL 语句:

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

JSON Where 子句

Laravel 还支持在提供 JSON 字段类型的数据库 (目前是 MySQL 5.7 和 PostgreSQL) 上使用操作符 `->` 获取指定 JSON 字段值:

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

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

排序、分组、限定

orderBy

`orderBy` 方法允许你通过给定字段对结果集进行排序, `orderBy` 的第一个参数应该是你希望排序的字段, 第二个参数控制着排序的方向 — `asc` 或 `desc`:

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

```
->orderBy('name', 'desc')
->get();
```

latest / oldest

`latest` 和 `oldest` 方法允许你通过日期对结果进行排序，默认情况下，结果集根据 `created_at` 字段进行排序，或者，你可以按照你想要排序的字段作为字段名传入：

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

inRandomOrder

`inRandomOrder` 方法可用于对查询结果集进行随机排序，比如，你可以用该方法获取一个随机用户：

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

groupBy / having

`groupBy` 和 `having` 方法用于对结果集进行分组，`having` 方法和 `where` 方法的用法类似：

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

关于 `having` 的更多高级用法，可查看 `havingRaw` 方法。

skip / take

想要限定查询返回的结果集的数目，或者在查询中跳过给定数目的结果，可以使用 `skip` 和 `take` 方法：

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

作为替代方法，还可以使用 `limit` 和 `offset` 方法：

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

条件子句

有时候你可能想要某些条件为 `true` 的时候才将条件子句应用到查询。例如，你可能只想给定值在请求中存在的情况下才应用 `where` 语句，这可以通过 `when` 方法实现：

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

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

`when` 方法只有在第一个参数为 `true` 的时候才执行给定闭包，如果第一个参数为 `false`，则闭包不执行。

你可以传递另一个闭包作为 `when` 方法的第三个参数，该闭包会在第一个参数为 `false` 的情况下执行。为了演示这个特性如何使用，我们来配置一个查询的默认排序：

```
$sortBy = null;

$users = DB::table('users')
->when($sortBy, function ($query) use ($sortBy) {
    return $query->orderBy($sortBy);
}, function ($query) {
    return $query->orderBy('name');
})
->get();
```

插入 (Insert)

查询构建器还提供了 `insert` 方法用于插入记录到数据表。`insert` 方法接收数组形式的字段名和字段值进行插入操作：

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

你甚至可以一次性通过传入多个数组来插入多条记录，每个数组代表要插入数据表的记录：

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

```
]);
```

自增 ID

如果数据表有自增 ID，使用 `insertGetId` 方法来插入记录并返回 ID 值：

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

注：当使用 PostgreSQL 时 `insertGetId` 方法默认自增列被命名为 `id`，如果你想要从其他“序列”获取 ID，可以将序列名作为第二个参数传递到 `insertGetId` 方法。

更新 (Update)

当然，除了插入记录到数据库，查询构建器还可以通过使用 `update` 方法更新已有记录。`update` 方法和 `insert` 方法一样，接收字段名和字段值的键值对数组，对应字段名就是要更新的列，你可以通过 `where` 子句来对 `update` 查询进行约束：

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

更新 JSON 字段

更新 JSON 字段的时候，需要使用 `->` 语法访问 JSON 对象上相应的值，该操作只能用于支持 JSON 字段类型的数据库：

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

增加/减少

查询构建器还为增减给定字段名对应数值提供方便。相较于编写 `update` 语句，这是一条捷径，提供了更好的体验和测试接口。这两个方法都至少接收一个参数：需要修改的列。第二个参数是可选的，用于控制列值增加/减少的数目。

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

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

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

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

在操作过程中你还可以指定额外的列进行更新：

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

删除 (Delete)

当然，查询构建器还可以通过 `delete` 方法从表中删除记录，你可以在调用 `delete` 方法前通过添加 `where` 子句来添加约束条件：

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

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

如果你希望清除整张表，也就是删除所有行并将自增 ID 置为 0，可以使用 `truncate` 方法：

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

悲观锁 & 乐观锁

悲观锁（Pessimistic Lock），顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁、表锁、读锁、写锁等，都是在做操作之前先上锁。

乐观锁（Optimistic Lock），顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制实现。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于 `write_condition` 机制的其实都是提供的乐观锁。

下面我们看下悲观锁和乐观锁在 Laravel 中的使用：

悲观锁使用

Laravel 查询构建器提供了一些方法帮助你在 `select` 语句中实现“悲观锁”。可以在查询中使用 `sharedLock` 方法从而在运行语句时带一把“共享锁”。共享锁可以避免被选择的行被修改直到事务提交：

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

上面这个查询等价于下面这条 SQL 语句：

```
select * from `users` where `votes` > '100' lock in share mode
```

此外你还可以使用 `lockForUpdate` 方法。“for update”锁避免选择行被其它共享锁修改或删除:

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

上面这个查询等价于下面这条 SQL 语句:

```
select * from `users` where `votes` > '100' for update
```

`for update` 与 `lock in share mode` 都是用于确保被选中的记录值不能被其它事务更新（上锁），两者的区别在于 `lock in share mode` 不会阻塞其它事务读取被锁定行记录的值，而 `for update` 会阻塞其他锁定性读对锁定行的读取（非锁定性读仍然可以读取这些记录，`lock in share mode` 和 `for update` 都是锁定性读）。

这么说比较抽象，我们举个计数器的例子：在一条语句中读取一个值，然后在另一条语句中更新这个值。使用 `lock in share mode` 的话可以允许两个事务读取相同的初始化值，所以执行两个事务之后最终计数器的值+1；而如果使用 `for update` 的话，会锁定第二个事务对记录值的读取直到第一个事务执行完成，这样计数器的最终结果就是+2 了。

乐观锁使用

乐观锁，大多是基于数据版本（Version）记录机制实现。何谓数据版本？即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个“version”字段来实现。

读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提交数据的版本数据与数据库表对应记录的当前版本信息进行比对，如果提交的数据版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。

总结

两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下，即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果经常产生冲突，上层应用会不断的进行重试，这样反倒是降低了性能，所以这种情况下用悲观锁就比较合适。

分页

简介

在其他框架中，分页可能是件非常痛苦的事，Laravel 让这件事变得简单、易于上手。Laravel 的分页器与[查询构建器](#)和[Eloquent ORM](#)集成在一起，并开箱提供方便的、易于使用的、基于数据库结果集的分页。分页器生成的 HTML 兼容 [Bootstrap CSS](#) 框架。

基本使用

基于查询构建器进行分页

有多种方式实现分页功能，最简单的方式就是使用查询构建器或 Eloquent 查询提供的 `paginate` 方法。该方法基于当前用户查看页自动设置合适的偏移（`offset`）和限制（`limit`），直白点说就是页码和每页显示数量。默认情况下，当前页通过 HTTP 请求查询字符串参数 `page` 的值判断。当然，该值由 Laravel 自动检测，然后自动插入分页器生成的链接中。

让我们先来看看如何在查询中调用 `paginate` 方法。在本例中，传递给 `paginate` 的唯一参数就是你每页想要显示的数目，这里我们指定每页显示 15 个：

```
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller
{
    /**
     * 显示应用中的所有用户
     *
     * @return Response
     */
    public function index()
    {
        $users = DB::table('users')->paginate(15);
        return view('user.index', ['users' => $users]);
    }
}
```

注：目前，使用 `groupBy` 的分页操作不能被 Laravel 有效执行，如果你需要在分页结果中使用 `groupBy`，推荐你手动查询数据库然后创建分页器。
简单分页

如果你只需要在分页视图中简单的显示“下一页”和“上一页”链接，可以使用 `simplePaginate` 方法来执行一个更加高效的查询。在渲染包含大数据集的视图且不需要显示每个页码时这一功能非常有用：

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

基于 Eloquent 结果集进行分页

你还可以对 Eloquent 查询结果进行分页，在本例中，我们对 `User` 模型进行分页，每页显示 15 条记录。正如你所看到的，该语法和基于查询构建器的分页差不多：

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

当然，你可以在设置其它约束条件之后调用 `paginate`，比如 `where` 子句：

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

在对 Eloquent 模型进行分页时你也可以使用 `simplePaginate` 方法：

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

手动创建分页器

有时候你可能想要通过传递数组数据来手动创建分页实例，你可以基于自己的需求通过创

建 `Illuminate\Pagination\Paginator` 或 `Illuminate\Pagination\LengthAwarePaginator` 实例来实现。

`Paginator` 类不需要知道结果集中数据项的总数；不过，正因如此，该类也没有提供获取最后一页索引的方法。`LengthAwarePaginator` 接收参数和 `Paginator` 几乎一样，唯一不同在于它要求传入结果集的总数。

换句话说，`Paginator` 对应 `simplePaginate` 方法，而 `LengthAwarePaginator` 对应 `paginate` 方法。

注：当手动创建分页器实例的时候，应该手动对传递到分页器的结果集进行“切片”，如果你不确定怎么做，查看 PHP 函数 `array_slice`。

显示分页结果

当调用 `paginate` 方法时，你将获取 `Illuminate\Pagination\LengthAwarePaginator` 实例，而调用方法 `simplePaginate` 时，将会获取 `Illuminate\Pagination\Paginator` 实例。这些对象提供相关方法描述这些结果集，除了这些辅助函数外，分页器实例本身就是迭代器，可以像数组一样对其进行循环调用。所以，获取到结果后，可以按如下方式使用 Blade 显示这些结果并渲染页面链接：

```
<div class="container">
    @foreach ($users as $user)
        {{ $user->name }}
    @endforeach
</div>

{{ $users->links() }}
```

`links` 方法将会将结果集中的其它页面链接渲染出来。每个链接已经包含了 `page` 查询字符串变量。记住，`render` 方法生成的 HTML 兼容 Bootstrap CSS 框架。

自定义分页链接

`withPath` 方法允许你生成分页链接时自定义分页器使用的 URI，例如，如果你想要分页器生成形如 `http://example.com/custom/url?page=N` 的链接，应该传递 `custom/url` 到 `withPath` 方法：

```
Route::get('users', function () {
    $users = App\User::paginate(15);
    $users->withPath('custom/url');
    //
});
```

添加参数到分页链接

你可以使用 `appends` 方法添加查询参数到分页链接查询字符串。例如，要添加 `&sort=votes` 到每个分页链接，应该像如下方式调用 `appends`：

```
{{ $users->appends(['sort' => 'votes'])->links() }}
```

如果你想要添加“哈希片段”到分页链接，可以使用 `fragment` 方法。例如，要添加 `#foo` 到每个分页链接的末尾，像这样调用 `fragment` 方法：

```
{{ $users->fragment('foo')->links() }}
```

将结果转化为 JSON

Laravel 分页器结果类实现了 `Illuminate\Contracts\Support\JsonableInterface` 契约并提供了 `toJson` 方法，所以将分页结果转化为 JSON 非常简单。你还可以通过从路由或控制器动作返回分页器实例将转其化为 JSON：

```
Route::get('users', function () {
    return App\User::paginate();
});
```

从分页器转化来的 JSON 包含了元信息如 `total`、`current_page`、`last_page` 等等，实际的结果对象数据可以通过该 JSON 数组中的 `data` 键访问。下面是一个通过从路由返回的分页器实例创建的 JSON 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": [
    {
        // Result Object
    },
    {
        // Result Object
    }
]
}

```

自定义分页视图

默认情况下，用于渲染分页链接的视图兼容 Bootstrap CSS 框架，如果你没有使用 Bootstrap，可以自定义视图来渲染这些链接。当调用分页器实例上的 `links` 方法时，传递视图名称作为第一个参数：

```

{{ $paginator->links('view.name') }}

// 传递数据到视图...
{{ $paginator->links('view.name', ['foo' => 'bar']) }}

```

不过，自定义分页视图最简单的方式是使用 `vendor:publish` 命令导出视图文件到 `resources/views/vendor` 目录：

```
php artisan vendor:publish --tag=laravel-pagination
```

该命令会将视图放到 `resources/views/vendor/pagination` 目录，该目录下的 `default.blade.php` 文件对应默认的视图文件，编辑该文件即可修改分页 HTML。

如果你想要指定其他文件作为默认分页视图，可以在 `AppServiceProvider` 中使用分页器的 `defaultView` 和 `defaultSimpleView` 方法：

```

use Illuminate\Pagination\Paginator;

public function boot()
{
    Paginator::defaultView('pagination::view');

    Paginator::defaultSimpleView('pagination::view');
}

```

分页器实例方法

每个分页器实例都可以通过以下方法提供更多分页信息：

```

$results->count()
$results->currentPage()
$results->firstItem()
$results->hasMorePages()
$results->lastItem()

$results->lastPage() (使用 simplePaginate 时无效)
$results->nextPageUrl()
$results->perPage()
$results->previousPageUrl()

$results->total() (使用 simplePaginate 时无效)
$results->url($page)

```

迁移

简介

所谓迁移就像是数据库的版本控制，这种机制允许团队简单轻松的编辑并共享应用的数据库表结构。迁移通常和 Laravel 的 schema 构建器结对从而可以很容易地构建应用的数据库表结构。如果你曾经频繁告知团队成员需要手动添加列到本地数据库表结构以维护本地开发环境，那么这正是数据库迁移所致力于解决的问题。

Laravel 的 Schema 门面提供了与数据库系统无关的创建和操纵表的支持，在 Laravel 所支持的所有数据库系统中提供一致的、优雅的、流式的 API。

生成迁移

使用 Artisan 命令 `make:migration` 就可以创建一个新的迁移：

```
php artisan make:migration create_users_table
```

新的迁移位于 `database/migrations` 目录下，每个迁移文件名都包含时间戳从而允许 Laravel 判断其顺序。

`--table` 和 `--create` 选项可以用于指定表名以及该迁移是否要创建一个新的数据表。这些选项只需要简单放在上述迁移命令后面并指定表名：

```
php artisan make:migration create_users_table --create=users
php artisan make:migration add_votes_to_users_table --table=users
```

如果你想要指定生成迁移的自定义输出路径，在执行 `make:migration` 命令时可以使用 `--path` 选项，提供的路径应该是相对于应用根目录的。

迁移结构

迁移类包含了两个方法：`up` 和 `down`。`up` 方法用于新增表，列或者索引到数据库，而 `down` 方法就是 `up` 方法的逆操作，和 `up` 里的操作相反。

在这两个方法中你都要用到 Laravel 的 Schema 构建器来创建和修改表，要了解更多 Schema 构建器提供的方法，[查看其文档](#)。下面让我们先看看创建 `flights` 表的简单示例：

```
<?php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateFlightsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('flights');
    }
}
```

运行迁移

要运行应用中所有未执行的迁移，可以使用 Artisan 命令提供的 `migrate` 方法：

```
php artisan migrate
```

注：如果你正在使用 Homestead 虚拟机，需要在虚拟机中运行上面这条命令。

在生产环境中强制运行迁移

有些迁移操作是毁灭性的，这意味着它们可能造成数据的丢失，为了避免在生产环境数据库中运行这些命令，你将会在运行这些命令之前被提示并确认。想要强制运行这些命令而不被提示，可以使用 `--force` 标记：

```
php artisan migrate --force
```

回滚迁移

想要回滚最新的一次迁移“操作”，可以使用 `rollback` 命令，注意这将会回滚最后一批运行的迁移，可能包含多个迁移文件：

```
php artisan migrate:rollback
```

你也可以通过 `rollback` 命令上提供的 `step` 选项来回滚指定数目的迁移，例如，下面的命令将会回滚最后五条迁移：

```
php artisan migrate:rollback --step=5
```

`migrate:reset` 命令将会回滚所有的应用迁移：

```
php artisan migrate:reset
```

在单个命令中回滚 & 迁移

`migrate:refresh` 命令将会先回滚所有数据库迁移，然后运行 `migrate` 命令。这个命令可以有效的重建整个数据库：

```
php artisan migrate:refresh
```

```
// 重建数据库并填充数据...
```

```
php artisan migrate:refresh --seed
```

当然，你也可以回滚或重建指定数量的迁移 —— 通过 `refresh` 命令提供的 `step` 选项，例如，下面的命令将会回滚或重建最后五条迁移：

```
php artisan migrate:refresh --step=5
```

删除所有表 & 迁移

`migrate:fresh` 命令将会先从数据库中删除所有表然后执行 `migrate` 命令：

```
php artisan migrate:fresh
```

```
php artisan migrate:fresh --seed
```

数据表

创建表

使用 Schema 门面上的 `create` 方法来创建新的数据表。`create` 方法接收两个参数，第一个是表名，第二个是获取用于定义新表的 `Blueprint` 对象的闭包：

```
Schema::create('users', function ($table) {
    $table->increments('id');
});
```

当然，创建新表的时候，可以使用 Schema 构建器中的任意 [列方法](#) 来定义数据表的列。

检查表/列是否存在

你可以轻松地使用 `hasTable` 和 `hasColumn` 方法检查表或列是否存在：

```
if (Schema::hasTable('users')) {
    //
}

if (Schema::hasColumn('users', 'email')) {
    //
}
```

数据库连接 & 表选项

如果你想要在一个数据库连接上执行表结构操作，而该数据库连接并不是默认数据库连接，可以使用 `connection` 方法：

```
Schema::connection('foo')->create('users', function (Blueprint $table) {
    $table->increments('id');
});
```

要设置表的存储引擎、字符编码等选项，可以在 Schema 构建器上使用如下命令：

命令	描述
<code>\$table->engine = 'InnoDB';</code>	指定表的存储引擎 (MySQL)
<code>\$table->charset = 'utf8';</code>	指定数据表的默认字符集 (MySQL)
<code>\$table->collation = 'utf8_unicode_ci';</code>	指定数据表的字符序 (MySQL)
<code>\$table->temporary();</code>	创建临时表 (除 SQL Server)

重命名/删除表

要重命名一个已存在的数据表，使用 `rename` 方法：

```
Schema::rename($from, $to);
```

要删除一个已存在的数据表，可以使用 `drop` 或 `dropIfExists` 方法：

```
Schema::drop('users');
Schema::dropIfExists('users');
```

通过外键重命名表

在重命名表之前，需要验证该表包含的外键在迁移文件中有明确的名字，而不是 Laravel 基于惯例分配的名字。否则，外键约束名将会指向旧的数据表。

数据列

创建数据列

要更新一个已存在的表，使用 Schema 门面上的 `table` 方法，和 `create` 方法一样，`table` 方法接收两个参数：表名和获取用于添加列到表的 `Blueprint` 实例的闭包：

```
Schema::table('users', function (Blueprint $table) {
    $table->string('email');
});
```

可用的数据列类型

当然，Schema 构建器包含一系列你可以用来构建表的列类型：

命令	描述
<code>\$table->bigIncrements('id');</code>	等同于自增 UNSIGNED BIGINT (主键) 列
<code>\$table->bigInteger('votes');</code>	等同于 BIGINT 类型列
<code>\$table->binary('data');</code>	等同于 BLOB 类型列
<code>\$table->boolean('confirmed');</code>	等同于 BOOLEAN 类型列
<code>\$table->char('name', 4);</code>	等同于 CHAR 类型列
<code>\$table->date('created_at');</code>	等同于 DATE 类型列
<code>\$table->dateTime('created_at');</code>	等同于 DATETIME 类型列
<code>\$table->dateTimeTz('created_at');</code>	等同于 DATETIME 类型 (带时区) 列
<code>\$table->decimal('amount', 5, 2);</code>	等同于 DECIMAL 类型列，带精度和范围
<code>\$table->double('column', 15, 8);</code>	等同于 DOUBLE 类型列，带精度，总共 15 位数字，小数点后 8 位
<code>\$table->enum('level', ['easy', 'hard']);</code>	等同于 ENUM 类型列
<code>\$table->float('amount', 8, 2);</code>	等同于 FLOAT 类型列，带精度和总位数
<code>\$table->geometry('positions');</code>	等同于 GEOMETRY 类型列
<code>\$table->geometryCollection('positions');</code>	等同于 GEOMETRYCOLLECTION 类型列
<code>\$table->increments('id');</code>	等同于自增 UNSIGNED INTEGER (主键) 类型列
<code>\$table->integer('votes');</code>	等同于 INTEGER 类型列
<code>\$table->ipAddress('visitor');</code>	等同于 IP 地址类型列
<code>\$table->json('options');</code>	等同于 JSON 类型列
<code>\$table->jsonb('options');</code>	等同于 JSONB 类型列
<code>\$table->lineString('positions');</code>	等同于 LINESTRING 类型列
<code>\$table->longText('description');</code>	等同于 LONGTEXT 类型列
<code>\$table->macAddress('device');</code>	等同于 MAC 地址类型列
<code>\$table->mediumIncrements('id');</code>	等同于自增 UNSIGNED MEDIUMINT 类型列 (主键)

命令	描述
<code>\$table->mediumInteger('numbers');</code>	等同于 MEDIUMINT 类型列
<code>\$table->mediumText('description');</code>	等同于 MEDIUMTEXT 类型列
<code>\$table->morphs('taggable');</code>	添加一个 UNSIGNED INTEGER 类型的 <code>taggable_id</code> 列和一个 VARCHAR 类型的 <code>taggable_type</code> 列
<code>\$table->multiLineString('positions');</code>	等同于 MULTILINESTRING 类型列
<code>\$table->multiPoint('positions');</code>	等同于 MULTIPOINT 类型列
<code>\$table->multiPolygon('positions');</code>	等同于 MULTIPOLYGON 类型列
<code>\$table->nullableMorphs('taggable');</code>	<code>morphs()</code> 列的 nullable 版本
<code>\$table->nullableTimestamps();</code>	<code>timestamps()</code> 的别名
<code>\$table->point('position');</code>	等同于 POINT 类型列
<code>\$table->polygon('positions');</code>	等同于 POLYGON 类型列
<code>\$table->rememberToken();</code>	等同于添加一个允许为空的 <code>remember_token</code> VARCHAR(100) 列
<code>\$table->smallIncrements('id');</code>	等同于自增 UNSIGNED SMALLINT (主键) 类型列
<code>\$table->smallInteger('votes');</code>	等同于 SMALLINT 类型列
<code>\$table->softDeletes();</code>	新增一个允许为空的 <code>deleted_at</code> TIMESTAMP 列用于软删除
<code>\$table->softDeletesTz();</code>	新增一个允许为空的 <code>deleted_at</code> TIMESTAMP (带时区) 列用于软删除
<code>\$table->string('name', 100);</code>	等同于 VARCHAR 类型列, 带一个可选长度参数
<code>\$table->text('description');</code>	等同于 TEXT 类型列
<code>\$table->time('sunrise');</code>	等同于 TIME 类型列
<code>\$table->timeTz('sunrise');</code>	等同于 TIME 类型 (带时区)
<code>\$table->timestamp('added_on');</code>	等同于 TIMESTAMP 类型列
<code>\$table->timestampTz('added_on');</code>	等同于 TIMESTAMP 类型 (带时区) 列
<code>\$table->timestamps();</code>	添加允许为空的 <code>created_at</code> 和 <code>updated_at</code> TIMESTAMP 类型列
<code>\$table->timestampsTz();</code>	添加允许为空的 <code>created_at</code> 和 <code>updated_at</code> TIMESTAMP 类型列 (带时区)
<code>\$table->tinyIncrements('numbers');</code>	等同于自增的 UNSIGNED TINYINT 类型列 (主键)
<code>\$table->tinyInteger('numbers');</code>	等同于 TINYINT 类型列
<code>\$table->unsignedBigInteger('votes');</code>	等同于无符号的 BIGINT 类型列
<code>\$table->unsignedDecimal('amount', 8, 2);</code>	等同于 UNSIGNED DECIMAL 类型列, 带有总位数和精度
<code>\$table->unsignedInteger('votes');</code>	等同于无符号的 INTEGER 类型列
<code>\$table->unsignedMediumInteger('votes');</code>	等同于无符号的 MEDIUMINT 类型列
<code>\$table->unsignedSmallInteger('votes');</code>	等同于无符号的 SMALLINT 类型列
<code>\$table->unsignedTinyInteger('votes');</code>	等同于无符号的 TINYINT 类型列
<code>\$table->uuid('id');</code>	等同于 UUID 类型列
<code>\$table->year('birth_year');</code>	等同于 YEAR 类型列

列修改器

除了上面列出的数据列类型之外，在添加列的时候还可以使用一些其它的列“修改器”，例如，要使列允许为 NULL，可以使用 `nullable` 方法：

```
Schema::table('users', function (Blueprint $table) {
    $table->string('email')->nullable();
});
```

下面是所有可用的列修改器列表，该列表不包含索引修改器：

修改器	描述
<code>->after('column')</code>	将该列置于另一个列之后 (MySQL)
<code>->autoIncrement()</code>	设置 INTEGER 列为自增主键
<code>->charset('utf8')</code>	指定数据列字符集 (MySQL)
<code>->collation('utf8_unicode_ci')</code>	指定数据列字符序 (MySQL/SQL Server)
<code>->comment('my comment')</code>	添加注释信息
<code>->default(\$value)</code>	指定列的默认值
<code>->first()</code>	将该列置为表中第一个列 (MySQL)
<code>->nullable(\$value = true)</code>	允许该列的值为 NULL
<code>->storedAs(\$expression)</code>	创建一个存储生成列 (MySQL)
<code>->unsigned()</code>	设置 INTEGER 列为 UNSIGNED (MySQL)
<code>->useCurrent()</code>	设置 TIMESTAMP 列使用 CURRENT_TIMESTAMP 作为默认值
<code>->virtualAs(\$expression)</code>	创建一个虚拟生成列 (MySQL)

修改数据列

先决条件

在修改列之前，确保已经将 `doctrine/dbal` 依赖添加到 `composer.json` 文件，Doctrine DBAL 库用于判断列的当前状态并创建对列进行指定调整所需的 SQL 语句：

```
composer require doctrine/dbal
```

更新列属性

`change` 方法允许你修改已存在的列为新的类型，或者修改列的属性。例如，你可能想要增加 字符串类型列的尺寸，下面让我们将 `name` 列的尺寸从 25 增加到 50：

```
Schema::table('users', function (Blueprint $table) {
    $table->string('name', 50)->change();
});
```

我们还可以修改该列允许 NULL 值：

```
Schema::table('users', function (Blueprint $table) {
    $table->string('name', 50)->nullable()->change();
});
```

注：只有以下数据列类型能修改：

`bigInteger`, `binary`, `boolean`, `date`, `dateTime`, `dateTimeTz`, `decimal`, `integer`, `json`, `longText`, `mediumText`, `smallInteger`, `string`, `text`, `time`, `unsignedBigInteger`, `unsignedInteger` 和 `unsignedSmallInteger`。

重命名列

要重命名一个列，可以使用表结构构建器上的 `renameColumn` 方法，在重命名一个列之前，确保 `doctrine/dbal` 依赖已经添加到 `composer.json` 文件并且已经运行了 `composer update` 命令：

```
Schema::table('users', function (Blueprint $table) {
    $table->renameColumn('from', 'to');
});
```

注：暂不支持 `enum` 类型的列的修改和重命名。

删除数据列

要删除一个列，使用 Schema 构建器上的 `dropColumn` 方法，同样，在此之前，确保已经安装了 `doctrine/dbal` 依赖：

```
Schema::table('users', function (Blueprint $table) {
    $table->dropColumn('votes');
```

```
});
```

你可以通过传递列名数组到 `dropColumn` 方法以便可以一次从数据表中删除多个列:

```
Schema::table('users', function (Blueprint $table) {
    $table->dropColumn(['votes', 'avatar', 'location']);
});
```

注: SQLite 数据库暂不支持在单个迁移中删除或修改多个列。

有效的命令别名

命令	描述
<code>\$table->dropRememberToken();</code>	删除 <code>remember_token</code> 列
<code>\$table->dropSoftDeletes();</code>	删除 <code>deleted_at</code> 列
<code>\$table->dropSoftDeletesTz();</code>	<code>dropSoftDeletes()</code> 方法别名
<code>\$table->dropTimestamps();</code>	删除 <code>created_at</code> 和 <code>updated_at</code> 列
<code>\$table->dropTimestampsTz();</code>	<code>dropTimestamps()</code> 方法别名

索引

创建索引

Schema 构建器支持多种类型的索引, 首先, 让我们看一个指定列值为唯一索引的例子。要创建该索引, 可以使用 `unique` 方法:

```
$table->string('email')->unique();
```

此外, 你可以在定义列之后创建索引, 例如:

```
$table->unique('email');
```

你甚至可以传递列名数组到索引方法来创建组合索引:

```
$table->index(['account_id', 'created_at']);
```

Laravel 会自动生成合理的索引名称, 不过你也可以传递第二个参数到该方法用于指定索引名称:

```
$table->index('email', 'unique_email');
```

可用索引类型

命令	描述
<code>\$table->primary('id');</code>	添加主键索引
<code>\$table->primary(['id', 'parent_id']);</code>	添加组合索引
<code>\$table->unique('email');</code>	添加唯一索引
<code>\$table->index('state');</code>	添加普通索引
<code>\$table->spatialIndex('location');</code>	添加空间索引 (不支持 SQLite)

索引长度 & MySQL / MariaDB

Laravel 默认使用 `utf8mb4` 字符集, 支持在数据库中存储 emoji 表情。如果你现在运行的 MySQL 版本低于 5.7.7 (或者低于 10.2.2 版本的 MariaDB), 需要手动配置迁移命令生成的默认字符串长度, 以便 MySQL 为它们创建索引。你可以通过在 `AppServiceProvider` 中调用 `Schema::defaultStringLength` 方法来完成配置:

```
use Illuminate\Support\Facades\Schema;

/**
 * Bootstrap any application services.
 *
 * @return void
 * @translator laravelacademy.org
 */
public function boot()
{
    Schema::defaultStringLength(191);
}
```

作为可选方案, 你可以为数据库启用 `innodb_large_prefix` 选项, 至于如何合理启用这个选项, 可以参考数据库文档说明。

删除索引

要删除索引，必须指定索引名。默认情况下，Laravel 自动分配适当的名称给索引 —— 连接表名、列名和索引类型。下面是一些例子：

命令	描述
<code>\$table->dropPrimary('users_id_primary');</code>	从 “users” 表中删除主键索引
<code>\$table->dropUnique('users_email_unique');</code>	从 “users” 表中删除唯一索引
<code>\$table->dropIndex('geo_state_index');</code>	从 “geo” 表中删除普通索引
<code>\$table->dropSpatialIndex('geo_location_spatialindex');</code>	从 “geo” 表中删除空间索引（不支持 SQLite）

如果要传递数据列数组到删除索引方法，那么相应的索引名称将会通过数据表名、列和键类型来自动生成：

```
Schema::table('geo', function (Blueprint $table) {
    $table->dropIndex(['state']); // Drops index 'geo_state_index'
});
```

外键约束

Laravel 还提供了创建外键约束的支持，用于在数据库层面强制引用完整性。例如，我们在 `posts` 表中定义了一个引用 `users` 表 `id` 列的 `user_id` 列：

```
Schema::table('posts', function (Blueprint $table) {
    $table->integer('user_id')->unsigned();
    $table->foreign('user_id')->references('id')->on('users');
});
```

你还可以为约束的“on delete”和“on update”属性指定期望的动作：

```
$table->foreign('user_id')
    ->references('id')->on('users')
    ->onDelete('cascade');
```

要删除一个外键，可以使用 `dropForeign` 方法。外键约束和索引使用同样的命名规则 —— 连接表名、外键名然后加上“_foreign”后缀：

```
$table->dropForeign('posts_user_id_foreign');
```

或者，你还可以传递在删除时会自动使用基于惯例的约束名数值数组：

```
$table->dropForeign(['user_id']);
```

你可以在迁移时通过以下方法启用或关闭外键约束：

```
Schema::enableForeignKeyConstraints();
Schema::disableForeignKeyConstraints();
```

学院君注：由于使用外键风险级联删除风险较高，一般情况下我们很少使用外键，而是通过代码逻辑来实现级联操作。

数据填充

简介

Laravel 使用填充类提供了一个简单方法来填充测试数据到数据库。所有的填充类都位于 `database/seeds` 目录。填充类的类名完全由你自定义，但最好还是遵循一定的规则，例如 `UsersTableSeeder` 等。安装完 Laravel 后，默认会提供一个 `DatabaseSeeder` 示例类。从这个类中，你可以使用 `call` 方法来运行其他填充类，从而允许你控制填充顺序。

编写填充器

要生成一个填充器，可以通过 Artisan 命令 `make:seeder`。所有框架生成的填充器都位于 `database/seeds` 目录：

```
php artisan make:seeder UsersTableSeeder
```

一个填充器类默认只包含一个方法：`run`。当 Artisan 命令 `db:seed` 运行时该方法被调用。在 `run` 方法中，可以插入任何你想插入数据库的数据，你可以使用 `查询构建器` 手动插入数据，也可以使用 `Eloquent 模型工厂`（后面数据库测试部分文档会详述这个概念）。

注：数据库填充期间 `模型批量赋值`（后面 Eloquent ORM 中会详述这个概念）自动被禁止。

举个例子，让我们修改 Laravel 安装时自带的 `DatabaseSeeder` 类，添加一个数据库插入语句到 `run` 方法：

```
<?php

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

```

class DatabaseSeeder extends Seeder
{
    /**
     * 运行数据库填充
     *
     * @return void
     */
    public function run()
    {
        DB::table('users')->insert([
            'name' => str_random(10),
            'email' => str_random(10).'@gmail.com',
            'password' => bcrypt('secret'),
        ]);
    }
}

```

使用模型工厂

当然，手动指定每一个模型填充的属性是很笨重累赘的，取而代之的，我们可以使用模型工厂来快速生成大量的数据库记录。首先，查看[模型工厂文档](#)来学习如何定义工厂，定义完工厂后，可以使用辅助函数 `factory` 来插入记录到数据库。

举个例子，让我们创建 50 个用户并添加关联关系到每个用户：

```

/**
 * 运行数据库填充
 *
 * @return void
 */
public function run() {
    factory(App\User::class, 50)->create()->each(function($u) {
        $u->posts()->save(factory(App\Post::class)->make());
    });
}

```

调用其他填充器

在 `DatabaseSeeder` 类中，你可以使用 `call` 方法执行其他填充类，使用 `call` 方法允许你将数据库填充分解成多个文件，这样单个填充器类就不会变得无比巨大，只需简单将你想要运行的填充器类名传递过去即可：

```

/**
 * 运行数据库填充
 *
 * @return void
 */
public function run() {
    $this->call(UsersTableSeeder::class);
    $this->call(PostsTableSeeder::class);
    $this->call(CommentsTableSeeder::class);
}

```

运行填充器

编写好填充器类之后，需要通过 `dump-autoload` 命令重新生成 Composer 的自动加载器：

```
composer dump-autoload
```

运行之后可以使用 Artisan 命令 `db:seed` 来填充数据库。默认情况下，`db:seed` 命令运行 `DatabaseSeeder` 类，不过，你也可以使用 `--class` 选项来指定你想要运行的独立的填充器类：

```
php artisan db:seed
php artisan db:seed --class=UsersTableSeeder
```

你还可以使用 `migrate:refresh` 命令来填充数据库，该命令还可以回滚并重新运行所有迁移，这在需要完全重建数据库时很有用：

```
php artisan migrate:refresh --seed
```

Redis

简介

Redis 是一个开源的、高级的键值对存储系统，经常被用作数据结构服务器，因为其支持字符串、Hash、列表、集合和有序集合等数据结构。在 Laravel 中使用 Redis 之前，需要通过 Composer 安装 `predis/predis` 包：

```
composer require predis/predis
```

作为替代方案，你还可以通过 PECL 安装 PHP 扩展 `PhpRedis`。该扩展安装起来更麻烦，但是对重度使用 Redis 的应用而言性能更好。

配置

应用的 Redis 配置位于配置文件 `config/database.php`。在这个文件中，可以看到包含被应用使用的 Redis 服务器的 `redis` 数组：

```
'redis' => [
    'client' => 'predis',
    'default' => [
        'host' => env('REDIS_HOST', 'localhost'),
        'password' => env('REDIS_PASSWORD', null),
        'port' => env('REDIS_PORT', 6379),
        'database' => 0,
    ],
],
```

默认服务器配置可以满足开发需要，不过，你可以基于自己的环境修改该数组。配置文件中定义的每个 Redis 服务器需要一个名字并指定该 Redis 服务器使用的主机和接口。

配置集群

如果应用使用了 Redis 服务器集群，需要在 Redis 配置中通过 `clusters` 定义这些集群：

```
'redis' => [
    'client' => 'predis',
    'clusters' => [
        'default' => [
            [
                'host' => env('REDIS_HOST', 'localhost'),
                'password' => env('REDIS_PASSWORD', null),
                'port' => env('REDIS_PORT', 6379),
                'database' => 0,
            ],
        ],
    ],
],
```

默认情况下，集群将会在节点之间进行客户端分区，从而允许你构建节点池并创建大量可用内存。不过，客户端分片并不处理故障转移，所以，非常适合从另一个主数据存储那里获取有效的缓存数据。如果你想要使用本地 Redis 集群，需要在 Redis 配置的 `options` 中进行指定：

```
'redis' => [
    'client' => 'predis',
    'options' => [
        'cluster' => 'redis',
    ],
    'clusters' => [
        // ...
    ],
],
```

Predis

除了默认的 `host`、`port`、`database` 和 `password` 服务器配置选项，Predis 还支持额外的用于定义每个 Redis 服务器的连接参数，要使用这些额外的配置项，只需在配置文件 `config/database.php` 中将它们添加到 Redis 服务器配置中：

```
'default' => [
    'host' => env('REDIS_HOST', 'localhost'),
```

```
'password' => env('REDIS_PASSWORD', null),
'port' => env('REDIS_PORT', 6379),
'database' => 0,
'read_write_timeout' => 60,
],
```

PhpRedis

注：如果你通过 PECL 安装了 PHP 扩展 PhpRedis，需要在配置文件 `config/app.php` 中重命名 `Redis` 别名。要使用 PhpRedis 扩展，需要在 Redis 配置中将 `client` 选项修改为 `phpredis`，该选项位于配置文件 `config/database`：

```
'redis' => [
    'client' => 'phpredis',
    // Rest of Redis configuration...
],
```

除了默认的 `host`、`port`、`database` 和 `password` 服务器配置选项，PhpRedis 还支持额外的连接参数：`persistent`、`prefix`、`read_timeout` 和 `timeout`，你可以在配置文件 `config/database.php` 中将它们添加到 Redis 服务器配置中：

```
'default' => [
    'host' => env('REDIS_HOST', 'localhost'),
    'password' => env('REDIS_PASSWORD', null),
    'port' => env('REDIS_PORT', 6379),
    'database' => 0,
    'read_timeout' => 60,
],
```

与 Redis 交互

你可以通过调用 `Redis` 门面上的方法来与 Redis 进行交互，该门面支持动态方法，所以你可以调用任何 `Redis` 命令，对应命令将会直接传递给 Redis，在本例中，我们通过调用 Redis 门面上的 `get` 方法来调用 Redis 上的 `GET` 命令：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Redis;
use App\Http\Controllers\Controller;

class UserController extends Controller{
    /**
     * 显示指定用户属性
     *
     * @param int $id
     * @return Response
     * @translator laravelacademy.org
     */
    public function showProfile($id)
    {
        $user = Redis::get('user:profile:'.$id);
        return view('user.profile', ['user' => $user]);
    }
}
```

当然，如上所述，可以在 `Redis` 门面上调用任何 Redis 命令。Laravel 使用魔术方法将命令传递给 Redis 服务器，所以只需简单传递参数和 Redis 命令如下：

```
Redis::set('name', 'Taylor');

$values = Redis::lrange('names', 5, 10);
```

此外，还可以使用 `command` 方法传递命令到服务器，该方法接收命令名作为第一个参数，参数值数组作为第二个参数：

```
$values = Redis::command('lrange', ['name', 5, 10]);
```

学院君注：如果要使用 Redis 作为缓存驱动，可以参考[缓存文档](#)；如果要使用 Redis 作为队列驱动，可以参考[队列文档](#)。

使用多个 Redis 连接

你可以通过调用 `Redis::connection` 方法获取 Redis 实例：

```
$redis = Redis::connection();
```

这将会获取默认 Redis 服务器实例，你还可以传递服务器名或集群名到 `connection` 方法来获取 Redis 配置中定义的指定服务器或集群：

```
$redis = Redis::connection('my-connection');
```

管道命令

当你需要在一次操作中发送多个命令到服务器的时候应该使用管道，`pipeline` 方法接收一个参数：接收 Redis 实例的闭包。你可以将所有 Redis 命令发送到这个 Redis 实例，然后这些命令会在一次操作中被执行：

```
Redis::pipeline(function ($pipe) {
    for ($i = 0; $i < 1000; $i++) {
        $pipe->set("key:$i", $i);
    }
});
```

发布/订阅

Redis 还提供了调用 Redis 的 `publish` 和 `subscribe` 命令的接口。这些 Redis 命令允许你在给定“频道”监听消息，你可以从另外一个应用发布消息到这个频道，甚至使用其它编程语言，从而允许你在不同的应用/进程之间轻松通信。

首先，让我们使用 `subscribe` 方法通过 Redis 在一个频道上设置监听器。由于调用 `subscribe` 方法会开启一个常驻进程，我们将在 `Artisan` 命令中调用该方法：

```
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;
use Illuminate\Support\Facades\Redis;

class RedisSubscribe extends Command
{
    /**
     * 控制台命令名称
     *
     * @var string
     */
    protected $signature = 'redis:subscribe';

    /**
     * 控制台命令描述
     *
     * @var string
     */
    protected $description = 'Subscribe to a Redis channel';

    /**
     * 执行控制台命令
     *
     * @return mixed
     */
    public function handle()
    {
        Redis::subscribe(['test-channel'], function($message) {
            echo $message;
        });
    }
}
```

现在，我们可以使用 `publish` 发布消息到该频道：

```
Route::get('publish', function () {
    // 路由逻辑...
    Redis::publish('test-channel', json_encode(['foo' => 'bar']));
});
```

通配符订阅

使用 `psubscribe` 方法，你可以订阅到一个通配符定义的频道，这在所有相应频道上获取所有消息时很有用。`$channel` 名将会作为第二个参数传递给提供的回调闭包：

```
Redis::psubscribe(['*'], function($message, $channel) {
    echo $message;
});

Redis::psubscribe(['users.*'], function($message, $channel) {
    echo $message;
```

```
});
```

学院君注：要使用 Redis 作为事件广播服务端，可以参考[广播文档](#)。

七、 Eloquent ORM

快速入门

简介

Laravel 内置的 Eloquent ORM 提供了一个美观、简单的与数据库打交道的 ActiveRecord 实现，每张数据表都对应一个与该表进行交互的模型（Model），通过模型类，你可以对数据表进行查询、插入、更新、删除等操作。

在开始之前，确保在 `config/database.php` 文件中配置好了数据库连接。更多关于数据库配置的信息，请[查看文档](#)。

定义模型

我们从创建一个 Eloquent 模型开始，模型类通常位于 `app` 目录下，你也可以将其放在其他可以被 `composer.json` 文件自动加载到的地方。所有 Eloquent 模型都继承自 `Illuminate\Database\Eloquent\Model` 类。

创建模型实例最简单办法就是使用 Artisan 命令 `make:model`：

```
php artisan make:model User
```

如果你想要在生成模型时生成数据库迁移，可以使用 `--migration` 或 `-m` 选项：

```
php artisan make:model User --migration
php artisan make:model User -m
```

Eloquent 模型约定

现在，让我们来看一个 `Flight` 模型的例子，我们将用该类获取和存取数据表 `flights` 中的信息：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    //
```

表名

注意我们并没有告诉 Eloquent 我们的 `Flight` 模型使用哪张表，默认规则是小写的模型类名复数格式作为与其对应的表名（除非在模型类中明确指定了其它名称）。所以，在本例中，Eloquent 认为 `Flight` 模型存储记录在 `flights` 表中。你也可以在模型中定义 `table` 属性来指定自定义的表名：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * 关联到模型的数据表
     *
     * @var string
     */
    protected $table = 'my_flights';
}
```

主键

Eloquent 默认每张表的主键名为 `id`，你可以在模型类中定义一个 `$primaryKey` 属性来覆盖该约定。

此外，Eloquent 默认主键字段是自增的整型数据，这意味着主键将会被自动转化为 `int` 类型，如果你想要使用非自增或非数字类型主键，必须在对应模型中设置 `$incrementing` 属性为 `false`，如果主键不是整型，还要设置 `$keyType` 属性值为 `string`。

时间戳

默认情况下，Eloquent 期望 `created_at` 和 `updated_at` 已经存在于数据表中，如果你不想要这些 Laravel 自动管理的数据列，在模型类中设置 `$timestamps` 属性为 `false`：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * 表明模型是否应该被打上时间戳
     *
     * @var bool
     */
    public $timestamps = false;
}
```

如果你需要自定义时间戳格式，设置模型中的 `$dateFormat` 属性。该属性决定日期被如何存储到数据库中，以及模型被序列化为数组或 JSON 时日期的格式：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * 模型日期列的存储格式
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

如果你需要自定义用于存储时间戳的字段名称，可以在模型中设置 `CREATED_AT` 和 `UPDATED_AT` 常量：

```
<?php

class Flight extends Model
{
    const CREATED_AT = 'creation_date';
    const UPDATED_AT = 'last_update';
}
```

数据库连接

默认情况下，所有的 Eloquent 模型使用应用配置中的默认数据库连接，如果你想要为模型指定不同的连接，可以通过 `$connection` 属性来设置：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The connection name for the model.
     *
     * @var string
     */
    protected $connection = 'connection-name';
}
```

获取模型

创建完模型及其[关联的数据表](#)后，就可以从数据库中获取数据了。将 Eloquent 模型看作功能强大的[查询构建器](#)，你可以使用它来流畅的查询与其关联的数据表。例如：

```
<?php

use App\Flight;
```

```
$flights = App\Flight::all();

foreach ($flights as $flight) {
    echo $flight->name;
}
```

添加额外约束

Eloquent 的 `all` 方法返回模型表的所有结果，由于每一个 Eloquent 模型都是一个查询构建器，你还可以添加约束条件到查询，然后使用 `get` 方法获取对应结果：

```
$flights = App\Flight::where('active', 1)
    ->orderBy('name', 'desc')
    ->take(10)
    ->get();
```

注：由于 Eloquent 模型本质上就是查询构建器，你可以在 Eloquent 查询中使用查询构建器的所有方法。

集合

对 Eloquent 中获取多个结果的方法（比如 `all` 和 `get`）而言，其返回值是 `Illuminate\Database\Eloquent\Collection` 的一个实例，`Collection` 类提供了多个有用的函数来处理 Eloquent 结果集：

```
$flights = $flights->reject(function ($flight) {
    return $flight->cancelled;
});
```

当然，你也可以像数组一样循环遍历该集合：

```
foreach ($flights as $flight) {
    echo $flight->name;
}
```

组块结果集

如果你需要处理数据量很大的 Eloquent 结果集，可以使用 `chunk` 方法。`chunk` 方法会获取一个指定数量的 Eloquent 模型“组块”，并将其填充到给定闭包进行处理。使用 `chunk` 方法在处理大量数据集合时能够有效减少内存消耗：

```
Flight::chunk(200, function ($flights) {
    foreach ($flights as $flight) {
        //
    }
});
```

传递给该方法的第一个参数是你想要获取的“组块”数目，闭包作为第二个参数被传入用于处理每个从数据库获取的组块数据。

使用游标

`cursor` 方法允许你使用游标迭代处理数据库记录，一次只执行单个查询，在处理大批量数据时，`cursor` 方法可大幅减少内存消耗：

```
foreach (Flight::where('foo', 'bar')->cursor() as $flight) {
    //
}
```

获取单个模型/聚合结果

当然，除了从给定表中获取所有记录之外，还可以使用 `find` 和 `first` 获取单个记录。这些方法返回单个模型实例而不是模型集合：

```
// 通过主键获取模型...
$flight = App\Flight::find(1);

// 获取匹配查询条件的第一个模型...
$flight = App\Flight::where('active', 1)->first();
```

还可以通过传递主键数组来调用 `find` 方法，这将会返回匹配记录集合：

```
$flights = App\Flight::find([1, 2, 3]);
```

Not Found 异常

有时候你可能想要在模型找不到的时候抛出异常，这在路由或控制器中非常有用，`findOrFail` 和 `firstOrFail` 方法会获取查询到的第一个结果。不过，如果没有任何查询结果，`Illuminate\Database\Eloquent\ModelNotFoundException` 异常将会被抛出：

```
$model = App\Flight::findOrFail(1);
$model = App\Flight::where('legs', '>', 100)->firstOrFail();
```

如果异常没有被捕获，那么 HTTP 404 响应将会被发送给用户，所以在使用这些方法的时候没有必要对返回 404 响应编写额外的检查：

```
Route::get('/api/flights/{id}', function ($id) {
    return App\Flight::findOrFail($id);
```

```
});
```

获取聚合结果

当然，你还可以使用查询构建器提供的聚合方法，例如 `count`、`sum`、`max`，以及其它查询构建器提供的聚合函数。这些方法返回计算后的结果而不是整个模型实例：

```
$count = App\Flight::where('active', 1)->count();
$max = App\Flight::where('active', 1)->max('price');
```

插入/更新模型

插入

想要在数据库中插入新的记录，只需创建一个新的模型实例，设置模型的属性，然后调用 `save` 方法：

```
<?php

namespace App\Http\Controllers;

use App\Flight;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class FlightController extends Controller{
    /**
     * 创建一个新的航班实例
     *
     * @param Request $request
     * @return Response
     * @translator laravelacademy.org
     */
    public function store(Request $request)
    {
        // 验证请求...

        $flight = new Flight;

        $flight->name = $request->name;

        $flight->save();
    }
}
```

在这个例子中，我们只是简单分配 HTTP 请求中的 `name` 参数值给 `App\Flight` 模型实例的 `name` 属性，当我们调用 `save` 方法时，一条记录将会被插入数据库。`created_at` 和 `updated_at` 时间戳在 `save` 方法被调用时会自动被设置，所以没必要手动设置它们。

更新

`save` 方法还可以用于更新数据库中已存在的模型。要更新一个模型，应该先获取它，设置你想要更新的属性，然后调用 `save` 方法。同样，`updated_at` 时间戳会被自动更新，所以没必要手动设置其值：

```
$flight = App\Flight::find(1);
$flight->name = 'New Flight Name';
$flight->save();
```

批量更新

更新操作还可以同时修改给定查询提供的多个模型实例，在本例中，所有有效且 `destination=San Diego` 的航班都被标记为延迟：

```
App\Flight::where('active', 1)
    ->where('destination', 'San Diego')
    ->update(['delayed' => 1]);
```

`update` 方法要求以数组形式传递键值对参数，代表着数据表中应该被更新的列。

注：通过 Eloquent 进行批量更新时，`saved` 和 `updated` 模型事件将不会在更新模型时触发。这是因为在进行批量更新时并没有从数据库获取模型。

批量赋值

还可以使用 `create` 方法保存一个新的模型。该方法返回被插入的模型实例。但是，在此之前，你需要指定模型的 `fillable` 或 `guarded` 属性，因为所有 Eloquent 模型都通过批量赋值（Mass Assignment）进行保护，这两个属性分别用于定义哪些模型字段允许批量赋值以及哪些模型字段是受保护的，不能显式进行批量赋值。

当用户通过 HTTP 请求传递一个不被期望的参数值时就会出现安全隐患，然后该参数以不被期望的方式修改数据库中的字段值。例如，恶意用户通过 HTTP 请求发送一个 `is_admin` 参数，然后该参数映射到模型的 `create` 方法，从而允许用户将自己变成管理员。

所以，你应该在模型中定义哪些属性是可以进行赋值的，使用模型上的 `$fillable` 属性即可实现。例如，我们设置 `Flight` 模型上的 `name` 属性可以被赋值：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * 可以被批量赋值的属性.
     *
     * @var array
     */
    protected $fillable = ['name'];
}
```

设置完可以被赋值的属性之后，我们就可以使用 `create` 方法在数据库中插入一条新的记录。`create` 方法返回保存后的模型实例：

```
$flight = App\Flight::create(['name' => 'Flight 10']);
```

如果你已经有了一个模型实例，可以使用 `fill` 方法通过数组属性来填充：

```
$flight->fill(['name' => 'Flight 22']);
```

黑名单属性

`$fillable` 就像是可以被赋值属性的“白名单”，还可以选择使用 `$guarded`。`$guarded` 属性包含你不想被赋值的属性数组。所以不被包含在其中的属性都是可以被赋值的，因此，`$guarded` 功能就像“黑名单”。当然，这两个属性你只能同时使用其中一个而不能一起使用，因为它们是互斥的。下面的例子中，除了 `price` 之外的所有属性都是可以赋值的：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * 不能被批量赋值的属性
     *
     * @var array
     */
    protected $guarded = ['price'];
}
```

如果你想要让所有属性都是可批量赋值的，可以将 `$guarded` 属性设置为空数组：

```
/**
 * The attributes that aren't mass assignable.
 *
 * @var array
 */
protected $guarded = [];
```

其它创建方法

`firstOrCreate`/`firstOrNew`

还有其它两种可以用来创建模型的方法：`firstOrCreate` 和 `firstOrNew`。`firstOrCreate` 方法先尝试通过给定列/值对在数据库中查找记录，如果没有找到的话则通过给定属性创建一个新的记录。

`firstOrNew` 方法和 `firstOrCreate` 方法一样先尝试在数据库中查找匹配的记录，如果没有找到，则返回一个新的模型实例。需要注意的是，通过 `firstOrNew` 方法返回的模型实例并没有持久化到数据库中，你还需要调用 `save` 方法手动持久化：

```
// 通过属性获取航班，如果不存在则创建...
$flight = App\Flight::firstOrCreate(['name' => 'Flight 10']);

// 通过 name 获取航班，如果不存在则通过 name 和 delayed 属性创建...
$flight = App\Flight::firstOrCreate(
    ['name' => 'Flight 10'], ['delayed' => 1]
);
```

```
// 通过属性获取航班，如果不存在初始化一个新的实例...
$flight = App\Flight::firstOrNew(['name' => 'Flight 10']);

// 通过 name 获取，如果不存在则通过 name 和 delayed 属性创建新实例...
$flight = App\Flight::firstOrNew(
    ['name' => 'Flight 10'], ['delayed' => 1]
);
```

updateOrCreate

与此类似的，你还会碰到如果模型已存在则更新，否则创建新模型的场景，Laravel 提供了一个 `updateOrCreate` 方法来一步完成。和 `firstOrCreate` 方法一样，`updateOrCreate` 方法会持久化模型，所以无需调用 `save()`：

```
// 如果有从奥克兰到圣地亚哥的航班则将价格设置为 $99
// 如果没有匹配的模型则创建之
$flight = App\Flight::updateOrCreate(
    ['departure' => 'Oakland', 'destination' => 'San Diego'],
    ['price' => 99]
);
```

删除模型

要删除一个模型，调用模型实例上的 `delete` 方法：

```
$flight = App\Flight::find(1);
$flight->delete();
```

通过主键删除模型

在上面的例子中，我们在调用 `delete` 方法之前从数据库中获取该模型，不过，如果你知道模型的主键的话，可以调用 `destroy` 方法直接删除而不需要获取它：

```
App\Flight::destroy(1);
App\Flight::destroy([1, 2, 3]);
App\Flight::destroy(1, 2, 3);
```

通过查询删除模型

当然，你还可以通过查询删除多个模型，在本例中，我们删除所有被标记为无效的航班：

```
$deletedRows = App\Flight::where('active', 0)->delete();
```

注：通过 Eloquent 进行批量删除时，`destroying` 和 `deleted` 模型事件在删除模型时不会被触发，这是因为在进行模型删除时不会获取模型。

软删除

除了从数据库物理删除记录外，Eloquent 还可以对模型进行“软删除”。当模型被软删除后，它们并没有真的从数据库删除，而是在模型上设置一个 `deleted_at` 属性并插入数据库，如果模型有一个非空 `deleted_at` 值，那么该模型已经被软删除了。要启用模型的软删除功能，可以使用模型上的 `Illuminate\Database\Eloquent\SoftDeletes` trait 并添加 `deleted_at` 列到 `$dates` 属性：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Flight extends Model
{
    use SoftDeletes;

    /**
     * 应该被调整为日期的属性
     *
     * @var array
     */
    protected $dates = ['deleted_at'];
}
```

当然，应该添加 `deleted_at` 列到数据表。Laravel Schema 构建器包含一个辅助函数来创建该数据列：

```
Schema::table('flights', function ($table) {
    $table->softDeletes();
});
```

现在，当调用模型的 `delete` 方法时，`deleted_at` 列将被设置为当前日期和时间，并且，当查询一个使用软删除的模型时，被软删除的模型将会自动从查询结果中排除。

判断给定模型实例是否被软删除，可以使用 `trashed` 方法：

```
if ($flight->trashed()) {
    //
}
```

查询被软删除的模型

包含软删除模型

正如上面提到的，软删除模型将会自动从查询结果中排除，不过，如果你想要软删除模型出现在查询结果中，可以使用 `withTrashed` 方法：

```
$flights = App\Flight::withTrashed()
    ->where('account_id', 1)
    ->get();
```

`withTrashed` 方法也可以用于关联查询中：

```
$flight->history()->withTrashed()->get();
```

只获取软删除模型

`onlyTrashed` 方法只获取软删除模型：

```
$flights = App\Flight::onlyTrashed()
    ->where('airline_id', 1)
    ->get();
```

恢复软删除模型

有时候你希望恢复一个被软删除的模型，可以使用 `restore` 方法：

```
$flight->restore();
```

你还可以在查询中使用 `restore` 方法来快速恢复多个模型，同样，这也不会触发任何模型事件：

```
App\Flight::withTrashed()
    ->where('airline_id', 1)
    ->restore();
```

和 `withTrashed` 方法一样，`restore` 方法也可以用于关联查询：

```
$flight->history()->restore();
```

永久删除模型

有时候你真的需要从数据库中删除一个模型，要从数据库中永久删除记录，可以使用 `forceDelete` 方法：

```
// 强制删除单个模型实例...
$flight->forceDelete();

// 强制删除所有关联模型...
$flight->history()->forceDelete();
```

查询作用域

全局作用域

全局作用域允许我们为给定模型的所有查询添加条件约束。Laravel 自带的软删除功能就使用了全局作用域来从数据库中拉出所有没有被删除的模型。编写自定义的全局作用域可以提供一种方便的、简单的方式来确保给定模型的每个查询都有特定的条件约束。

编写全局作用域

自定义全局作用域很简单，首先定义一个实现 `Illuminate\Database\Eloquent\Scope` 接口的类，该接口要求你实现一个方法：`apply`。需要的话可以在 `apply` 方法中添加 `where` 条件到查询：

```
<?php

namespace App\Scopes;

use Illuminate\Database\Eloquent\Scope;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Builder;

class AgeScope implements Scope
{
    /**
     * 应用作用域到给定的 Eloquent 查询构建器。
     *
     * @param \Illuminate\Database\Eloquent\Builder $builder
     */
    public function apply(Builder $builder)
    {
        $builder->where('age', '>', 18);
    }
}
```

```

 * @param \Illuminate\Database\Eloquent\Model $model
 * @return void
 * @translator laravelacademy.org
 */
public function apply(Builder $builder, Model $model)
{
    return $builder->where('age', '>', 200);
}
}

```

Laravel 应用默认并没有为作用域预定义文件夹，所以你可以按照自己的喜好在 `app` 目录下创建 `Scopes` 目录。

注：如果你的全局作用域需要添加列到查询的 `select` 子句，需要使用 `addSelect` 方法来替代 `select`，这样就可以避免已存在的 `select` 查询子句造成影响。

应用全局作用域

要将全局作用域应用到模型，需要重写给定模型的 `boot` 方法并使用 `addGlobalScope` 方法：

```

<?php

namespace App;

use App\Scopes\AgeScope;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 模型的“启动”方法.
     *
     * @return void
     */
    protected static function boot()
    {
        parent::boot();

        static::addGlobalScope(new AgeScope());
    }
}

```

添加作用域后，如果使用 `User::all()` 查询则会生成如下 SQL 语句：

```
select * from `users` where `age` > 200
```

匿名的全局作用域

Eloquent 还允许我们使用闭包定义全局作用域，这在实现简单作用域的时候特别有用，这样的话，我们就没必要定义一个单独的类了：

```

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Builder;

class User extends Model{
    /**
     * The "booting" method of the model.
     *
     * @return void
     */
    protected static function boot()
    {
        parent::boot();

        static::addGlobalScope('age', function(Builder $builder) {
            $builder->where('age', '>', 200);
        });
    }
}

```

移除全局作用域

如果想要在给定查询中移除指定全局作用域，可以使用 `withoutGlobalScope` 方法，该方法接收全局作用域的类名作为其唯一参数：

```
User::withoutGlobalScope(AgeScope::class)->get();
```

或者，如果你使用闭包定义的全局作用域的话：

```
User::withoutGlobalScope('age')->get();
```

如果你想要移除某几个或全部全局作用域，可以使用 `withoutGlobalScopes` 方法：

```
// 移除所有全局作用域
User::withoutGlobalScopes()->get();

// 移除某些全局作用域
User::withoutGlobalScopes([FirstScope::class, SecondScope::class])->get();
```

本地作用域

本地作用域允许我们定义通用的约束集合以便在应用中复用。例如，你可能经常需要获取最受欢迎的用户，要定义这样的一个作用域，只需简单在对应 Eloquent 模型方法前加上一个 `scope` 前缀。

作用域总是返回查询构建器实例：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 只包含活跃用户的查询作用域
     *
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopePopular($query)
    {
        return $query->where('votes', '>', 100);
    }

    /**
     * 只包含激活用户的查询作用域
     *
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopeActive($query)
    {
        return $query->where('active', 1);
    }
}
```

使用本地作用域

作用域被定义好了之后，就可以在查询模型的时候调用作用域方法，但调用时不需要加上 `scope` 前缀，你甚至可以同时调用多个作用域，例如：

```
$users = App\User::popular()->active()->orderBy('created_at')->get();
```

动态作用域

有时候你可能想要定义一个可以接收参数的作用域，你只需要将额外的参数添加到你的作用域即可。作用域参数应该被定义在 `$query` 参数之后：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 让查询只包含给定类型的用户
     *
     * @param \Illuminate\Database\Eloquent\Builder $query
     * @param mixed $type
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopeOfType($query, $type)
    {
```

```

        return $query->where('type', $type);
    }
}

```

现在，你可以在调用作用域时传递参数了：

```
$users = App\User::ofType('admin')->get();
```

事件

Eloquent 模型可以触发事件，允许你在模型生命周期中的多个时间点调用如下这些方法：

`retrieved, creating, created, updating, updated, saving, saved, deleting, deleted, restoring, restored`。事件允许你在一个指定模型类每次保存或更新的时候执行代码。

`retrieved` 事件会在从数据库中获取已存在模型时触发。当一个新模型被首次保存的时候，`creating` 和 `created` 事件会被触发。如果一个模型已经在数据库中存在并调用 `save` 方法，`updating/updated` 事件会被触发，无论是创建还是更新，`saving/saved` 事件都会被触发。

举个例子，在 Eloquent 模型中定义一个 `$dispatchesEvents` 属性来映射模型生命周期中多个时间点与对应事件类：

```

<?php

namespace App;

use App\Events\UserSaved;
use App\Events\UserDeleted;
use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * The event map for the model.
     *
     * @var array
     */
    protected $dispatchesEvents = [
        'saved' => UserSaved::class,
        'deleted' => UserDeleted::class,
    ];
}

```

观察者

如果你在给定模型中监听多个事件，可以使用观察者来对所有监听器进行分组，观察者类拥有反射你想要监听的 Eloquent 事件对应的方法名，每个方法接收模型作为唯一参数。Laravel 并没有为观察者提供默认目录，所以你可以创建任意目录来存放观察者类：

```

<?php

namespace App\Observers;

use App\User;

class UserObserver
{
    /**
     * 监听用户创建事件.
     *
     * @param User $user
     * @return void
     */
    public function created(User $user)
    {
        //
    }

    /**
     * 监听用户删除事件.
     *
     * @param User $user
     * @return void
     */

```

```
public function deleting(User $user)
{
    //
}
}
```

要注册观察者，使用你想要观察模型的 `observe` 方法，你可以在某个服务提供者的 `boot` 方法中注册观察者，在本例中，我们在 `AppServiceProvider` 中注册观察者：

```
<?php

namespace App\Providers;

use App\User;
use App\Observers\UserObserver;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        User::observe(UserObserver::class);
    }

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

关联关系

简介

数据表经常要与其它表做关联，比如一篇博客文章可能有很多评论，或者一个订单会被关联到下单用户，Eloquent 让组织和处理这些关联关系变得简单，并且支持多种不同类型的关联关系：

- 一对一
- 一对多
- 多对多
- 远层一对多
- 多态关联
- 多对多的多态关联

定义关联关系

Eloquent 关联关系以 Eloquent 模型类方法的方式定义。和 Eloquent 模型本身一样，关联关系也是强大的查询构建器，定义关联关系为方法可以提供功能强大的方法链和查询能力。例如，我们可以添加更多约束条件到 `posts` 关联关系：

```
$user->posts()->where('active', 1)->get();
```

不过，在深入使用关联关系之前，让我们先学习如何定义每种关联类型。

一对一

一对一关联是一个非常简单的关联关系，例如，一个 `User` 模型有一个与之关联的 `Phone` 模型。要定义这种关联关系，我们需要将 `phone` 方法置于 `User` 模型中，`phone` 方法会调用 `Illuminate\Database\Eloquent\Concerns\HasRelationships` trait 中的 `hasOne` 方法并返回其结果：

```
<?php

namespace App;
```

```
use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 获取关联到用户的手机
     */
    public function phone()
    {
        return $this->hasOne('App\Phone');
    }
}
```

传递给 `hasOne` 方法的第一个参数是关联模型的名称，关联关系被定义后，我们可以使用 Eloquent 的动态属性获取关联记录。动态属性允许我们访问关联方法，就像它们是定义在模型上的属性一样：

```
$phone = User::find(1)->phone;
```

Eloquent 默认关联关系的外键基于模型名称，在本例中，`Phone` 模型默认有一个 `user_id` 外键，如果你希望覆盖这种约定，可以传递第二个参数到 `hasOne` 方法：

```
return $this->hasOne('App\Phone', 'foreign_key');
```

此外，Eloquent 假设外键应该在父级上有一个与之匹配的 `id`（或者自定义 `$primaryKey`），换句话说，Eloquent 将会通过 `user` 表的 `id` 值去 `phone` 表中查询 `user_id` 与之匹配的 `Phone` 记录。如果你想要关联关系使用其他值而不是 `id`，可以传递第三个参数到 `hasOne` 来指定自定义的主键：

```
return $this->hasOne('App\Phone', 'foreign_key', 'local_key');
```

我们通过传递完整参数改写上述示例代码就是：

```
return $this->hasOne('App\Phone', 'user_id', 'id');
```

定义相对的关联

我们可以从 `User` 中访问 `Phone` 模型，相应地，也可以在 `Phone` 模型中定义关联关系从而让我们可以拥有该手机的 `User`。我们可以使用 `belongsTo` 方法定义与 `hasOne` 关联关系相对的关联：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Phone extends Model{
    /**
     * 获取拥有该手机的用户
     */
    public function user()
    {
        return $this->belongsTo('App\User');
    }
}
```

在上面的例子中，Eloquent 默认将会尝试通过 `Phone` 模型的 `user_id` 去 `User` 模型查找与之匹配的记录。Eloquent 通过在关联关系方法名后加 `_id` 后缀来生成默认的外键名。不过，如果 `Phone` 模型上的外键不是 `user_id`，也可以将自定义的键名作为第二个参数传递到 `belongsTo` 方法：

```
/**
 * 获取手机对应的用户
 */
public function user(){
    return $this->belongsTo('App\User', 'foreign_key');
}
```

如果父模型不使用 `id` 作为主键，或者你希望使用别的数据列来连接子模型，可以将父表自定义键作为第三个参数传递给 `belongsTo` 方法：

```
/**
 * 获取手机对应的用户
 */
public function user(){
    return $this->belongsTo('App\User', 'foreign_key', 'other_key');
}
```

同样，我们通过传递完整的参数来改写上述示例代码：

```
return $this->belongsTo('App\User', 'user_id', 'id');
```

默认模型

`belongsTo` 关联关系允许你在给定关联关系为 `null` 的情况下定义一个默认的返回模型，我们将这种模式称之为 **空对象模式**，使用这种模式的好处是不用在代码中编写大量的判断检查逻辑。在下面的例子中，`user` 关联将会在没有用户与文章关联的情况下返回一个空的 `App\User` 模型：

```
/**
 * 获取文章作者
 */
public function user()
{
    return $this->belongsTo('App\User')->withDefault();
}
```

要通过属性填充默认的模型，可以传递数据或闭包到 `withDefault` 方法：

```
/**
 * 获取文章作者
 */
public function user()
{
    return $this->belongsTo('App\User')->withDefault([
        'name' => 'Guest Author',
    ]);
}

/**
 * 获取文章作者
 */
public function user()
{
    return $this->belongsTo('App\User')->withDefault(function ($user) {
        $user->name = 'Guest Author';
    });
}
```

一对多

“一对多”关联是用于定义单个模型拥有多个其它模型的关联关系。例如，一篇博客文章拥有多条评论，和其他关联关系一样，一对多关联通过在 Eloquent 模型中定义方法来定义：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model{
    /**
     * 获取博客文章的评论
     */
    public function comments()
    {
        return $this->hasMany('App\Comment');
    }
}
```

记住，Eloquent 会自动判断 `Comment` 模型的外键，为方便起见，Eloquent 将拥有者模型名称加上 `_id` 后缀作为外键。因此，在本例中，Eloquent 假设 `Comment` 模型上的外键是 `post_id`。

关联关系被定义后，我们就可以通过访问 `comments` 属性来访问评论集合。由于 Eloquent 提供了“动态属性”，我们可以像访问模型的属性一样访问关联方法：

```
$comments = App\Post::find(1)->comments;

foreach ($comments as $comment) {
    //
}
```

当然，由于所有关联同时也是查询构建器，我们可以添加更多的条件约束到通过调用 `comments` 方法获取到的评论上：

```
$comments = App\Post::find(1)->comments()->where('title', 'foo')->first();
```

和 `hasOne` 方法一样，你还可以通过传递额外参数到 `hasMany` 方法来重新设置外键和本地主键：

```
return $this->hasMany('App\Comment', 'foreign_key');
return $this->hasMany('App\Comment', 'foreign_key', 'local_key');
```

```
// 在本例中，传递完整参数代码如下
return $this->hasMany('App\Comment', 'post_id', 'id');
```

一对多（逆向）

现在我们可以访问文章的所有评论了，接下来让我们定义一个关联关系允许通过评论访问所属文章。要定义与 `hasMany` 相对的关联关系，需要在子模型中定义一个关联方法去调用 `belongsTo` 方法：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model{
    /**
     * 获取评论对应的博客文章
     */
    public function post()
    {
        return $this->belongsTo('App\Post');
    }
}
```

关联关系定义好之后，我们可以通过访问动态属性 `post` 来获取某条 `Comment` 对应的 `Post`：

```
$comment = App\Comment::find(1);
echo $comment->post->title;
```

在上面这个例子中，Eloquent 尝试匹配 `Comment` 模型的 `post_id` 与 `Post` 模型的 `id`，Eloquent 通过关联方法名加上 `_id` 后缀生成默认外键，当然，你也可以通过传递自定义外键名作为第二个参数传递到 `belongsTo` 方法，如果你的外键不是 `post_id`，或者你想自定义的话：

```
/**
 * 获取评论对应的博客文章
 */
public function post(){
    return $this->belongsTo('App\Post', 'foreign_key');
}
```

如果你的父模型不使用 `id` 作为主键，或者你希望通过其他数据列来连接子模型，可以将自定义键名作为第三个参数传递给 `belongsTo` 方法：

```
/**
 * 获取评论对应的博客文章
 */
public function post(){
    return $this->belongsTo('App\Post', 'foreign_key', 'other_key');
}
```

类似的，通过传递完整参数改写上述调用代码如下：

```
return $this->belongsTo('App\Post', 'post_id', 'id');
```

多对多

多对多关联比 `hasOne` 和 `hasMany` 关联关系要稍微复杂一些。这种关联关系的一个例子就是在权限管理中，一个用户可能有多个角色，同时一个角色可能被多个用户共用。例如，很多用户可能都有一个“Admin”角色。要定义这样的关联关系，需要三张数据表：`users`、`roles` 和 `role_user`，`role_user` 表按照关联模型名的字母顺序命名，并且包含 `user_id` 和 `role_id` 两个列。

多对多关联通过编写调用 `belongsToMany` 方法返回结果的方式来定义，例如，我们在 `User` 模型上定义 `roles` 方法：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 用户角色
     */
    public function roles()
    {
        return $this->belongsToMany('App\Role');
```

```

    }
}
```

关联关系被定义之后，可以使用动态属性 `roles` 来访问用户的角色：

```
$user = App\User::find(1);

foreach ($user->roles as $role) {
    //
}
```

当然，和所有其它关联关系类型一样，你可以调用 `roles` 方法来添加条件约束到关联查询上：

```
$roles = App\User::find(1)->roles()->orderBy('name')->get();
```

正如前面所提到的，为了确定关联关系连接表的表名，Eloquent 以字母顺序连接两个关联模型的名字。不过，你可以重写这种约定 —— 通过传递第二个参数到 `belongsToMany` 方法：

```
return $this->belongsToMany('App\Role', 'user_roles');
```

除了自定义连接表的表名，你还可以通过传递额外参数到 `belongsToMany` 方法来自定义该表中字段的列名。第三个参数是你定义关联关系模型的外键名称，第四个参数你要连接到的模型的外键名称：

```
return $this->belongsToMany('App\Role', 'user_roles', 'user_id', 'role_id');
```

定义相对的关联关系

要定义与多对多关联相对的关联关系，只需在关联模型中调用一下 `belongsToMany` 方法即可。我们在 `Role` 模型中定义 `users` 方法：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Role extends Model{
    /**
     * 角色用户
     */
    public function users()
    {
        return $this->belongsToMany('App\User');
    }
}
```

正如你所看到的，定义的关联关系和与其对应的 `User` 中定义的一模一样，只是前者引用 `App\Role`，后者引用 `App\User`，由于我们再次使用了 `belongsToMany` 方法，所有的常用表和键自定义选项在定义与多对多相对的关联关系时都是可用的。

获取中间表字段

正如你已经了解到的，处理多对多关联要求一个中间表。Eloquent 提供了一些有用的方法来与这个中间表进行交互，例如，我们假设 `User` 对象有很多与之关联的 `Role` 对象，访问这些关联关系之后，我们可以使用这些模型上的 `pivot` 属性访问中间表：

```
$user = App\User::find(1);

foreach ($user->roles as $role) {
    echo $role->pivot->created_at;
}
```

注意我们获取到的每一个 `Role` 模型都被自动赋上了 `pivot` 属性。该属性包含一个代表中间表的模型，并且可以像其它 Eloquent 模型一样使用。

默认情况下，只有模型主键才能用在 `pivot` 对象上，如果你的 `pivot` 表包含额外的属性，必须在定义关联关系时进行指定：

```
return $this->belongsToMany('App\Role')->withPivot('column1', 'column2');
```

如果你想要你的 `pivot` 表自动包含 `created_at` 和 `updated_at` 时间戳，在关联关系定义时使用 `withTimestamps` 方法：

```
return $this->belongsToMany('App\Role')->withTimestamps();
```

自定义 `pivot` 属性名

上面已经提到，我们可以通过在模型上使用 `pivot` 属性来访问中间表字段，此外，我们还可以在应用中自定义这个属性名称来提升可读性。

例如，如果你的应用包含已经订阅播客的用户，那么就会有一个用户与播客之间的多对多关联，在这个例子中，你可能希望将中间表访问器改为 `subscription` 来取代 `pivot`，这可以通过在定义关联关系时使用 `as` 方法来实现：

```
return $this->belongsToMany('App\Podcast')
    ->as('subscription')
    ->withTimestamps();
```

定义好之后，就可以使用自定义的属性名来访问中间表数据了：

```
$users = User::with('podcasts')->get();

foreach ($users->flatMap->podcasts as $podcast) {
    echo $podcast->subscription->created_at;
}
```

通过中间表字段过滤关联关系

你还可以在定义关联关系的时候使用 `wherePivot` 和 `wherePivotIn` 方法过滤 `belongsToMany` 返回的结果集：

```
return $this->belongsToMany('App\Role')->wherePivot('approved', 1);

return $this->belongsToMany('App\Role')->wherePivotIn('priority', [1, 2]);
```

自定义中间表模型

如果你想要定义自定义的模型来表示关联关系中间表，可以在定义关联关系的时候调用 `using` 方法，所有用于表示关联关系中间表的自定义模型都必须继承自 `Illuminate\Database\Eloquent\Relations\Pivot` 类，例如，我们可以定义一个使用 `UserRole` 中间模型的 `Role`：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Relations\Pivot;

class Role extends Pivot
{
    /**
     * The users that belong to the role.
     */
    public function users()
    {
        return $this->belongsToMany('App\User')->using('App\UserRole');
    }
}
```

`UserRole` 继承自 `Pivot` 类：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Relations\Pivot;

class UserRole extends Pivot
{
    //
}
```

远层的一对多

“远层一对多”关联为通过中间关联访问远层的关联关系提供了一个便捷之道。例如，`Country` 模型通过中间的 `User` 模型可能拥有多个 `Post` 模型。在这个例子中，你可以轻易的聚合给定国家的所有文章，让我们看看定义这个关联关系需要哪些表：

```
countries
id - integer
name - string

users
id - integer
country_id - integer
name - string

posts
id - integer
user_id - integer
title - string
```

尽管 `posts` 表不包含 `country_id`，但是 `hasManyThrough` 关联提供了 `$country->posts` 来访问一个国家的所有文章。要执行该查询，Eloquent 在中间表 `$users` 上检查 `country_id`，查找到相匹配的用户 ID 后，通过用户 ID 来查询 `posts` 表。

既然我们已经查看了该关联关系的数据表结构，接下来让我们在 `Country` 模型上进行定义：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Country extends Model{
    /**
     * 获取指定国家的所有文章
     */
    public function posts()
    {
```

```

    return $this->hasManyThrough('App\Post', 'App\User');
}
}

```

第一个传递到 `hasManyThrough` 方法的参数是最终我们希望访问的模型的名称，第二个参数是中间模型名称。

当执行这种关联查询时通常 Eloquent 外键规则会被使用，如果你想要自定义该关联关系的外键，可以将它们作为第三个、第四个参数传递给 `hasManyThrough` 方法。第三个参数是中间模型的外键名，第四个参数是最终模型的外键名，第五个参数是本地主键。

```

class Country extends Model
{
    public function posts()
    {
        return $this->hasManyThrough(
            'App\Post',
            'App\User',
            'country_id', // users 表使用的外键...
            'user_id', // posts 表使用的外键...
            'id', // countries 表主键...
            'id' // users 表主键...
        );
    }
}

```

多态关联

表结构

多态关联允许一个模型在单个关联下属于多个不同模型。例如，假设应用用户既可以对文章进行评论也可以对视频进行评论，使用多态关联，你可以在这两种场景下使用单个 `comments` 表，首先，让我们看看构建这种关联关系需要的表结构：

```

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

```

两个重要的需要注意的字段是 `comments` 表上的 `commentable_id` 和 `commentable_type`。`commentable_id` 列对应 `Post` 或 `Video` 的 ID 值，而 `commentable_type` 列对应所属模型的类名。当访问 `commentable` 关联时，ORM 根据 `commentable_type` 字段来判断所属模型的类型并返回相应模型实例。

模型结构

接下来，让我们看看构建这种关联关系需要在模型中定义什么：

```

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
     * Get all of the owning commentable models.
     */
    public function commentable()
    {
        return $this->morphTo();
    }
}

class Post extends Model
{
}

```

```

    /**
 * Get all of the post's comments.
 */
public function comments()
{
    return $this->morphMany('App\Comment', 'commentable');
}

class Video extends Model
{
    /**
 * Get all of the video's comments.
 */
public function comments()
{
    return $this->morphMany('App\Comment', 'commentable');
}
}

```

获取多态关联

数据表和模型定义好以后，可以通过模型访问关联关系。例如，要访问一篇文章的所有评论，可以使用动态属性 `comments`：

```

$post = App\Post::find(1);

foreach ($post->comments as $comment) {
    //
}

```

你还可以通过调用 `morphTo` 方法从多态模型中获取多态关联的所属对象。在本例中，就是 `Comment` 模型中的 `commentable` 方法。因此，我们可以用动态属性的方式访问该方法：

```

$comment = App\Comment::find(1);

$commentable = $comment->commentable;

```

`Comment` 模型的 `commentable` 关联返回 `Post` 或 `Video` 实例，这取决于哪个类型的模型拥有该评论。

自定义多态类型

默认情况下，Laravel 使用完全限定类名（包含命名空间的完整类名）来存储关联模型的类型。举个例子，上面示例中的 `Comment` 可能属于某个 `Post` 或 `Video`，默认的 `commentable_type` 可能是 `App\Post` 或 `App\Video`。不过，有时候你可能需要解除数据库和应用内部结构之间的耦合，这样的情况下，可以定义一个 `morphMap` 关联来告知 Eloquent 为每个模型使用自定义名称替代完整类名：

```

use Illuminate\Database\Eloquent\Relations\Relation;

Relation::morphMap([
    'posts' => 'App\Post',
    'videos' => 'App\Video',
]);

```

你可以在 `AppServiceProvider` 的 `boot` 方法中注册这个 `morphMap`，如果需要的话，也可以创建一个独立的服务提供者来实现这一功能。

多对多的多态关联

表结构

除了传统的多态关联，还可以定义“多对多”的多态关联，例如，一个博客的 `Post` 和 `Video` 模型可能共享一个 `Tag` 模型的多态关联。使用对多对的多态关联允许你在博客文章和视频之间有唯一的标签列表。首先，让我们看看表结构：

```

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

```

模型结构

接下来，我们准备在模型中定义该关联关系。`Post` 和 `Video` 模型都有一个 `tags` 方法调用 Eloquent 基类的 `morphToMany` 方法：

```
<?php
```

```
namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
     * 获取指定文章所有标签
     */
    public function tags()
    {
        return $this->morphToMany('App\Tag', 'taggable');
    }
}
```

定义相对的关联关系

接下来，在 Tag 模型中，应该为每一个关联模型定义一个方法，例如，我们定义一个 posts 方法和 videos 方法：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Tag extends Model
{
    /**
     * 获取所有分配该标签的文章
     */
    public function posts()
    {
        return $this->morphedByMany('App\Post', 'taggable');
    }

    /**
     * 获取分配该标签的所有视频
     */
    public function videos()
    {
        return $this->morphedByMany('App\Video', 'taggable');
    }
}
```

获取关联关系

定义好数据库和模型后可以通过模型访问关联关系。例如，要访问一篇文章的所有标签，可以使用动态属性 tags：

```
$post = App\Post::find(1);

foreach ($post->tags as $tag) {
    //
}
```

还可以通过访问调用 morphedByMany 的方法名从多态模型中获取多态关联的所属对象。在本例中，就是 Tag 模型中的 posts 或者 videos 方法：

```
$tag = App\Tag::find(1);

foreach ($tag->videos as $video) {
    //
}
```

关联查询

由于 Eloquent 所有关联关系都是通过方法定义，你可以调用这些方法来获取关联关系的实例而不需要再去手动执行关联查询。此外，所有 Eloquent 关联关系类型同时也是 [查询构建器](#)，允许你在最终数据库执行 SQL 之前继续添加条件约束到关联查询上。

例如，假定在一个博客系统中一个 User 模型有很多相关的 Post 模型：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
```

```
class User extends Model{
    /**
     * 获取指定用户的所有文章
     */
    public function posts()
    {
        return $this->hasMany('App\Post');
    }
}
```

你可以像这样查询 `posts` 关联并添加额外的条件约束到该关联关系上：

```
$user = App\User::find(1);
$user->posts()->where('active', 1)->get();
```

你可以在关联关系上使用任何查询构建器提供任何的方法！

关联方法 Vs. 动态属性

如果你不需要添加额外的条件约束到 Eloquent 关联查询，你可以简单通过动态属性来访问关联对象，例如，还是拿 `User` 和 `Post` 模型作为例子，你可以像这样访问用户的所有文章：

```
$user = App\User::find(1);

foreach ($user->posts as $post) {
    //
}
```

动态属性是“懒惰式加载”，意味着当你真正访问它们的时候才会加载关联数据。正因为如此，开发者经常使用 [渴求式加载](#) 来预加载他们知道在加载模型时要被访问的关联关系。渴求式加载有效减少了必须要被执行用以加载模型关联的 SQL 查询。

查询存在的关联关系

访问一个模型的记录的时候，你可能希望基于关联关系是否存在来限制查询结果的数目。例如，假设你想要获取所有至少有一个评论的博客文章，要实现这个功能，可以传递关联关系的名称到 `has` 和 `orHas` 方法：

```
// 获取所有至少有一条评论的文章...
$posts = App\Post::has('comments')->get();
```

你还可以指定操作符和数目来自定义查询：

```
// 获取所有至少有三条评论的文章...
$posts = Post::has('comments', '>=', 3)->get();
```

还可以使用`“.”`来构造嵌套 `has` 语句，例如，你要获取所有至少有一条评论及投票的文章：

```
// 获取所有至少有一条评论获得投票的文章...
$posts = Post::has('comments.votes')->get();
```

如果你需要更强大的功能，可以使用 `whereHas` 和 `orWhereHas` 方法将「`where`」条件放到 `has` 查询上，这些方法允许你添加自定义条件约束到关联关系条件约束，例如检查一条评论的内容：

```
// 获取所有至少有一条评论包含 foo 字样的文章
$posts = Post::whereHas('comments', function ($query) {
    $query->where('content', 'like', 'foo%');
})->get();
```

无关联结果查询

访问一个模型的记录时，你可能需要基于缺失关联关系的模型对查询结果进行限定。例如，假设你想要获取所有没有评论的博客文章，可以传递关联关系名称到 `doesntHave` 和 `orDoesntHave` 方法来实现：

```
$posts = App\Post::doesntHave('comments')->get();
```

如果你需要更多功能，可以使用 `whereDoesntHave` 和 `orWhereDoesntHave` 方法添加更多「`where`」条件到 `doesntHave` 查询，这些方法允许你添加自定义约束条件到关联关系约束，例如检查评论内容：

```
$posts = Post::whereDoesntHave('comments', function ($query) {
    $query->where('content', 'like', 'foo%');
})->get();
```

统计关联模型

如果你想要在不加载关联关系的情况下统计关联结果数目，可以使用 `withCount` 方法，该方法会放置一个 `{relation}_count` 字段到结果模型。例如：

```
$posts = App\Post::withCount('comments')->get();

foreach ($posts as $post) {
    echo $post->comments_count;
}
```

你可以像添加约束条件到查询一样来添加多个关联关系的「计数」：

```
$posts = Post::withCount(['votes', 'comments' => function ($query) {
    $query->where('content', 'like', 'foo%');
}])->get();

echo $posts[0]->votes_count;
echo $posts[0]->comments_count;
```

还可以为关联关系计数结果设置别名，从而允许在一个关联关系上进行多维度计数：

```
$posts = App\Post::withCount([
    'comments',
    'comments as pending_comments' => function ($query) {
        $query->where('approved', false);
    }
])->get();

echo $posts[0]->comments_count;

echo $posts[0]->pending_comments_count;
```

渴求式加载

当以属性方式访问 Eloquent 关联关系的时候，关联关系数据是「懒惰式加载」的，这意味着关联关系数据直到第一次访问的时候才被加载。不过，Eloquent 还可以在查询父级模型的同时「渴求式加载」关联关系。渴求式加载缓解 N+1 查询问题，要阐明 N+1 查询问题，查看关联到 `Author` 的 `Book` 模型：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Book extends Model
{
    /**
     * 获取写这本书的作者
     */
    public function author()
    {
        return $this->belongsTo('App\Author');
    }
}
```

现在，让我们获取所有书及其作者：

```
$books = App\Book::all();

foreach ($books as $book) {
    echo $book->author->name;
}
```

该循环先执行 1 次查询获取表中的所有书，然后另一个查询获取每一本书的作者，因此，如果有 25 本书，要执行 26 次查询：1 次是获取书本身，剩下的 25 次查询是为每一本书获取其作者。

谢天谢地，我们可以使用渴求式加载来减少该操作到 2 次查询。当查询的时候，可以使用 `with` 方法指定应该被渴求式加载的关联关系：

```
$books = App\Book::with('author')->get();

foreach ($books as $book) {
    echo $book->author->name;
}
```

在该操作中，只执行两次查询即可：

```
select * from books
select * from authors where id in (1, 2, 3, 4, 5, ...)
```

渴求式加载多个关联关系

有时候你需要在单个操作中渴求式加载多个不同的关联关系。要实现这个功能，只需要添加额外的参数到 `with` 方法即可：

```
$books = App\Book::with('author', 'publisher')->get();
```

嵌套的渴求式加载

要渴求式加载嵌套的关联关系，可以使用`“.”`语法。例如，我们在一个 Eloquent 语句中渴求式加载所有书的作者及所有作者的个人联系方式：

```
$books = App\Book::with('author.contacts')->get();
```

渴求式加载指定字段

并不是每次获取关联关系时都需要所有字段，因此，Eloquent 允许你在关联查询时指定要查询的字段：

```
$users = App\User::with('author:id,name')->get();
```

注：使用这个特性时，`id` 字段是必须列出的。

带条件约束的渴求式加载

有时候我们希望渴求式加载一个关联关系，但还想为渴求式加载指定更多的查询条件：

```
$users = App\User::with(['posts' => function ($query) {
    $query->where('title', 'like', '%first%');
}])->get();
```

在这个例子中，Eloquent 只渴求式加载 `title` 包含 `first` 的文章。当然，你还可以调用其它 [查询构建器](#) 来自定义渴求式加载操作：

```
$users = App\User::with(['posts' => function ($query) {
    $query->orderBy('created_at', 'desc');
}])->get();
```

懒惰渴求式加载

有时候你需要在父模型已经被获取后渴求式加载一个关联关系。例如，这在你需要动态决定是否加载关联模型时可能很有用：

```
$books = App\Book::all();

if ($someCondition) {
    $books->load('author', 'publisher');
}
```

如果你需要设置更多的查询条件到渴求式加载查询上，可以传递一个包含你想要记载的关联关系数组到 `load` 方法，数组的值应该是接收查询实例的闭包：

```
$books->load(['author' => function ($query) {
    $query->orderBy('published_date', 'asc');
}]);
```

如果想要在关系管理尚未被加载的情况下加载它，可以使用 `loadMissing` 方法：

```
public function format(Book $book)
{
    $book->loadMissing('author');

    return [
        'name' => $book->name,
        'author' => $book->author->name
    ];
}
```

插入 & 更新关联模型

save 方法

Eloquent 为添加新模型到关联关系提供了便捷方法。例如，如果你需要插入新的 `Comment` 到 `Post` 模型，可以从关联关系的 `save` 方法直接插入 `Comment` 而不是手动设置 `Comment` 的 `post_id` 属性：

```
$comment = new App\Comment(['message' => 'A new comment.']);
$post = App\Post::find(1);
$post->comments()->save($comment);
```

注意我们没有用动态属性方式访问 `comments`，而是调用 `comments` 方法获取关联关系实例。`save` 方法会自动添加 `post_id` 值到新的 `Comment` 模型。

如果你需要保存多个关联模型，可以使用 `saveMany` 方法：

```
$post = App\Post::find(1);

$post->comments()->saveMany([
    new App\Comment(['message' => 'A new comment.']),
    new App\Comment(['message' => 'Another comment.']),
]);
```

create 方法

除了 `save` 和 `saveMany` 方法外，还可以使用 `create` 方法，该方法接收属性数组、创建模型、然后插入数据库。`save` 和 `create` 的不同之处在于 `save` 接收整个 Eloquent 模型实例而 `create` 接收原生 PHP 数组：

```
$post = App\Post::find(1);

$comment = $post->comments()->create([
    'message' => 'A new comment.',
]);
```

注：使用 `create` 方法之前确保先浏览属性批量赋值文档。

还可以使用 `createMany` 方法来创建多个关联模型：

```
$post = App\Post::find(1);

$post->comments()->createMany([
    [
        'message' => 'A new comment.',
    ],
    [
        'message' => 'Another new comment.',
    ],
]);
```

从属关联关系

更新 `belongsTo` 关联的时候，可以使用 `associate` 方法，该方法会在子模型设置外键：

```
$account = App\Account::find(10);
$user->account()->associate($account);
$user->save();
```

移除 `belongsTo` 关联的时候，可以使用 `dissociate` 方法。该方法会设置关联关系的外键为 `null`：

```
$user->account()->dissociate();
$user->save();
```

多对多关联

附加/分离

处理多对多关联的时候，Eloquent 还提供了一些额外的辅助函数使得处理关联模型变得更加方便。例如，我们假定一个用户可能有多个角色，同时一个角色属于多个用户，要通过在连接模型的中间表中插入记录附加角色到用户上，可以使用 `attach` 方法：

```
$user = App\User::find(1);
$user->roles()->attach($roleId);
```

附加关联关系到模型，还可以以数组形式传递额外被插入数据到中间表：

```
$user->roles()->attach($roleId, ['expires' => $expires]);
```

当然，有时候有必要从用户中移除角色，要移除一个多对多关联记录，使用 `detach` 方法。`detach` 方法将会从中间表中移除相应的记录；但是，两个模型在数据库中都保持不变：

```
// 从指定用户中移除角色...
$user->roles()->detach($roleId);

// 从指定用户移除所有角色...
$user->roles()->detach();
```

为了方便，`attach` 和 `detach` 还接收数组形式的 ID 作为输入：

```
$user = App\User::find(1);

$user->roles()->detach([1, 2, 3]);

$user->roles()->attach([
    1 => ['expires' => $expires],
    2 => ['expires' => $expires]
]);
```

```
]);
```

同步关联

你还可以使用 `sync` 方法构建多对多关联。`sync` 方法接收数组形式的 ID 并将其放置到中间表。任何不在该数组中的 ID 对应记录将会从中间表中移除。因此，该操作完成后，只有在数组中的 ID 对应记录还存在于中间表：

```
$user->roles()->sync([1, 2, 3]);
```

你还可以和 ID 一起传递额外的中间表值：

```
$user->roles()->sync([1 => ['expires' => true], 2, 3]);
```

如果你不想要删除已存在的 ID，可以使用 `syncWithoutDetaching` 方法：

```
$user->roles()->syncWithoutDetaching([1, 2, 3]);
```

切换关联

多对多关联还提供了一个 `toggle` 方法用于切换给定 ID 的附加状态，如果给定 ID 当前被附加，则取消附加，类似的，如果当前没有附加，则附加：

```
$user->roles()->toggle([1, 2, 3]);
```

在中间表上保存额外数据

处理多对多关联时，`save` 方法接收额外中间表属性数组作为第二个参数：

```
App\>User::find(1)->roles()->save($role, ['expires' => $expires]);
```

更新中间表记录

如果你需要更新中间表中已存在的行，可以使用 `updateExistingPivot` 方法。该方法接收中间记录外键和属性数组进行更新：

```
$user = App\User::find(1);
```

```
$user->roles()->updateExistingPivot($roleId, $attributes);
```

触发父级时间戳更新

当一个模型属于另外一个时，例如 `Comment` 属于 `Post`，子模型更新时父模型的时间戳也被更新将很有用，例如，当 `Comment` 模型被更新时，你可能想要“触发”更新其所属模型 `Post` 的 `updated_at` 时间戳。Eloquent 使得这项操作变得简单，只需要添加包含关联关系名称的 `touches` 属性到子模型即可：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model{
    /**
     * 要触发的所有关联关系
     *
     * @var array
     */
    protected $touches = ['post'];

    /**
     * 评论所属文章
     */
    public function post()
    {
        return $this->belongsTo('App\Post');
    }
}
```

现在，当你更新 `Comment` 时，所属模型 `Post` 将也会更新其 `updated_at` 值，从而方便得知何时更新 `Post` 模型缓存：

```
$comment = App\Comment::find(1);
$comment->text = 'Edit to this comment!';
$comment->save();
```

集合

简介

Eloquent 返回的包含多条记录的结果集都是 `\Illuminate\Database\Eloquent\Collection` 对象的实例，包括通过 `get` 方法或者通过访问关联关系获取的结果。Eloquent 集合对象继承自 Laravel 的 [集合基类](#)，因此很自然的继承了很多处理 Eloquent 模型底层数组的方法。

当然，集合也是迭代器，允许你像 PHP 数组一样对其进行循环：

```
$users = App\User::where('active', 1)->get();

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

不过，集合使用直观的接口提供了各种映射/简化操作，因此比数组更加强大。例如，我们可以通过以下方式移除所有无效的模型并聚合剩下用户的姓名：

```
$users = App\User::where('active', 1)->get();

$names = $users->reject(function ($user) {
    return $user->active === false;
})->map(function ($user) {
    return $user->name;
});
```

注：尽管大多数 Eloquent 集合返回的是一个新的 Eloquent 集合实例，但是 `pluck`、`keys`、`zip`、`collapse`、`flatten` 和 `flip` 方法返回的是 [集合基类](#) 实例。类似地，如果 `map` 操作返回的集合不包含任何 Eloquent 模型，将会被自动转化成集合基类。

可用方法

集合基类

所有的 Eloquent 集合继承自 Laravel 集合对象基类，因此，它们继承所有集合基类提供的强大方法：[集合方法大全](#)。

自定义集合

如果你需要在自己扩展的方法中使用自定义的集合对象，可以重写模型上的 `newCollection` 方法：

```
<?php

namespace App;

use App\CustomCollection;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 创建一个新的 Eloquent 集合实例
     *
     * @param array $models
     * @return \Illuminate\Database\Eloquent\Collection
     */
    public function newCollection(array $models = [])
    {
        return new CustomCollection($models);
    }
}
```

定义好 `newCollection` 方法后，无论何时 Eloquent 返回该模型的 `Collection` 实例你都会获取到自定义的集合。如果你想要在应用中的每一个模型中使用自定义集合，需要在模型基类中重写 `newCollection` 方法。

访问器和修改器

简介

访问器和修改器允许你在获取模型属性或设置其值时格式化 Eloquent 属性。例如，你可能想要使用 [Laravel 加密器](#) 对存储在数据库中的数据进行加密，并且在 Eloquent 模型中访问时自动进行解密。

除了自定义访问器和修改器，Eloquent 还可以自动转换日期字段为 `Carbon` 实例甚至将文本转换为 JSON。

访问器 & 修改器

定义访问器

要定义一个访问器，需要在模型中创建一个 `getFooAttribute` 方法，其中 `Foo` 是你想要访问的字段名（使用驼峰式命名规则）。在本例中，我们将会为 `first_name` 属性定义一个访问器，该访问器在获取 `first_name` 的值时被 Eloquent 自动调用：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 获取用户的名字
     *
     * @param string $value
     * @return string
     */
    public function getFirstNameAttribute($value)
    {
        return ucfirst($value);
    }
}
```

正如你所看到的，该字段的原生值被传递给访问器，然后返回处理过的值。要访问该值只需要简单访问 `first_name` 即可：

```
$user = App\User::find(1);
$firstName = $user->first_name;
```

当然，你也可以使用访问器将已存在的属性转化为全新的、经过处理的值：

```
/**
 * 获取用户的全名
 *
 * @return string
 */
public function getFullNameAttribute()
{
    return "{$this->first_name} {$this->last_name}";
}
```

定义修改器

要定义一个修改器，需要在模型中定义 `setFooAttribute` 方法，其中 `Foo` 是你想要访问的字段（使用驼峰式命名规则）。接下来让我们为 `first_name` 属性定义一个修改器，当我们为模型上的 `first_name` 赋值时该修改器会被自动调用：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 设置用户的名字
     *
     * @param string $value
     * @return string
     */
    public function setFirstNameAttribute($value)
    {
        $this->attributes['first_name'] = strtolower($value);
    }
}
```

该修改器获取要被设置的属性值，允许你操纵该值并设置 Eloquent 模型内部属性值为操作后的值。例如，如果你尝试设置 `Sally` 的 `first_name` 属性：

```
$user = App\User::find(1);
$user->first_name = 'Sally';
```

在本例中，`setFirstNameAttribute` 方法会被调用，传入参数为 `Sally`，修改器会对其调用 `strtolower` 函数并将处理后的值设置为内部属性的值。

日期修改器

默认情况下，Eloquent 将会转化 `created_at` 和 `updated_at` 列的值为 `Carbon` 实例，该类继承自 PHP 原生的 `Datetime` 类，并提供了各种有用的方法。你可以自定义哪些字段被自动调整修改，甚至可以通过重写模型中的 `$dates` 属性完全禁止调整：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 应该被调整为日期的属性
     *
     * @var array
     */
    protected $dates = [
        'created_at',
        'updated_at',
        'disabled_at'
    ];
}
```

如果字段是日期格式时，你可以将其值设置为 UNIX 时间戳，日期字符串（`Y-m-d`），日期-时间字符串，`Datetime/Carbon` 实例，日期的值将会自动以正确格式存储到数据库中：

```
$user = App\User::find(1);
$user->disabled_at = Carbon::now();
$user->save();
```

正如上面提到的，当获取被罗列在 `$dates` 数组中的属性时，它们会被自动转化为 `Carbon` 实例，并允许你在属性上使用任何 `Carbon` 类的方法：

```
$user = App\User::find(1);
return $user->disabled_at->getTimestamp();
```

日期格式化

默认情况下，时间戳的格式是 '`Y-m-d H:i:s`'，如果你需要自定义时间戳格式，在模型中设置 `$dateFormat` 属性，该属性决定日期属性存储在数据库以及序列化为数组或 JSON 时的格式：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * 模型日期的存储格式
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

属性转换

模型中的 `$casts` 属性为属性字段转换到通用数据类型提供了便利方法。`$casts` 属性是数组格式，其键是要被转换的属性名称，其值时你想要转换的类型。目前支持的转换类型包括：`integer`, `real`, `float`, `double`, `string`, `boolean`, `object`, `array`, `collection`, `date`, `datetime` 和 `timestamp`。

例如，让我们转换 `is_admin` 属性，将其由 `integer` 值（0 或 1）转换为 `boolean` 值：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
```

```
{
    /**
     * 应该被转化为原生类型的属性
     *
     * @var array
     */
    protected $casts = [
        'is_admin' => 'boolean',
    ];
}
```

现在, `is_admin` 属性在被访问时总是被转换为 `boolean`, 即使底层存储在数据库中的值是 `integer`:

```
$user = App\User::find(1);

if ($user->is_admin) {
    //
}
```

数组 & JSON 转换

`array` 类型转换在处理被存储为序列化 JSON 格式的字段时特别有用, 例如, 如果数据库有一个 `JSON` 或 `TEXT` 字段类型包含了序列化 JSON, 添加 `array` 类型转换到该属性将会在 Eloquent 模型中访问其值时自动将其反序列化为 PHP 数组:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 应该被转化为原生类型的属性
     *
     * @var array
     */
    protected $casts = [
        'options' => 'array',
    ];
}
```

类型转换被定义后, 访问 `options` 属性将会自动从 JSON 反序列化为 PHP 数组, 反之, 当你设置 `options` 属性的值时, 给定数组将会自动转化为 JSON 以供存储:

```
$user = App\User::find(1);
$options = $user->options;
$options['key'] = 'value';
$user->options = $options;
$user->save();
```

API 资源类

简介

构建 API 时, 在 Eloquent 模型和最终返回给应用用户的 JSON 响应之间可能需要一个转化层。Laravel 的资源类允许你以简单优雅的方式将模型和模型集合转化为 JSON 格式数据。

生成资源类

要生成一个资源类, 可以使用 Artisan 命令 `make:resource`, 默认情况下, 资源类存放在应用的 `app/Http/Resources` 目录下, 资源类都继承自 `Illuminate\Http\Resources\Json\Resource` 基类:

```
php artisan make:resource UserResource
```

资源集合

除了生成转化独立模型的资源类之外, 还可以生成转化模型集合的资源类。这样响应就可以包含链接和其他与整个给定资源集合相关的元信息。

要创建一个资源集合处理类, 需要在创建资源类的时候使用 `--collection` 标记, 或者在资源名称中包含单词 `Collection` 以便告知 Laravel 需要创建一个资源集合类, 资源集合类继承自 `Illuminate\Http\Resources\Json\ResourceCollection` 基类:

```
php artisan make:resource Users --collection
php artisan make:resource UserCollection
```

核心概念

注：这是一个关乎资源和资源集合的高屋建瓴的概述，强烈推荐你阅读本文档的其他部分来深入强化理解资源类提供的功能和定制化。在深入了解资源类提供的所有功能之前，我们先来高屋建瓴的看一下如何在 Laravel 中使用资源类。一个资源类表示一个单独的需要被转化为 JSON 数据结构的模型，例如，下面是一个简单的 `UserResource` 类：

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\Resource;

class UserResource extends Resource
{
    /**
     * Transform the resource into an array.
     *
     * @param \Illuminate\Http\Request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
}
```

每一个资源类都包含一个 `toArray` 方法用来返回在发送响应时需要被转化 JSON 的属性数组，注意这里我们可以通过 `$this` 变量直接访问模型属性，这是因为资源类是一个代理，可以访问底层对应模型提供的属性和方法。资源类定义好之后，可以从路由或控制器返回：

```
use App\User;
use App\Http\Resources\UserResource;

Route::get('/user', function () {
    return new UserResource(User::find(1));
});
```

资源集合

如果你需要返回资源集合或者分页响应，可以在路由或控制器中创建资源实例时使用 `collection` 方法：

```
use App\User;
use App\Http\Resources\UserResource;

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

当然，这种方式不能添加除模型数据之外的其他需要和集合一起返回的元数据，如果你想要自定义资源集合响应，可以创建专用的资源类来表示集合：

```
php artisan make:resource UserCollection
```

资源集合类生成之后，可以很轻松地定义需要包含到响应中的元数据：

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @param \Illuminate\Http\Request
     */
```

```

 * @return array
 */
public function toArray($request)
{
    return [
        'data' => $this->collection,
        'links' => [
            'self' => 'link-value',
        ],
    ];
}

```

定义好资源集合类之后，就可以从路由或控制器中返回它：

```

use App\User;
use App\Http\Resources\UserCollection;

Route::get('/users', function () {
    return new UserCollection(User::all());
});

```

编写资源类

注：如果你还没有阅读[核心概念](#)，强烈建议你在继续下去之前阅读那部分文档。

其实资源类很简单，它们所做的只是将给定模型转化为数组，因此，每个资源类都包含 `toArray` 方法，用于将模型属性转化为一个可以返回给用户的、API 友好的数组：

```

<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\Resource;

class UserResource extends Resource
{
    /**
     * Transform the resource into an array.
     *
     * @param \Illuminate\Http\Request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
}

```

定义好资源类之后，就可以从路由或控制器中将其直接返回：

```

use App\User;
use App\Http\Resources\UserResource;

Route::get('/user', function () {
    return new UserResource(User::find(1));
});

```

关联关系

如果你想要在响应中包含关联资源，可以将它们添加到 `toArray` 方法返回的数组。在本例中，我们使用 `Post` 资源类的 `collection` 方法添加用户博客文章到资源响应：

```

/**
 * 将资源转化为数组
 *
 * @param \Illuminate\Http\Request
 * @return array
 */
public function toArray($request)

```

```
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'posts' => Post::collection($this->posts),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

注：如果你想要在关联关系被加载之后包含它们，请查看[带条件的关联关系](#)部分文档。

资源集合

资源类用于将单个模型转化为数组，而资源集合类用于将模型集合转化为数组。并不是每种类型的模型都需要定义一个资源集合类，因为所有资源类都提供了一个 `collection` 方法立马生成一个特定的资源集合：

```
use App\User;
use App\Http\Resources\UserResource;

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

不过，如果你需要自定义与集合一起返回的元数据，就需要定义一个资源集合类了：

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @param \Illuminate\Http\Request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'data' => $this->collection,
            'links' => [
                'self' => 'link-value',
            ],
        ];
    }
}
```

和单个资源类一样，资源集合类也可以从路由或控制器中直接返回：

```
use App\User;
use App\Http\Resources\UserCollection;

Route::get('/users', function () {
    return new UserCollection(User::all());
});
```

数据包装

默认情况下，在资源响应被转化为 JSON 的时候最外层的资源都会包裹到一个 `data` 键里，例如，一个典型的资源集合响应数据如下所示：

```
{
    "data": [
        {
            "id": 1,
            "name": "Eladio Schroeder Sr.",
            "email": "therese28@example.com",
        },
        {
            "id": 2,
            "name": "Liliana Mayert",
            "email": "evandervort@example.com",
        }
    ]
}
```

```
]
}
```

如果你想要禁止包装最外层资源，可以调用资源基类提供的 `withoutWrapping` 方法，通常，你需要在 `AppServiceProvider` 或者其他每个请求都会加载的服务提供者中调用这个方法：

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Http\Resources\Json\Resource;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Resource::withoutWrapping();
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

注：`withoutWrapping` 只影响最外层资源，并不会移除你手动添加到自己的资源集合类中的 `data` 键。

包装嵌套资源

你完全可以自己决定如何包装资源的关联关系。如果你想要所有资源集合包裹到 `data` 键里，而不管它们之间的嵌套，那么就需要为每个资源定义一个资源集合类并通过 `data` 键返回这个集合。

当然，你可能会担心这样做会不会导致最外层的资源被包装到两个 `data` 键，如果你这样想的话就是完全多虑了，Laravel 永远不会让资源出现双层包装，所以你大可不必担心转化资源集合的嵌套问题：

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class CommentsCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @param \Illuminate\Http\Request
     * @return array
     */
    public function toArray($request)
    {
        return ['data' => $this->collection];
    }
}
```

数据包装和分页

在资源响应中返回分页集合时，Laravel 会把资源数据包装到 `data` 键里，即使调用了 `withoutWrapping` 方法也不例外。这是因为分页响应总是会包含带有分页器状态信息的 `meta` 和 `links` 键：

```
{
    "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
}
}
}

```

分页

你可能经常需要传递分页器实例到资源类的 `collection` 方法或者自定义的资源集合类:

```

use App\User;
use App\Http\Resources\UserCollection;

Route::get('/users', function () {
    return new UserCollection(User::paginate());
});

```

分页响应总是会包含带有分页器状态信息的 `meta` 和 `links` 键:

```

{
    "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
    }
}

```

带条件的属性

有时候你可能希望只有在满足给定条件的情况下才在资源响应中包含某个属性。例如，你可能希望只有在当前用户是管理员的情况下才包含某个值。为此，Laravel 提供了多个辅助函数来帮助你实现这种功能，`when` 方法就可以用于在满足某种条件的前提下添加属性到资源响应:

```

/**
 * 将资源转化为数组
 *
 * @param \Illuminate\Http\Request

```

```
* @return array
*/
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'secret' => $this->when($this->isAdmin(), 'secret-value'),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

在这个例子中，`secret` 键只有在 `$this->isAdmin()` 方法返回 `true` 的前提下才会出现在最终资源响应中。如果这个方法返回 `false`，`secret` 键将会在资源响应发送给客户端之前从返回数据中完全移除。`when` 方法让你在任何时候都可以优雅地定义资源类，而不必在构建数组时重新编写条件语句。

`when` 方法还可以接受闭包作为第二个参数，从而允许你在给定条件为 `true` 的时候计算返回值：

```
'secret' => $this->when($this->isAdmin(), function () {
    return 'secret-value';
}),
```

注：记住，在资源类上的方法调用实际上代理的是底层模型实例方法，所以，在这个示例中，`isAdmin` 方法调用的实际上底层 `User` 模型上的方法。

合并带条件的属性

有时候你可能有多个属性基于同一条件才会包含到资源响应中，在这种情况下，你可以使用 `mergeWhen` 方法在给定条件为 `true` 的前提下来包含这些属性到响应中：

```
/**
 * Transform the resource into an array.
 *
 * @param \Illuminate\Http\Request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        $this->mergeWhen($this->isAdmin(), [
            'first-secret' => 'value',
            'second-secret' => 'value',
        ]),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

再次说明，如果给定条件为 `false`，这些属性将会在资源响应发送给客户端之前从响应数据中完全移除。

注：`mergeWhen` 方法不能在混合字符串和数字键的数组中使用，此外，也不能在没有顺序排序的纯数字键数组中使用。

带条件的关联关系

除了通过条件加载属性之外，还可以基于给定关联关系是否在模型上加载的条件在资源响应中包含关联关系。这样的话在控制器中就可以决定哪些关联关系需要被加载，然后资源类就可以在关联关系确实已经被加载的情况下很轻松地包含它们。

最后，这种机制也让我们在资源类中避免 `N+1` 查询问题变得简单。`whenLoaded` 方法可用于带条件的加载关联关系。为了避免不必要的关联关系加载，该方法接收关联关系名称而不是关联关系本身：

```
/**
 * Transform the resource into an array.
 *
 * @param \Illuminate\Http\Request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'posts' => Post::collection($this->whenLoaded('posts')),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

```
}
```

在这个例子中，如果关联关系还没有被加载，`posts` 键将会在资源响应发送到客户端之前从响应数据中移除。

带条件的中间表信息

除了在资源响应中带条件的包含关联关系信息之外，你还可以使用 `whenPivotLoaded` 方法从多对多关联关系中间表中引入满足条件的数据。

`whenPivotLoaded` 方法接收中间表名称作为第一个参数，第二个参数是一个闭包，该闭包定义了如果模型对应中间表信息有效的情况下返回的数据：

```
/**
 * Transform the resource into an array.
 *
 * @param \Illuminate\Http\Request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'expires_at' => $this->whenPivotLoaded('role_users', function () {
            return $this->pivot->expires_at;
        }),
    ];
}
```

添加元数据

有些 JSON API 标准要求添加额外的元数据到资源响应或资源集合响应。这些元数据通常包含链接到资源或关联资源的 `links`，或者关于资源本身的元数据。如果你需要返回关于资源的额外元数据，可以在 `toArray` 方法中包含它们。例如，你可以在转化资源集合时引入 `links` 信息：

```
/**
 * Transform the resource into an array.
 *
 * @param \Illuminate\Http\Request
 * @return array
 */
public function toArray($request)
{
    return [
        'data' => $this->collection,
        'links' => [
            'self' => 'link-value',
        ],
    ];
}
```

从资源类返回额外元数据的场景下，在返回分页响应时永远不必担心意外覆盖 Laravel 自动添加的 `links` 或 `meta` 键，任何自定义的额外 `links` 键都会被合并到分页器提供的链接中。

顶层元数据

有时候你可能希望只有在资源是最外层被返回数据的情况下包含特定元数据。通常情况下，这将会包含关于整个响应的元数据，要定义这个元数据，需要添加一个 `with` 方法到你的资源类。该方法只有在资源是最外层被渲染数据的情况下才会返回一个被包含到资源响应中的元数据数组：

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @param \Illuminate\Http\Request
     * @return array
     */
    public function toArray($request)
    {
        return parent::toArray($request);
    }

    /**
     * Get additional data that should be returned with the resource array.
     *
     * @param \Illuminate\Http\Request $request
     */
}
```

```
* @return array
*/
public function with($request)
{
    return [
        'meta' => [
            'key' => 'value',
        ],
    ];
}
```

构造资源类时添加元数据

你还可以在路由或控制器中构造资源类实例时添加顶层数据，在所有资源类中都有效的 `additional` 方法，可用于添加数组数据到资源响应：

```
return (new UserCollection(User::all() -> load('roles')))
    -> additional(['meta' => [
        'key' => 'value',
    ]]);
```

资源响应

正如你所了解到的，资源类可以直接从路由和控制器中返回：

```
use App\User;
use App\Http\Resources\UserResource;

Route::get('/user', function () {
    return new UserResource(User::find(1));
});
```

不过，有时候你可能需要在响应发送到客户端之前自定义输出 HTTP 响应。有两种方法来实现这个功能，第一种是链接 `response` 方法到资源类，该方法会返回一个 `Illuminate\Http\Response` 实例，从而允许你完全控制响应头：

```
use App\User;
use App\Http\Resources\UserResource;

Route::get('/user', function () {
    return (new UserResource(User::find(1)))
        -> response()
        -> header('X-Value', 'True');
});
```

另一种方法是在资源类中定义一个 `withResponse` 方法，该方法会在资源在响应中作为最外层数据返回时被调用：

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\Resource;

class UserResource extends Resource
{
    /**
     * Transform the resource into an array.
     *
     * @param \Illuminate\Http\Request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'id' => $this->id,
        ];
    }

    /**
     * Customize the outgoing response for the resource.
     *
     * @param \Illuminate\Http\Request
     * @param \Illuminate\Http\Response
     * @return void
     */
    public function withResponse($request, $response)
    {
```

```

    $response->header('X-Value', 'True');
}
}

```

序列化

简介

当构建 JSON API 时，经常需要转化模型和关联关系为数组或 JSON。Eloquent 提供了便捷方法以便实现这些转换，以及控制哪些属性被包含到序列化中。

序列化模型 & 集合

序列化为数组

要转化模型及其加载的关联关系为数组，可以使用 `toArray` 方法。这个方法是递归的，所以所有属性及其关联对象属性（包括关联的关联）都会被转化为数组：

```

$user = App\User::with('roles')->first();
return $user->toArray();

```

还可以转化整个模型集合为数组：

```

$users = App\User::all();
return $users->toArray();

```

序列化为 JSON

要转化模型为 JSON，可以使用 `toJson` 方法，和 `toArray` 一样，`toJson` 方法也是递归的，所有属性及其关联属性都会被转化为 JSON。还可以指定 PHP 支持的 JSON 编码选项：

```

$user = App\User::find(1);
return $user->toJson();
return $user->toJson(JSON_PRETTY_PRINT);

```

你还可以转化模型或集合为字符串，这将会自动调用 `toJson` 方法：

```

$user = App\User::find(1);
return (string) $user;

```

由于模型和集合在转化为字符串的时候会被转化为 JSON，你可以从应用的路由或控制器中直接返回 Eloquent 对象：

```

Route::get('users', function() {
    return App\User::all();
});

```

在 JSON 中隐藏属性

有时候你希望在模型数组或 JSON 显示中隐藏某些属性，比如密码，要实现这个功能，在定义模型的时候设置 `$hidden` 属性：

```

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 在数组中隐藏的属性
     *
     * @var array
     */
    protected $hidden = ['password'];
}

```

注：如果要隐藏关联关系，使用关联关系的方法名，而不是动态属性名。

此外，可以使用 `$visible` 属性来定义模型数组和 JSON 显示的属性白名单：

```
<?php
```

```
namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 在数组中显示的属性
     *
     * @var array
     */
    protected $visible = ['first_name', 'last_name'];
}
```

临时暴露隐藏属性

如果你想要在特定模型中临时显示隐藏的属性，可以使用 `makeVisible` 方法，该方法以方法链的调用方式返回模型实例：

```
return $user->makeVisible('attribute')->toArray();
```

类似的，如果你想要隐藏给定模型实例上某些显示的属性，可以使用 `makeHidden` 方法：

```
return $user->makeHidden('attribute')->toArray();
```

追加值到 JSON

有时候，需要添加数据库中没有的字段到数组中，要实现这个功能，首先要为这个值定义一个访问器：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 为用户获取管理员标识
     *
     * @return bool
     */
    public function getIsAdminAttribute()
    {
        return $this->attributes['admin'] == 'yes';
    }
}
```

定义好访问器后，添加字段名到该模型的 `appends` 属性。需要注意的是，尽管访问器使用“camel case”（驼峰）形式定义，属性名通常以“snake case”（下划线）的方式被引用：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 追加到模型数组表单的访问器
     *
     * @var array
     */
    protected $appends = ['is_admin'];
}
```

字段被添加到 `appends` 列表之后，将会被包含到模型数组和 JSON 中，`appends` 数组中的属性还会遵循模型中配置的 `visible` 和 `hidden` 设置。
运行时追加

你可以在单个模型上使用 `append` 方法来追加属性，或者，你可以使用 `setAppends` 方法为给定模型覆盖整个追加属性数组：

```
return $user->append('is_admin')->toArray();

return $user->setAppends(['is_admin'])->toArray();
```

日期序列化

自定义每个属性的日期格式

你可以通过指定[转化声明](#)中的日期格式来自定义单个 Eloquent 日期属性的序列化格式：

```
protected $casts = [
    'birthday' => 'date:Y-m-d',
    'joined_at' => 'datetime:Y-m-d H:00',
];
```

通过 Carbon 全局自定义

Laravel 扩展了 Carbon 日期库以便自定义 Carbon 的 JSON 序列化格式，要自定义所有 Carbon 日期在整个应用中如何被序列化，可以使
用 `Carbon::serializeUsing` 方法。`serializeUsing` 方法接收一个闭包，该闭包返回字符串形式的日期用于 JSON 序列化：

```
<?php

namespace App\Providers;

use Illuminate\Support\Carbon;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Carbon::serializeUsing(function ($carbon) {
            return $carbon->format('U');
        });
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

八、安全系列

登录认证

简介

注：想要快速上手？只需要在新安装的 Laravel 应用下运行 `php artisan make:auth` 和 `php artisan migrate`，这两个命令会生成用户登录注册所需要的所有东西，然后在浏览器中访问 `http://your-app.dev/register` 即可。

Laravel 中实现登录认证非常简单。实际上，几乎所有东西 Laravel 都已经为你配置好了。配置文件位于 `config/auth.php`，其中包含了用于调整认证服务行为的、文档友好的选项配置。

在底层代码中，Laravel 的认证组件由“guards”和“providers”组成，Guard 定义了用户在每个请求中如何实现认证，例如，Laravel 通过 `session guard` 来维护 Session 存储的状态和 Cookie。

Provider 定义了如何从持久化存储中获取用户信息，Laravel 底层支持通过 Eloquent 和数据库查询构建器两种方式来获取用户，如果需要的话，你还可以定义额外的 Provider。

如果看到这些名词觉得云里雾里，大可不必太过担心，因为对绝大多数应用而言，只需使用默认认证配置即可，不需要做什么改动。

学院君注：通俗点说，在进行登录认证的时候，要做两件事，一个是从数据库存取用户数据，一个是把用户登录状态保存起来，在 Laravel 的底层实现中，通过 Provider 存取数据，通过 Guard 存储用户认证信息，前者主要和数据库打交道，后者主要和 Session 打交道（API 例外）。

数据库考量

默认情况下，Laravel 在 `app` 目录下包含了一个 Eloquent 模型 `App\User`，这个模型可以和默认的 Eloquent 认证驱动一起使用。如果你的应用不使用 Eloquent，你可以使用 `database` 认证驱动，该驱动使用 Laravel 查询构建器与数据库交互。

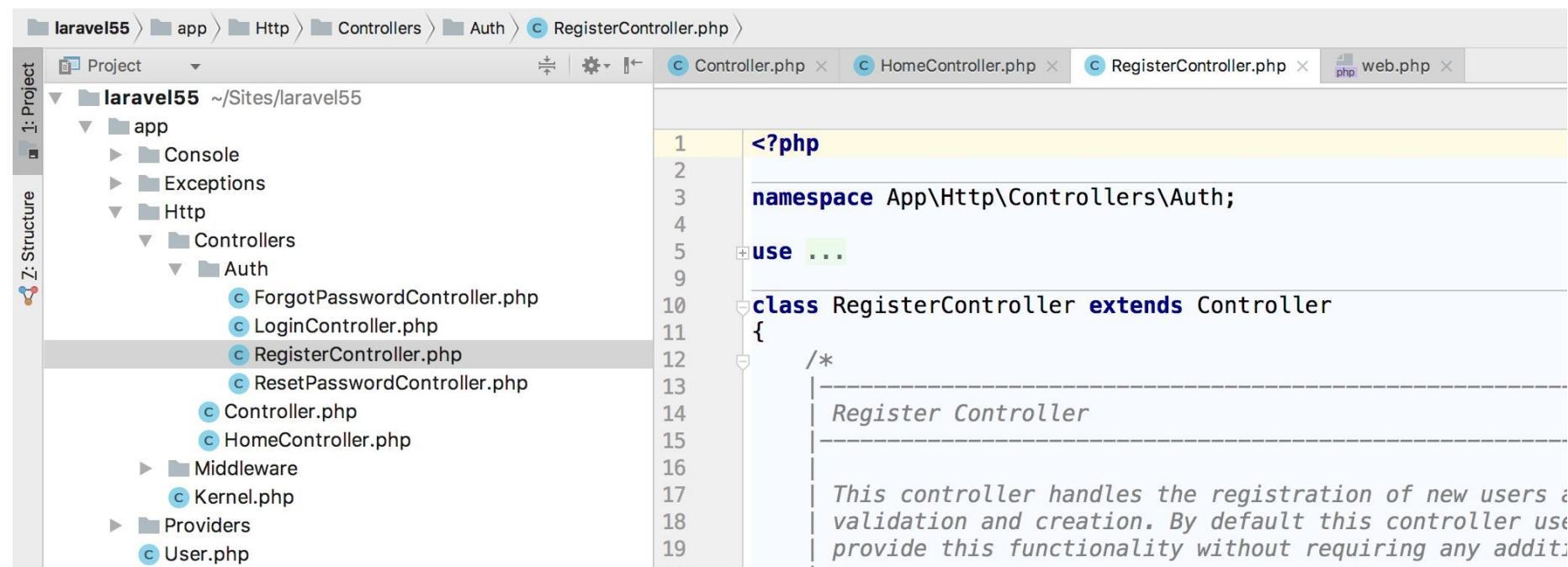
为 `App\User` 模型构建数据库表结构的时候，确保 `password` 字段长度至少有 60 位。保持默认字符串长度（255）是个不错的选择。

还有，你需要验证 `users` 表包含了 `remember_token`，该字段是个可以为空的字符串类型，字段长度为 100，用于在登录时存储应用维护的“记住我”Session 令牌。

TABLES	Field	Type	Length	Unsigned	Zerofill	Binary	Allow Null	Key	Default	Extra	Encoding	Collation	Comment
migrations	id	INT	10	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	PRI	auto_increment	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
password_resets	name	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	UNI	None	<input checked="" type="checkbox"/>	UTF-8 Unicode	utf8mb4_unicode_ci	
users	email	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	UNI	None	<input checked="" type="checkbox"/>	UTF-8 Unicode	utf8mb4_unicode_ci	
	password	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		None	<input checked="" type="checkbox"/>	UTF-8 Unicode	utf8mb4_unicode_ci	
	remember_token	VARCHAR	100	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL	NULL	<input checked="" type="checkbox"/>	UTF-8 Unicode	utf8mb4_unicode_ci	
	created_at	TIMESTAMP		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL	NULL	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
	updated_at	TIMESTAMP		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL	NULL	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

快速入门

Laravel 提供了几个预置的认证控制器，位于 `App\Http\Controllers\Auth` 命名空间下，`RegisterController` 用于处理新用户注册，`LoginController` 用于处理用户登录认证，`ForgotPasswordController` 用于处理重置密码邮件链接，`ResetPasswordController` 包含重置密码逻辑，每个控制器都使用 trait 来引入它们需要的方法。对很多应用而言，你根本不需要修改这些控制器：



```
<?php
namespace App\Http\Controllers\Auth;

use ...

class RegisterController extends Controller
{
    /*
     * Register Controller
     *
     * This controller handles the registration of new users &
     * validation and creation. By default this controller uses
     * the "RegistersUsers" trait to provide this functionality without
     * requiring any additional code.
     */

    public function index()
    {
        return view('auth.register');
    }

    public function store(RegisterRequest $request)
    {
        $user = User::create([
            'name' => $request->name,
            'email' => $request->email,
            'password' => Hash::make($request->password),
        ]);

        Auth::login($user);

        return redirect()->intended('/');
    }
}
```

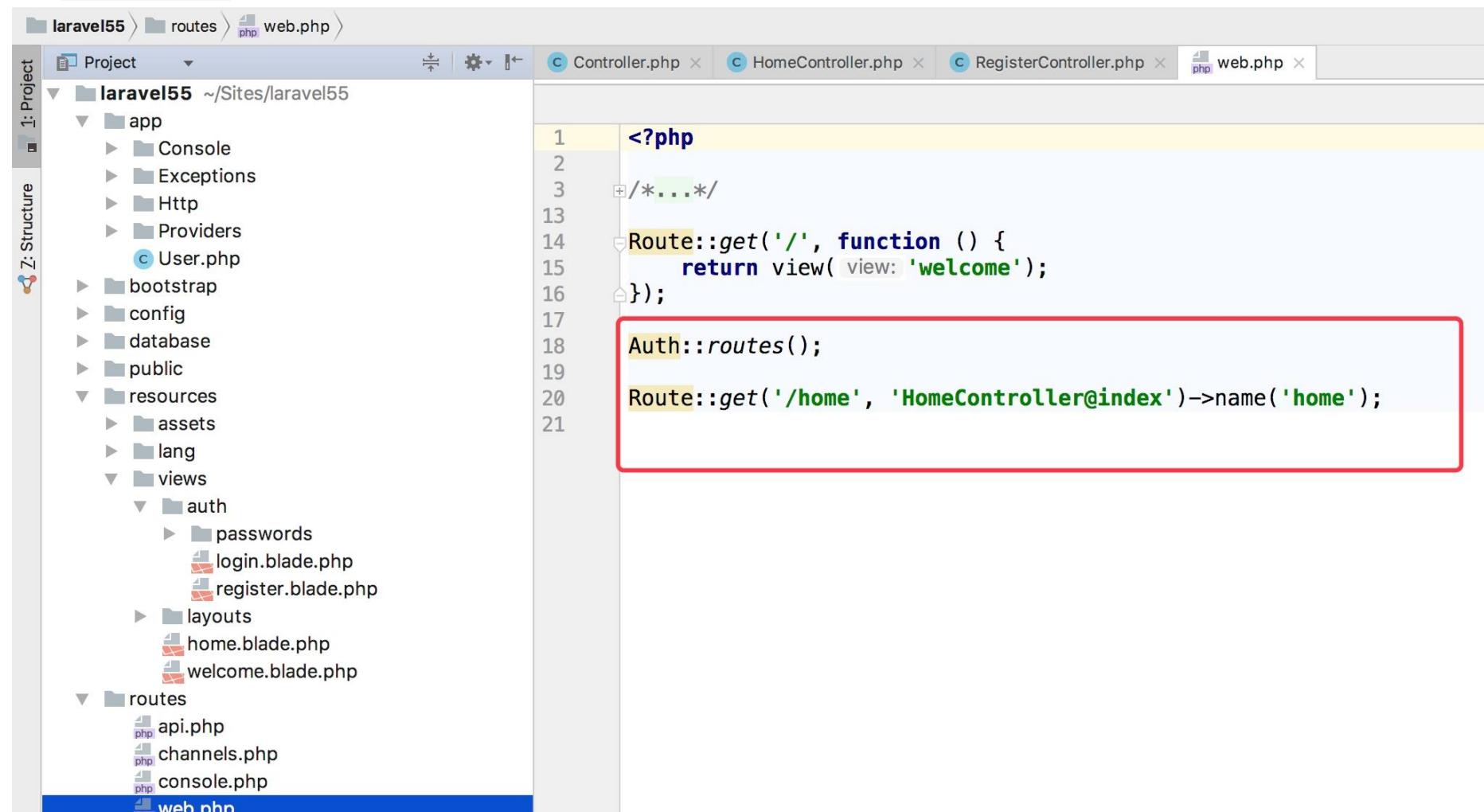
路由

Laravel 通过运行如下命令可快速生成认证所需要的路由和视图：

```
php artisan make:auth
```

新安装的应用运行该命令会生成布局、注册和登录视图，以及所有的认证路由，同时生成 `HomeController` 用于处理应用的登录请求。

打开 `routes/web.php` 路由文件会发现新增了两行：



```
<?php
/* ... */

Route::get('/', function () {
    return view('welcome');
});

Auth::routes();

Route::get('/home', 'HomeController@index')->name('home');
```

登录注册相关路由都定义在上面 `Auth::routes()` 方法内。

视图

正如上面所提到的，`php artisan make:auth` 命令会在 `resources/views/auth` 目录下创建所有认证需要的视图。

`make:auth` 命令还创建了 `resources/views/layouts` 目录，该目录下包含了应用的基础布局文件。所有这些视图都使用了 Bootstrap CSS 框架，你也可以根据需要对其进行自定义。

认证

现在你已经为自带的认证控制器设置好了路由和视图，接下来我们来实现新用户注册和登录认证。你可以在浏览器中访问定义好的路由，认证控制器默认已经包含了注册及登录逻辑（通过 trait）。

我们先来注册一个新用户，在浏览器中访问 <http://blog.test/register>，即可进入注册页面：

填写表单点击「Register」按钮即可完成注册。注册成功后页面跳转到认证后的页面 <http://blog.test/home>：

要测试登录功能，可以先退出当前用户，然后访问登录页面 <http://blog.test/login>：

使用我们之前注册的信息登录成功后，同样会跳转到 <http://blog.test/home>。

自定义路径

我们已经知道，当一个用户成功进行登录认证后，默认将会跳转到 `/home`，你可以通过在 `LoginController`、`RegisterController` 和 `ResetPasswordController` 中定义 `redirectTo` 属性来自定义登录认证成功之后的跳转路径：

```
protected $redirectTo = '/';
```

接下来，你需要编辑 `RedirectIfAuthenticated` 中间件的 `handle` 方法来使用新的重定向 URI。

如果重定向路径需要自定义生成逻辑可以定义一个 `redirectTo` 方法来取代 `redirectTo` 属性：

```
protected function redirectTo()
{
    return '/path';
}
```

注： `redirectTo` 方法优先级大于 `redirectTo` 属性。

自定义用户名

默认情况下，Laravel 使用 `email` 字段进行认证，如果你想要自定义认证字段，可以在 `LoginController` 中定义 `username` 方法：

```
public function username()
{
    return 'username';
}
```

自定义 Guard

你还可以自定义用于实现用户注册登录的“guard”，要实现这一功能，需要在 `LoginController`、`RegisterController` 和 `ResetPasswordController` 中定义 `guard` 方法，该方法将会返回一个 `guard` 实例：

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

```
protected function guard()
{
    return Auth::guard('guard-name');
}
```

需要注意的是，「guard」名称需要在配置文件 `config/auth.php` 中配置过：

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

自定义验证/存储

要修改新用户注册所必需的表单字段，或者自定义新用户字段如何存储到数据库，你可以修改 `RegisterController` 类。该类负责为应用验证输入参数和创建新用户。

`RegisterController` 的 `validator` 方法包含了新用户注册的验证规则，你可以按需要自定义该方法。

`RegisterController` 的 `create` 方法负责使用 `Eloquent ORM` 在数据库中创建新的 `App\User` 记录。当然，你也可以基于自己的需要自定义该方法。

获取登录用户

你可以通过 `Auth` 门面访问认证用户：

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

// 获取当前认证用户...
$user = Auth::user();

// 获取当前认证用户的 ID...
$id = Auth::id();
```

此外，用户通过认证后，你还可以通过 `Illuminate\Http\Request` 实例访问认证用户（类型提示类会通过依赖注入自动注入到控制器方法中）：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class ProfileController extends Controller{
    /**
     * 更新用户属性.
     *
     * @param Request $request
     * @return Response
     */
    public function update(Request $request)
    {
        // $request->user() 返回认证用户实例...
    }
}
```

上面两种方式返回的结果完全一致：

User {#210 ▼

```
#fillable: array:3 [▶]
#hidden: array:2 [▶]
#connection: "mysql"
#table: null
#primaryKey: "id"
#keyType: "int"
+incrementing: true
#with: []
#withCount: []
#perPage: 15
+exists: true
+wasRecentlyCreated: false
#attributes: array:7 [▶]
#original: array:7 [▶]
#changes: []
#casts: []
#dates: []
#dateFormat: null
#appends: []
#dispatchesEvents: []
#observables: []
#relations: []
#touches: []
+timestamps: true
#visible: []
#guarded: array:1 [▶]
#rememberTokenName: "remember_token"
}
```

判断当前用户是否通过认证

要判断某个用户是否登录到应用，可以使用 `Auth` 面面的 `check` 方法，如果用户通过认证则返回 `true`:

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

if (Auth::check()) {
    // The user is logged in...
}
```

注：尽管我们可以使用 `check` 方法判断用户是否通过认证，但是我们通常的做法是在用户访问特定路由/控制器之前使用中间件来验证用户是否通过认证，想要了解更多，可以查看下面的路由保护。

路由保护

路由中间件可用于只允许通过认证的用户访问给定路由。Laravel 通过定义在 `Illuminate\Auth\Middleware\Authenticate` 中的 `auth` 中间件来实现这一功能。由于该中间件已经在 `HTTP kernel` 中注册，你所要做的仅仅是将该中间件加到相应的路由定义中：

```
Route::get('profile', function() {
    // 只有认证用户可以进入...
})->middleware('auth');
```

当然，如果你也可以在**控制器**的构造方法中调用 `middleware` 方法而不是在路由器中直接定义实现同样的功能：

```
public function __construct() {
    $this->middleware('auth');
}
```

比如我们的 `HomeController` 就是这么做的：

```

<?php

namespace App\Http\Controllers;

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

class HomeController extends Controller
{
    /**
     * Create a new controller instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->middleware('auth');
    }
}

```

如果我们在没有登录的情况下访问 <http://blog.test/home> 页面就会重定向到登录页面。

重定向未认证用户

当 `auth` 中间件判定某个用户未认证，会返回一个 JSON `401` 响应，或者，如果不是 Ajax 请求的话，将用户重定向到 `login` 命名路由（也就是登录页面）。

你可以通过在 `app/Exceptions/Handler.php` 文件中定义一个 `unauthenticated` 方法来改变这一行为：

```

use Illuminate\Auth\AuthenticationException;

protected function unauthenticated($request, AuthenticationException $exception)
{
    return $request->expectsJson()
        ? response()->json(['message' => $exception->getMessage()], 401)
        : redirect()->guest(route('login'));
}

```

指定一个 Guard

添加 `auth` 中间件到路由后，还可以指定使用哪个 `guard` 来实现认证，指定的 `guard` 对应配置文件 `config/auth.php` 中 `guards` 数组的某个键：

```

public function __construct()
{
    $this->middleware('auth:api');
}

```

如果没有指定的话，默认 `guard` 是 `web`，这也是配置文件中配置的：

```

'defaults' => [
    'guard' => 'web',
    'passwords' => 'users',
],

```

登录失败次数限制

如果你使用了 Laravel 自带的 `LoginController` 类，就已经启用了内置的 `Illuminate\Foundation\Auth\ThrottlesLogins` trait 来限制用户登录失败次数。默认情况下，用户在几次登录失败后将在一分钟内不能登录，这种限制基于用户的用户名/邮箱地址+IP 地址作为唯一键。

手动认证用户

当然，你也可以不使用 Laravel 自带的认证控制器。如果你选择移除这些控制器，需要直接使用 Laravel 认证类来管理用户认证。别担心，这很简单！

我们可以通过 `Auth` 门面来访问认证服务，因此我们需要确保在类的顶部导入了 `Auth` 门面，接下来，让我们看看如何通过 `attempt` 方法实现登录认证：

```

<?php

namespace App\Http\Controllers;

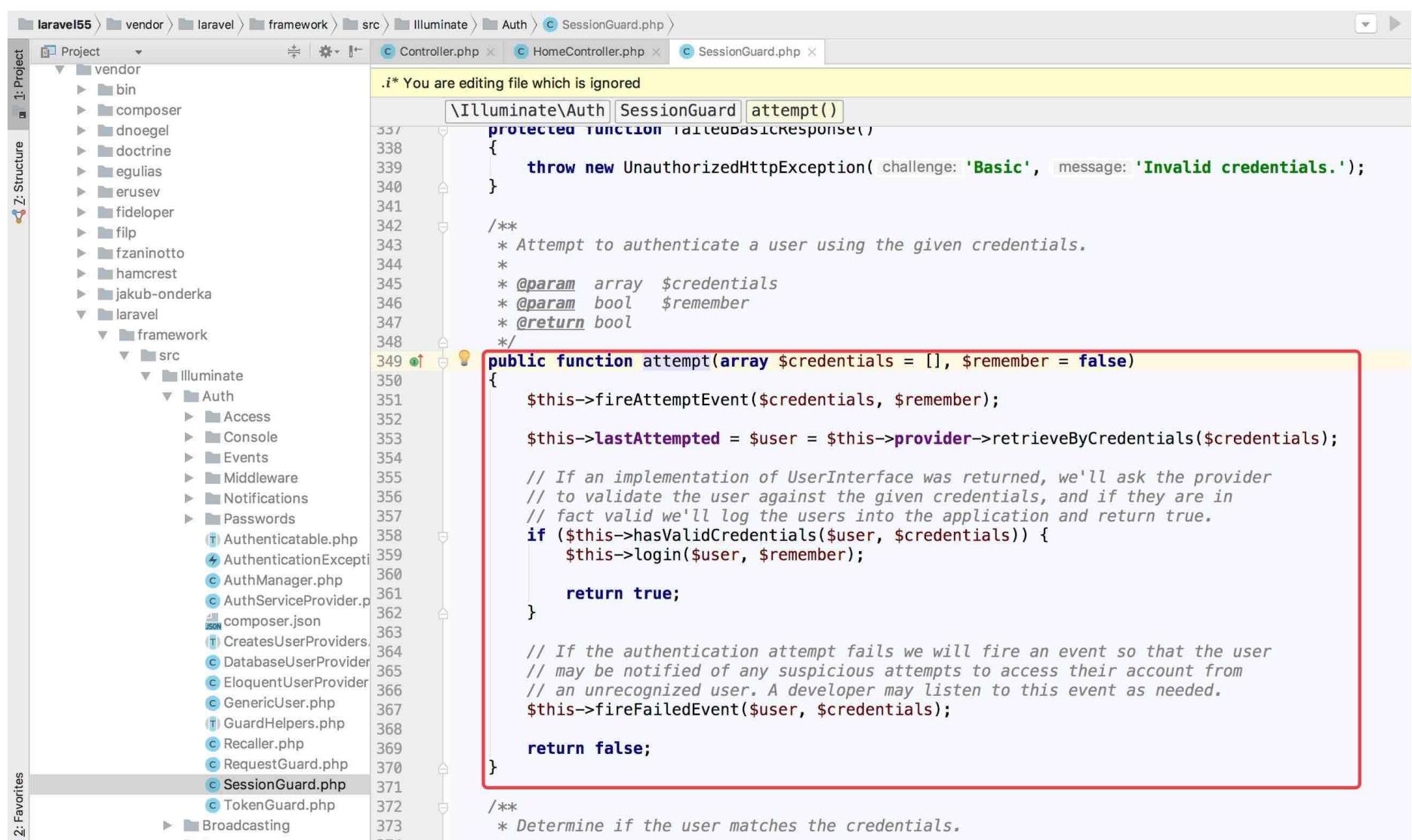
use Illuminate\Support\Facades\Auth;

class LoginController extends Controller

```

```
{
    /**
     * 处理登录认证
     *
     * @return Response
     * @translator laravelacademy.org
     */
    public function authenticate()
    {
        if (Auth::attempt(['email' => $email, 'password' => $password])) {
            // 认证通过...
            return redirect()->intended('dashboard');
        }
    }
}
```

`attempt` 方法接收键/值对作为第一个参数，数组中的值被用于从数据表中查找对应用户。在上面的例子中，将会通过 `email` 的值作为查询条件去数据库获取对应用户，如果用户被找到，经哈希运算后存储在数据库中的密码将会和传递过来的经哈希运算处理的密码值进行比较。如果两个经哈希运算的密码相匹配，那么将会为这个用户设置一个认证 Session，标识该用户登录成功。感兴趣的同学可以去看下底层源码实现逻辑：



如果认证成功的话 `attempt` 方法将会返回 `true`。否则，返回 `false`。

重定向器上的 `intended` 方法将用户重定向到登录之前用户想要访问的 URL，在目标 URL 无效的情况下回退 URI 将会传递给该方法。

指定额外条件

如果需要的话，除了用户邮件和密码之外还可以在认证查询时添加额外的条件，例如，我们可以验证被标记为有效的用户：

```
if (Auth::attempt(['email' => $email, 'password' => $password, 'active' => 1])) {
    // The user is active, not suspended, and exists.
}
```

这里的实现原理是在查询用户记录时，只是排除了数组中的密码字段，其他字段都会作为查询条件之一进行筛选：

```
public function retrieveByCredentials(array $credentials)
{
    if (empty($credentials)) {
        return;
    }

    // First we will add each credential element to the query as a where clause.
    // Then we can execute the query and, if we found a user, return it in a
    // Eloquent User "model" that will be utilized by the Guard instances.
    $query = $this->createModel()->newQuery();

    foreach ($credentials as $key Rightarrow $value) {
        if (! Str::contains($key, needles: 'password')) {
            $query->where($key, $value);
        }
    }

    return $query->first();
}
```

注：在这些例子中，并不仅仅限于使用 `email` 进行登录认证，这里只是作为演示示例，你可以将其修改为数据库中任何其他可用作“username”的字段。

访问指定 Guard 实例

你可以使用 `Auth` 门面的 `guard` 方法指定想要使用的 `guard` 实例，这种机制允许你在同一个应用中对不同的认证模型或用户表实现完全独立的用户认证。

该功能可用于为不同表的不同类型用户（同一个表不同类型用户理论上也可以）实现隔离式登录提供了方便，我们只要为每张表配置一个独立的 `guard` 就可以了。比如我们除了 `users` 表之外还有一张 `admins` 表用于存放后台管理员，要实现管理员的单独登录，就可以这么配置 `auth.php` 配置文件：

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],
    'api' => [
        'driver' => 'token',
        'provider' => 'users',
    ],
    'admin' => [
        'driver' => 'session',
        'provider' => 'admins',
    ],
],
'providers' => [
    'users' => [
        'driver' => 'eloquent',
        'model' => App\User::class,
    ],
    'admins' => [
        'driver' => 'eloquent',
        'model' => App\Admin::class,
    ],
],
```

友情提示：新建的用于登录认证的模型类需要继承 `Illuminate\Foundation\Auth\User` 基类，不然后面就会出现不能认证的窘况。

传递给 `guard` 方法的 `guard` 名称对应配置文件 `auth.php` 中 `guards` 配置的 `admin` 键：

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

需要注意的是使用这种方式认证的用户在后续操作需要传递 `guard` 时也要传递相匹配的 `guard`，比如上面提到的 `auth` 中间件，对应的调用方式也要调整（在路由中使用也是一样）：

```
$this->middleware('auth:admin');
```

获取用户时也是一样：

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

退出

要退出应用，可以使用 `Auth` 门面的 `logout` 方法，这将会清除用户 `Session` 中的认证信息：

```
Auth::logout();
```

记住用户

如果你想要在应用中提供“记住我”的功能，可以传递一个值为 `true` 的布尔值作为第二个参数到 `attempt` 方法（不传的话默认是 `false`），这样用户登录认证状态就会一直保持直到他们手动退出。当然，你的 `users` 表必须包含 `remember_token` 字段，该字段用于存储“记住我”令牌。

```
if (Auth::attempt(['email' => $email, 'password' => $password], $remember)) {
    // The user is being remembered...
}
```

注：如果你在使用自带的 `LoginController` 控制器，相应的记住用户逻辑已经通过控制器使用的 `trait` 实现了。

如果你在使用“记住”用户功能，可以使用 `viaRemember` 方法来判断用户是否通过“记住我”Cookie 进行认证：

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

其它认证方法

认证一个用户实例

如果你需要将一个已存在的用户实例直接登录到应用，可以调用 `Auth` 门面的 `login` 方法并传入用户实例，传入实例必须是 `Illuminate\Contracts\Auth\Authenticatable` 契约的实现，当然，Laravel 自带的 `App\User` 模型已经实现了该接口：

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

```
// 登录并“记住”给定用户...
```

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

当然，你可以指定想要使用的 `guard` 实例：

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

通过 ID 认证用户

要通过用户 ID 登录到应用，可以使用 `loginUsingId` 方法，该方法接收你想要认证用户的主键作为参数：

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

```
// 登录并“记住”给定用户...
```

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

一次性认证用户

你可以使用 `once` 方法只在单个请求中将用户登录到应用，而不存储任何 Session 和 Cookie，这在构建无状态的 API 时很有用：

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

基于 HTTP 的基本认证

HTTP 基本认证能够帮助用户快速实现登录认证而不用设置专门的登录页面，首先要在路由中加上 `auth.basic` 中间件。该中间件是 Laravel 自带的，所以不需要自己定义：

```
Route::get('profile', function() {
    // 只有认证用户可以进入...
})->middleware('auth.basic');
```

中间件加到路由中后，当在浏览器中访问该路由时，会自动提示需要认证信息，默认情况下，`auth.basic` 中间件使用用户记录上的 `email` 字段作为「用户名」。

学院君注：这种基本认证除了没有独立的登录表单视图之外底层实现逻辑和正常的登录认证没有区别。

FastCGI 上的注意点

如果你使用 PHP FastCGI，HTTP 基本认证将不能正常工作，需要在 `.htaccess` 文件加入如下内容：

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

无状态的 HTTP 基本认证

你也可以在使用 HTTP 基本认证时不在 Session 中设置用户标识 Cookie，这在 API 认证中非常有用。要实现这个功能，需要定义一个调用 `onceBasic` 方法的中间件。如果该方法没有返回任何响应，那么请求会继续走下去：

```
<?php
namespace Illuminate\Auth\Middleware;
```

```

use Illuminate\Support\Facades\Auth;

class AuthenticateOnceWithBasicAuth
{
    /**
     * 处理输入请求.
     *
     * @param \Illuminate\Http\Request $request
     * @param Closure $next
     * @return mixed
     * @translator laravelacademy.org
     */
    public function handle($request, $next)
    {
        return Auth::onceBasic() ?: $next($request);
    }
}

```

接下来，将 `AuthenticateOnceWithBasicAuth` 注册到路由中间件并在路由中使用它：

```

Route::get('api/user', function() {
    // 只有认证用户可以进入...
})->middleware('auth.basic.once');

```

添加自定义 Guard 驱动

你可以通过 `Auth` 门面的 `extend` 方法定义自己的认证 `guard` 驱动，该功能需要在某个服务提供者的 `boot` 方法中实现，由于 Laravel 已经自带了一个 `AuthServiceProvider`，所以我们将代码放到这个服务提供者中：

```

<?php

namespace App\Providers;

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

class AuthServiceProvider extends ServiceProvider
{
    /**
     * 注册任意应用认证/授权服务
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        Auth::extend('jwt', function($app, $name, array $config) {
            // 返回一个 Illuminate\Contracts\Auth\Guard 实例...
            return new JwtGuard(Auth::createUserProvider($config['provider']));
        });
    }
}

```

正如你在上面例子中所看到的，传递给 `extend` 方法的闭包回调需要返回 `Illuminate\Contracts\Auth\Guard` 的实现实例，该接口包含了自定义认证 `guard` 驱动需要的一些方法。定义好自己的认证 `guard` 驱动之后，就可以在配置文件 `auth.php` 的 `guards` 配置中使用这个新的 `guard` 驱动：

```

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

```

添加自定义用户提供者

如果你没有使用传统的关系型数据库存储用户信息，则需要使用自己的认证用户提供者来扩展 Laravel。我们使用 `Auth` 门面上的 `provider` 方法定义自定义该提供者：

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Auth;
use App\Extensions\RiakUserProvider;
use Illuminate\Support\ServiceProvider;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * 注册任意应用认证/授权服务.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        Auth::provider('riak', function($app, array $config) {
            // 返回一个 Illuminate\Contracts\Auth\UserProvider 实例...
            return new RiakUserProvider($app->make('riak.connection'));
        });
    }
}
```

通过 `provider` 方法注册用户提供者后，你可以在配置文件 `config/auth.php` 中切换到新的用户提供者。首先，定义一个使用新驱动的 `provider`：

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

然后，可以在你的 `guards` 配置中使用这个提供者：

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

UserProvider 契约

`Illuminate\Contracts\Auth\UserProvider` 实现只负责从持久化存储系统中获取 `Illuminate\Contracts\Auth\Authenticatedable` 实现，例如 MySQL、Riak 等等。这两个接口允许 Laravel 认证机制继续起作用而不管用户数据如何存储或者何种类来展现。让我们先看看 `Illuminate\Contracts\Auth\UserProvider` 契约：

```
<?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` 方法通常获取一个代表用户的键，例如 MySQL 数据中的自增 ID。该方法获取并返回匹配该 ID 的 `Authenticatedable` 实现。

`retrieveByToken` 函数通过唯一标识和存储在 `remember_token` 字段中的“记住我”令牌获取用户。和上一个方法一样，该方法也返回 `Authenticatedable` 实现。

`updateRememberToken` 方法使用新的 `$token` 更新 `$user` 的 `remember_token` 字段，新令牌可以是新生成的令牌（在登录时选择“记住我”被成功赋值）或者 `null`（用户退出）。

`retrieveByCredentials` 方法在尝试登录系统时获取传递给 `Auth::attempt` 方法的认证信息数组。该方法接下来去底层持久化存储系统查询与认证信息匹配的用户，通常，该方法运行一个带“where”条件 (`$credentials['username']`) 的查询。然后该方法返回 `Authenticatable` 的实现。这个方法不应该做任何密码校验和认证。

`validateCredentials` 方法比较给定 `$user` 和 `$credentials` 来认证用户。例如，这个方法比较 `$user->getAuthPassword()` 字符串和经 `Hash::check` 处理的 `$credentials['password']`。这个方法根据密码是否有效返回布尔值 `true` 或 `false`。

Authenticatable 契约

既然我们已经探索了 `UserProvider` 上的每一个方法，接下来让我们看看 `Authenticatable`。记住，提供者需要从 `retrieveById` 和 `retrieveByCredentials` 方法中返回接口实现：

```
<?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();
}
```

这个接口很简单，`getAuthIdentifierName` 方法会返回用户的主键字段名称，`getAuthIdentifier` 方法返回用户“主键”，在后端 MySQL 中这将是自增 ID，`getAuthPassword` 返回经哈希处理的用户密码，这个接口允许认证系统处理任何用户类，不管是你使用的是 ORM 还是存储抽象层。默认情况下，Laravel `app` 目录下的 `User` 类实现了这个接口，所以你可以将这个类作为实现例子。

事件

Laravel 支持在认证过程中触发多种事件，你可以在自己的 `EventServiceProvider` 中监听这些事件：

```
/**
 * 应用的事件监听器映射.
 *
 * @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\Logout' => [
        'App\Listeners\LogSuccessfulLogout',
    ],

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

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

API 认证

简介

Laravel 通过传统的登录表单已经让用户认证变得很简单，但是 API 认证怎么实现？API 通常使用令牌（token）进行认证并且在请求之间不维护会话（Session）状态。Laravel 官方扩展包 Laravel Passport 让 API 认证变得轻而易举，Passport 基于 Alex Bilbie 维护的 League OAuth2 server，可以在数分钟内为 Laravel 应用提供完整的 OAuth2 服务器实现。

OAuth2 概述

正式开始之前我们先简单了解下 OAuth2。

什么是 OAuth 协议

OAuth 是 Open Authorization 的简写，OAuth 协议为用户资源的授权提供了一个安全的、开放而又简易的标准。与以往的授权方式不同之处是 OAuth 的授权不会使第三方触涉及到用户的帐号信息（如用户名与密码），即第三方无需使用用户的用户名与密码就可以申请获得该用户资源的授权，因此 OAuth 是安全的。

OAuth 本身不存在一个标准的实现，后端开发者自己根据实际的需求和标准的规定实现。其步骤一般如下：

- 客户端要求用户给予授权
- 用户同意给予授权
- 根据上一步获得的授权，向认证服务器请求令牌（token）
- 认证服务器对授权进行认证，确认无误后发放令牌
- 客户端使用令牌向资源服务器请求资源
- 资源服务器使用令牌向认证服务器确认令牌的正确性，确认无误后提供资源

OAuth2 解决什么问题

任何身份认证，本质上都是基于对请求方的不信任所产生的。同时，请求方是信任被请求方的，例如用户请求服务时，会信任服务方。所以，身份认证就是为了解决身份的可信任问题。

在 OAuth 中，简单来说有三方：用户（这里是指属于服务方的用户）、服务方、第三方应用（客户端）。

服务方不信任用户，所以需要用户提供密码或其他可信凭据；

服务方不信任第三方，所以需要第三方提供自己交给它的凭据（通常的一些安全签名之类的就是）；

用户部分信任第三方，所以用户愿意把自己在服务方里的某些服务交给第三方使用，但不愿意把自己在服务方的密码交给第三方；

在 OAuth 的流程中，用户登录了第三方的系统后，会先跳去服务方获取一次性用户授权凭据，再跳回来把它交给第三方，第三方的服务器会把授权凭据以及服务方给它的的身份凭据一起交给服务方，这样，服务方一可以确定第三方得到了用户对此次服务的授权（根据用户授权凭据），二可以确定第三方的身份是可以信任的（根据身份凭据），所以，最终的结果就是，第三方顺利地从服务方获取到了此次所请求的服务。

从上面的流程中可以看出，OAuth 完整地解决掉了用户、服务方、第三方 在某次服务时这三者之间的信任问题。

OAuth 基本流程

Abstract Protocol Flow



涉及成员：

- Resource Owner (资源拥有者：用户)
- Client (第三方接入平台：请求者)
- Resource Server (服务器资源：数据中心)
- Authorization Server (认证服务器)

注：Passport 需要你对 OAuth2 非常熟悉才能自如使用，上面关于 OAuth2 的概述转自[理解 OAuth2.0 认证](#)一文，更多关于 OAuth2 模式的探讨，还可以参考阮一峰博客：[理解 OAuth 2.0](#)。

2、安装

首先通过 Composer 包管理器安装 Passport:

```
composer require laravel/passport
```

注: 如果安装过程中提示需要更高版本的 Laravel: `laravel/passport v5.0.0 requires illuminate/http ~5.6`, 可以通过指定版本来安装 `composer require laravel/passport ~4.0`。

Passport 服务提供者为框架注册了自己的数据库迁移目录, 所以在注册服务提供者之后 (Laravel 5.5 之后会自动注册服务提供者) 需要迁移数据库, Passport 迁移将会为应用生成用于存放客户端和访问令牌的数据表:

```
php artisan migrate
```

注: 如果你不想使用 Passport 的默认迁移, 需要在 `AppServiceProvider` 的 `register` 方法中调用 `Passport::ignoreMigrations` 方法。你可以使用 `php artisan vendor:publish --tag=passport-migrations` 导出默认迁移。

接下来, 需要运行 `passport:install` 命令, 该命令将会创建生成安全访问令牌 (token) 所需的加密键, 此外, 该命令还会创建「personal access」和「password grant」客户端用于生成访问令牌:

```
php artisan passport:install
```

生成记录存放在数据表 `oauth_clients`:

	id	user_id	name	secret	redirect	personal_access_client	password_client	revoked	created_at	updated_at
1	NULL	Laravel Personal Access Client	A4G474s67EAjtw5Ehj5n2OidhcKIV...	http://localhost		1	0	0	2018-02-11 05:11:01	2018-02-11 05:11:01
2	NULL	Laravel Password Grant Client	XtkyWdevgTnqbVtTd8l7ASx76VtB...	http://localhost		0	1	0	2018-02-11 05:11:01	2018-02-11 05:11:01

运行完这个命令后, 添加 `Laravel\Passport\HasApiTokens` trait 到 `App\User` 模型, 该 trait 将会为模型类提供一些辅助函数用于检查认证用户的 token 和 scope:

```
<?php

namespace App;

use Laravel\Passport\HasApiTokens;
use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;
}
```

接下来, 你需要在 `AuthServiceProvider` 的 `boot` 方法中调用 `Passport::routes` 方法, 该方法将会为颁发访问令牌、撤销访问令牌、客户端以及私人访问令牌注册必要的路由:

```
<?php

namespace App\Providers;

use Laravel\Passport\Passport;
use Illuminate\Support\Facades\Gate;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        'App\Model' => 'App\Policies\ModelPolicy',
    ];

    /**
     * Register any authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();
    }
}
```

```

        Passport::routes();
    }
}

```

最后，在配置文件 `config/auth.php` 中，需要设置 `api` 认证 `guard` 的 `driver` 选项为 `passport`。这将告知应用在认证输入的 API 请求时使用 `Passport` 的 `TokenGuard`：

```

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

```

前端快速入门

注：如果要使用 `Passport Vue` 组件，前端 `JavaScript` 必须使用 `Vue` 框架，这些组件同时也使用了 `Bootstrap CSS` 框架。不过，即使你不使用这些工具，这些组件同样可以为你实现自己的前端组件提供有价值的参考。

`Passport` 附带了 `JSON API` 以便用户创建客户端和私人访问令牌（`access token`）。不过，考虑到编写前端代码与这些 `API` 交互是一件很花费时间的事，`Passport` 还预置了 `Vue` 组件作为示例以供使用（或者作为自己实现的参考）。

要发布 `Passport Vue` 组件，可以使用 `vendor:publish` 命令：

```
php artisan vendor:publish --tag=passport-components
```

发布后的组件位于 `resources/assets/js/components` 目录下，组件发布之后，还需要将它们注册到 `resources/assets/js/app.js` 文件：

```

Vue.component(
    'passport-clients',
    require('./components/passport/Clients.vue')
);

Vue.component(
    'passport-authorized-clients',
    require('./components/passport/AuthorizedClients.vue')
);

Vue.component(
    'passport-personal-access-tokens',
    require('./components/passport/PersonalAccessTokens.vue')
);

```

注册完组件后，确保运行 `npm run dev` 来重新编译前端资源。重新编译前端资源后，就可以将这些组件放到应用的某个模板中以便创建客户端和私人访问令牌：

```
<passport-clients></passport-clients>
<passport-authorized-clients></passport-authorized-clients>
<passport-personal-access-tokens></passport-personal-access-tokens>
```

部署 Passport

第一次部署 `Passport` 到生产服务器时，可能需要运行 `passport:keys` 命令。这个命令生成 `Passport` 需要的加密 `keys` 以便生成访问令牌，生成的 `keys` 将不会存放在源代码控制中：

```
php artisan passport:keys
```

配置

令牌生命周期

默认情况下，`Passport` 颁发的访问令牌（`access token`）是长期有效的，如果你想要配置生命周期短一点的令牌，可以使 `tokensExpireIn` 和 `refreshTokensExpireIn` 方法，这些方法需要在 `AuthServiceProvider` 的 `boot` 方法中调用：

```

/**
 * 注册任意认证/授权服务
 *
 * @return void
 */
public function boot()
{

```

```
$this->registerPolicies();

Passport::routes();

Passport::tokensExpireIn(now()->addDays(15));

Passport::refreshTokensExpireIn(now()->addDays(30));
}
```

颁发访问令牌

通过授权码使用 OAuth2 是大多数开发者熟悉的方式。使用授权码的时候，客户端应用会将用户重定向到你的服务器，服务器可以通过或拒绝颁发访问令牌到客户端的请求。

管理客户端

约定：我们参考 OAuth2 认证流程，这里将 Laravel 应用约定为服务方，开发者开发应用作为第三方客户端。

首先，开发者构建和 Laravel 应用 API 交互的应用时，需要通过创建一个“客户端”将他们的应用注册到 Laravel 应用。通常，这包括提供应用的名称以及用户授权请求通过后重定向到的 URL。（想想你是怎么使用微博、微信、QQ 第三方登录 API 的，就明白这里的流程了）

passport:client 命令

创建客户端最简单的方式就是使用 Artisan 命令 `passport:client`，该命令可用于创建你自己的客户端以方便测试 OAuth2 功能。当你运行 `client` 命令时，Passport 会提示你输入更多关于客户端的信息，并且为你生成 client ID 和 secret：

```
php artisan passport:client
```

```
localhost:laravel55 sunqiang$ php artisan passport:client

Which user ID should the client be assigned to?:
> 1

What should we name the client?:
> Laravel Test Access Client

Where should we redirect the request after authorization? [http://localhost/auth/callback]:
> http://laravel55.dev/auth/callback

New client created successfully.
Client ID: 3
Client secret: tBxbskNg9fJTIh0Ufk4eKdpneSkLx1H5HxGy2VTk
```

新生成记录存放在 `oauth_clients`：

	id	user_id	name	secret	redirect	personal_access_client	password_client	revoked	created_at	updated_at
migrations	1	NULL	Laravel Personal Access Client	A4G474s67EAjtwSEhj5n2OidhcK...	http://localhost	1	0	0	2018-02-11 05:11:01	2018-02-11 0
oauth_access_tokens	2	NULL	Laravel Password Grant Client	XtkyWdvevgTnqbVtTd8l7ASx76Vt...	http://localhost	0	1	0	2018-02-11 05:11:01	2018-02-11 0
oauth_auth_codes	3	1	Laravel Test Access Client	tBxbskNg9fJTIh0Ufk4eKdpneSkLx...	http://laravel55.dev/auth/callback	0	0	0	2018-02-11 05:57:30	2018-02-11 0
oauth_clients										
oauth_personal_access_clients										

JSON API

由于第三方应用开发者不能直接使用 Laravel 服务端提供的 `client` 命令，为此，Passport 提供了一个 JSON API 用于创建客户端，这省去了你手动编写控制器用于创建、更新以及删除客户端的麻烦。

不过，你需要配对 Passport 的 JSON API 和自己的前端以便为第三方开发者提供一个可以管理他们自己客户端的后台，下面，我们来概览下所有用于管理客户端的 API，为了方便起见，我们将会使用 Axios 来演示发送 HTTP 请求到 API：

注：如果你不想要自己实现整个客户端管理前端，可以使用 [前端快速上手教程](#) 在数分钟内搭建拥有完整功能的前端。

GET /oauth/clients

这个路由为认证用户返回所有客户端，这在展示用户客户端列表时很有用，可以让用户很容易编辑或删除客户端：

```
axios.get('/oauth/clients')
  .then(response => {
    console.log(response.data);
  });
```

POST /oauth/clients

这个路由用于创建新的客户端，要求传入两个数据：客户端的 `name` 和 `redirect` URL，`redirect` URL 是用户授权请求通过或拒绝后重定向到的位置。

当客户端被创建后，会附带一个 `client ID` 和 `secret`，这两个值会在请求访问令牌时用到。客户端创建路由会返回新的客户端实例：

```
const data = {
  name: 'Client Name',
  redirect: 'http://example.com/callback'
};

axios.post('/oauth/clients', data)
  .then(response => {
    console.log(response.data);
  })
  .catch (response => {
```

```
// List errors on response...
});
```

PUT /oauth/clients/{client-id}

这个路由用于更新客户端，要求传入两个参数：客户端的 `name` 和 `redirect` URL。`redirect` URL 是用户授权请求通过或拒绝后重定向到的位置。该路由将会返回更新后的客户端实例：

```
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 => {
        // List errors on response...
    });
});
```

DELETE /oauth/clients/{client-id}

这个路由用于删除客户端：

```
axios.delete('/oauth/clients/' + clientId)
    .then(response => {
        //
    });
});
```

请求令牌

授权重定向

客户端被创建后，开发者就可以使用相应的 client ID 和 secret 从应用请求授权码和访问令牌。首先，客户端应用要生成一个重定向请求到服务端应用的 `/oauth/authorize` 路由：

```
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'code',
        'scope' => '',
    ]);

    return redirect('http://your-app.com/oauth/authorize?' . $query);
});
```

注：`/oauth/authorize` 路由已经通过 `Passport::routes` 方法定义了，不需要手动定义这个路由。

我们使用上面创建的 client ID 等于 3 的测试客户端做测试，改写上面的测试代码如下（路由定义在 `routes/api.php` 中）：

```
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => '3',
        'redirect_uri' => 'http://laravel55.dev/auth/callback',
        'response_type' => 'code',
        'scope' => '',
    ]);

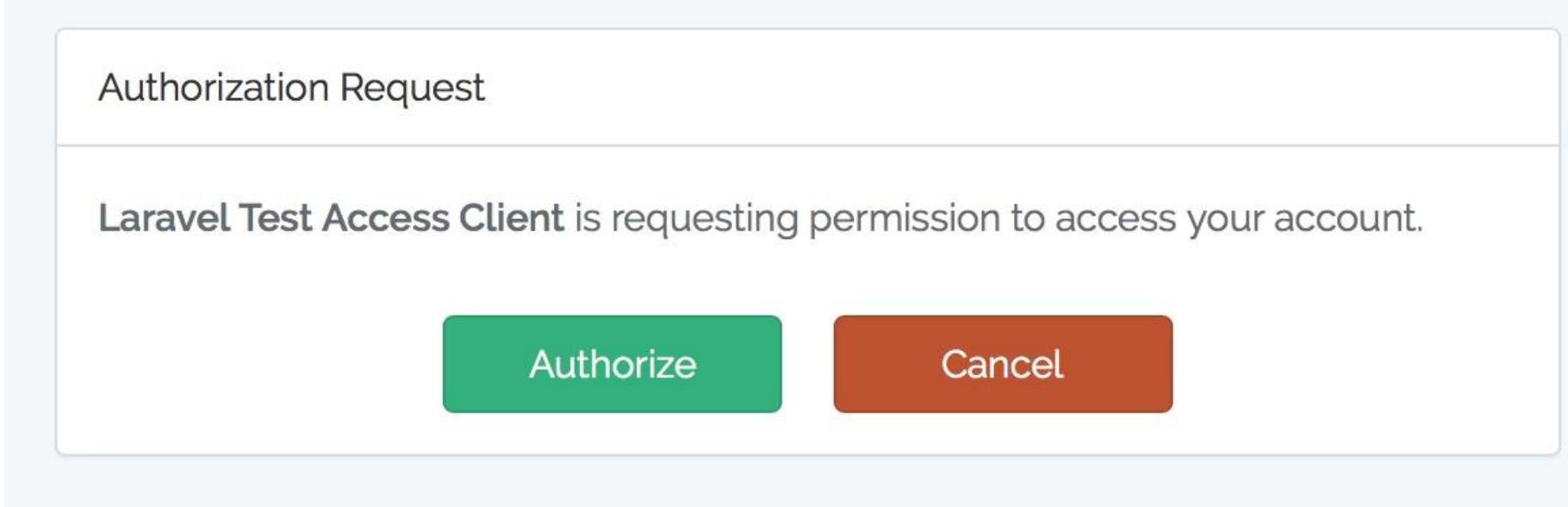
    return redirect('http://laravel55.dev/oauth/authorize?' . $query);
});
```

同时在 `routes/web.php` 注册 `auth/callback` 路由：

```
Route::get('/auth/callback', function (\Illuminate\Http\Request $request) {
    if ($request->get('code')) {
        return 'Login Success';
    } else {
        return 'Access Denied';
    }
});
```

然后在浏览器中访问 `http://blog.test/api/redirect`，如果用户尚未在 `blog` 应用中登录，首选会重定向到表单登录页面，登录成功之后就会跳转到第三方授权登录页

面 `http://blog.test/oauth/authorize?client_id=3&redirect_uri=http%3A%2F%2Fblog.test%2Fauth%2Fcallback&response_type=code&scope=:`



通过请求

接收授权请求的时候，Passport 会自动显示一个视图模板给用户从而允许他们通过或拒绝授权请求（如上图所示），如果用户通过请求，就会被重定向回第三方应用指定的 `redirect_uri`（本例中是 `http://blog.test/auth/callback`），这个 `redirect_uri` 必须和客户端创建时指定的 `redirect URL` 一致。

如果你想要自定义授权通过界面，可以使用 Artisan 命令 `vendor:publish` 发布 Passport 的视图模板，发布的视图位于 `resources/views/vendor/passport`：

```
php artisan vendor:publish --tag=passport-views
```

将授权码转化为访问令牌

如果用户通过了授权请求，会被重定向回第三方应用。第三方应用接下来会发送一个 `POST` 请求到服务端应用来请求访问令牌。这个请求应该包含用户通过授权请求时指定的授权码。在这个例子中，我们会使用 Guzzle HTTP 库来生成 `POST` 请求：

```
Route::get('/auth/callback', function (Request $request) {
    $http = new GuzzleHttp\Client;

    $response = $http->post('http://blog.test/oauth/token', [
        'form_params' => [
            'grant_type' => 'authorization_code',
            'client_id' => '3', // your client id
            'client_secret' => 'tBxbeskNg9fJTIh0Ufk4eKdpneSkLx1H5HxGy2VTk', // your client secret
            'redirect_uri' => 'http://blog.test/auth/callback',
            'code' => $request->code,
        ],
    ]);

    return json_decode((string) $response->getBody(), true);
});
```

`/oauth/token` 路由会返回一个包含 `access_token`、`refresh_token` 和 `expires_in` 属性的 JSON 响应。`expires_in` 属性包含访问令牌的过期时间（s）：

```
{
    "token_type": "Bearer",
    "expires_in": 31536000,
    "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImp0aSI6IjhjNDU0NzRlNDMxOGE0ZjAzOTM3MWUwZjViYzEzYjhjYTZmYmViODBkNDZhYWExNTY1NzQzzjI4YzU1ZTA1MmYyOGI1Nzg1Nzk0Zje0ODNkIn0.eJhdWQioIzIiwianRpIjoiOGM0NTQ3NGU0MzE4YTRmMDM5MzcxZTBmNWJjMTNioGNhNmZiZWI4MGQ0NmFhYTE1Nju3NDNmMjhjNTV1MDUyZjI4Yju3ODU30TRmMTQ4M2QiLCJpYXQiOjE1MTgzMzE4MTAsIm5iZiI6MTUxODMzMtGzMcWiZXhwIjoxNTQ5ODY3ODEwLCJzdWIiOiIxIiwic2NvcGVzIjpBX0.MEosfnNq-vv3Nt2kuEbHbr6apj_ICfvHZaXY7eakPzyUb8hDSAFpzA4wildnmejAapCcEKLUAZJiyE5fcmbcp8pTSDWAmR_9Xsz-HQmwUiUsIpL0Mi2XDasIT1AhrsI6TSFa7zHBtm-aDBWLFC0zMbqTYn0gjDrsEo1NmEPXKY700cj4Ej41eekaTpKSDmpgLisWh_DIjt-2tPebeSqm7gm8Hzq9QhP6RSfl12K17VapGkx1bbXnb7CD-tq2i52I0P4xQ8oSg5vfQ9_v08D6FA4JtHHqgyNYsMCn4DvJ-6U4Snt7NfEiQ21xsEMFka8JvbwYunheCpobBmdqtrrqZyuu-PMScKzwfIQC99jaE1k9iVRQt2EM94ZLhvBrEruZYXkJdaUIuKqlB7MIghkJ18WLbBdT6JD9eX6LhmGQsejWOKk7ivz6qgnZjbnxqmGv0ePYBgmZz25EbXZqSmd2K8JQedG1nOf8DT21Z9g09gDB-1ZYfGZQyQdIZNBP2hryEy60-kYAPX2tdjqyDn2QppcEhQyzW52n1MnZxrQIeavOHkjN33p4vT7jd4DFh1ahPYL7wOn1CF7OHCZxdWWwY0xSGeIGb0YkvGodj-MJeIn4sD4iIrkc60UsrEM7oMzPQm47p1ZZP0EAawLnwuyVh7pgj4eDdEre8TVuPGM",
    "refresh_token": "def5020092bab24860c9aaa5560508169285835c6889252c794eaa52da53ae6c7b4ac658c3a00224ce2b62e99f5eedc2f9d8d7645bdd6a3a08b65e1aecb63fcdaad5658f47da866a397105a3807c119814cdcb1a390c7d22169367af8a7b5231519467fce5c56eb01a8669ac9762c4877b539477119ebbe168728d601288ad3e706726262360afb1135f7e1a144bd993f75b5f13883f06fbb6499384d448791df752e0fa66b9cf839eaeb0ab64461d3fe12170f66513fa3cc487cb36df0631db4450c1a6d0517b881aa6945420fb5e295ea862b7b904238d9feb131e2dc1476cc93314a9b810a4981aa8d563684bb342328acf9d1dc5f896dc2890a3f6c8533aa745c6472ec04a1a42c865ade56d78aebe9d4b09dfe16f55e73a6ea97621df06c150a7a9f0466fab0afc190052e7a1e121b148b0431fca48187b0023cf4425c7d08f0dbae5fabeaed2731894b2ec8d06b79e991d01d6a0973a345148ecb78f8"
}
```

注：和 `/oauth/authorize` 路由一样，`/oauth/token` 路由已经通过 `Passport::routes` 方法定义过了，不需要手动定义这个路由。拿到有效的 `access_token` 就可以通过它去服务端获取其他需要的资源信息了。至此就完成了通过授权码方式实现 API 认证的流程。实际开发中，99% 的 API 认证都是通过这种方式实现的。

刷新令牌

如果应用颁发的是短期有效的访问令牌，那么用户需要通过访问令牌颁发时提供的 `refresh_token` 刷新访问令牌，在本例中，我们使用 Guzzle HTTP 库来刷新令牌：

```
$http = new GuzzleHttp\Client;

$response = $http->post('http://blog.test/oauth/token', [
    'form_params' => [
        'grant_type' => 'refresh_token',
        'refresh_token' => 'the-refresh-token',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'scope' => '',
    ],
]);

return json_decode((string) $response->getBody(), true);
```

`/oauth/token` 路由会返回一个包含 `access_token`、`refresh_token` 和 `expires_in` 属性的 JSON 响应，同样，`expires_in` 属性包含访问令牌过期时间 (s)。

密码授权令牌

OAuth2 密码授权允许你的其他第一方客户端，例如移动应用，使用邮箱地址/用户名+密码获取访问令牌。这使得你可以安全地颁发访问令牌给第一方客户端而不必要求你的用户走整个 OAuth2 授权码重定向流程。

创建密码发放客户端

在应用可以通过密码授权颁发令牌之前，需要创建一个密码授权客户端，你可以通过使用带 `--password` 选项的 `passport:client` 命令来实现。如果你已经运行了 `passport:install` 命令，则不必再运行这个命令：

```
php artisan passport:client --password
```

这里我们使用一开始通过 `passport:install` 命令创建的记录作为测试记录。

请求令牌

创建完密码授权客户端后，可以通过发送 `POST` 请求到 `/oauth/token` 路由（带上用户邮箱地址和密码）获取访问令牌。这个路由已经通过 `Passport::routes` 方法注册过了，不需要手动定义。如果请求成功，就可以从服务器返回的 JSON 响应中获取 `access_token` 和 `refresh_token`：

```
Route::get('/auth/password', function (\Illuminate\Http\Request $request) {
    $http = new \GuzzleHttp\Client();

    $response = $http->post('http://blog.test/oauth/token', [
        'form_params' => [
            'grant_type' => 'password',
            'client_id' => '2',
            'client_secret' => 'XtkyWdevgTnqbVtWTd817ASx76VtBBuZHzlAbCvm',
            'username' => 'yaojinbu@163.com',
            'password' => 'test123',
            'scope' => '',
        ],
    ]);

    return json_decode((string)$response->getBody(), true);
});
```

请求的时候直接访问 `http://blog.test/auth/password` 即可。返回数据如下：

```
{
    "token_type": "Bearer",
    "expires_in": 31535999,
    "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImp0aSI6IjQ1MjYzYjZjNDhmOTUzYzc3NzU3MTA5ZDgwZmQwYTQxNTU1MmZ1YThhNTAwYzZhMTYzOWZkY2U2NWFjYZa5MjBiOGI3YjRjYjUwZWU1In0.eyJhdWQiOiIyIiwanRpIjoiNDUyNjNiNmM0OGY5NTNjNzc3NTcxMDlkODBmZDBhNDE1NTUyZmVhOGE1MDBjNmExNjM5ZmIxZmRjZTY1YWNjMDkyMGI4YjdiNGNiNTB1ZTUilCJpYXQiOjE1MTgzMzM1ODksIm5iZiI6MTUxODMzMzU4OSwizXhwIjoxNTQ5ODY5NTg4LCJzdWIiOixIiwic2NvcGVzIjpbXX0.tuWpRqoKQBI2jgrKCK5lN0Htfvvb_DZz0WSw8TI6CTNrLo89j0Bk2YveixN864KHO4vHnSn01b7btCj_dSrr3K2AnfyuNny5iQ1gNe_rWZ-9qdU05ztFeDTxtivx_oT39Q0yEnQKJHq5NGytEec6FDEHfM6sE6TPyIIKriAglnxoR0gIYvTrAtItX24OEON6zNVmpEMcTqlxFxKcUEv7neuT_TMIBh3eF5JfCTBYcXuMoOEGDMm0pCgleAWRrsMRpIzxwyewCIHBXypvd1w2aF1BzfzpMVFT_g-96hyAyTWEykinazuLdfETQBj8M0f2n61seQU8KKY_b3Mn4IZu9qtChNh1rxetZTE08wsb41XymTvpcbXRnTssGgPp8Fan_MAhGpIZkVG3iErblkAAJMqBOXnaEXKCmX_ZwNXesn4wf1m22t3ttZ3j8JxpncFHTuHP6fRYB8YslaEsDvjry-R5DRSuzgyi8UAgjlpbwPtXrFdnvoOwkJpv8usmeu11U3HiCt4LoADctuTSBykg6z2nTALdk9AwkJcLUA33ITIO_N_wwRfNk6uNP1NVv_jlTCepBXdy44DHK7TTLsFr_sqglwoETf58j8nyMGten1YhQWPX050vQw5-1nbk2nPCAdfQgGNyDKTom9t_24A7ekMRGuFrcGcXJKUxDow",
    "refresh_token": "def50200f2dc4d76b4b56b59f75cc35cf4ade7d1916cf845d2ad47612a7871ce597aee676d99019d7dd372c3c5a2759bdb53d9b3c0004d7f69cc238bf403491a0f782e0cd49fa32e564ab4edb7a814630def00cf4c3ef7afdcce20e585f70dd16ccb8a7810158c4349e95219dcbb83e36bc30e21109a1f872ea0021f6b47c58d9bbd89d56b09df036b72d0076e896dde1280ec52f86b11f0036fe578e088a8bf5940c432f02e1c0f78c2af3ff4cf16bd6244eff343d38d2e588db5a9cc9bab92682e7915c3154fd8751a22b5196eb74e047d70e68bd63e7380860fdfdc937ec3185afe8140aa150546ce39d9f226f4862abfc3c38c67d48082449c7d8f1200d61c4126e77970e114f53b34117c012368cdf3b280494a69c79230ca0bcc336242ed86805304c3d017fd7f6194aalbe87c7b58d93836d28bac20fe1a2144d3d81128c26e789fb7342827067cd300f46b99be6ef1a4e016efa3361e0fea4bf4ab9e8"
}
```

和通过授权码返回数据格式一致。

注：记住，访问令牌默认长期有效，不过，如果需要的话你也可以[配置访问令牌的最长生命周期](#)。

请求所有域

使用密码授权的时候，你可能想要对应用所支持的所有域进行令牌授权，这可以通过请求 * 域来实现。如果你请求的是 * 域，则令牌实例上的 can 方法总是返回 true，这个域只会分配给使用 password 授权的令牌：

```
$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'password',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'username' => 'taylor@laravel.com',
        'password' => 'my-password',
        'scope' => '*',
    ],
]) ;
```

隐式授权令牌

隐式授权和授权码授权有点相似，不过，无需获取授权码，令牌就会返回给客户端。这种授权通常应用于 JavaScript 或移动应用这些客户端凭证不能被安全存储的地方。要启用该授权，在 `AuthServiceProvider` 中调用 `enableImplicitGrant` 方法即可：

```
/***
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::enableImplicitGrant();
}
```

授权启用后，开发者就可以使用他们的 client ID 从应用中请求访问令牌，第三方应用需要像这样发送重定向请求到应用的 `/oauth/authorize` 路由：

```
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'token',
        'scope' => '',
    ]);

    return redirect('http://your-app.com/oauth/authorize?' . $query);
});
```

注：`/oauth/authorize` 路由已经在 `Passport::routes` 方法中定义过了，无需再手动定义这个路由。

客户端凭证授权令牌

客户端凭证授权适用于机器对机器的认证，例如，你可以在调度任务中使用这种授权来通过 API 执行维护任务。要使用这个方法，首先需要在 `app/Http/Kernel.php` 中添加新的中间件到 `$routeMiddleware`：

```
use Laravel\Passport\Http\Middleware\CheckClientCredentials;

protected $routeMiddleware = [
    'client' => CheckClientCredentials::class,
];
```

然后将这个中间件应用到路由：

```
Route::get('/user', function(Request $request) {
    ...
})->middleware('client');
```

要获取令牌，发送请求到 `oauth/token`：

```
$guzzle = new GuzzleHttp\Client;

$response = $guzzle->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'client_credentials',
    ],
]);
```

```
'client_id' => 'client-id',
'client_secret' => 'client-secret',
'scope' => 'your-scope',
],
]);

return json_decode((string) $response->getBody(), true);
```

私人访问令牌

有时候，你的用户可能想要颁发访问令牌给自己而不走典型的授权码重定向流程。允许用户通过应用的 UI 颁发令牌给自己在用户体验你的 API 或者作为更简单的颁发访问令牌方式时会很有用。

注：私人访问令牌总是一直有效的，它们的生命周期在使用 `tokensExpireIn` 或 `refreshTokensExpireIn` 方法时不会修改。

创建私人访问客户端

在你的应用可以颁发私人访问令牌之前，需要创建一个私人访问客户端。你可以通过带 `--personal` 选项的 `passport:client` 命令来实现，如果你已经运行过了 `passport:install` 命令，则不必再运行此命令：

```
php artisan passport:client --personal
```

管理私人访问令牌

创建好私人访问客户端之后，就可以使用 `User` 模型实例上的 `createToken` 方法为给定用户颁发令牌。`createToken` 方法接收令牌名称作为第一个参数，以及一个可选的域数组作为第二个参数：

```
$user = App\User::find(1);

// Creating a token without scopes...
$token = $user->createToken('Token Name')->accessToken;

// Creating a token with scopes...
$token = $user->createToken('My Token', ['place-orders'])->accessToken;
```

JSON API

Passport 还提供了一个 JSON API 用于管理私人访问令牌，你可以将其与自己的前端配对以便为用户提供管理私人访问令牌的后台。下面，我们来概览用于管理私人访问令牌的所有 API。为了方便起见，我们使用 `Axios` 来演示发送 HTTP 请求到 API。

注：如果你不想要实现自己的私人访问令牌前端，可以使用 [前端快速上手教程](#) 在数分钟内打造拥有完整功能的前端。

GET /oauth/scopes

这个路由会返回应用所定义的所有域。你可以使用这个路由来列出用户可以分配给私人访问令牌的所有域：

```
axios.get('/oauth/scopes')
.then(response => {
  console.log(response.data);
});
```

GET /oauth/personal-access-tokens

这个路由会返回该认证用户所创建的所有私人访问令牌，这在列出用户的所有令牌以便编辑或删除时很有用：

```
axios.get('/oauth/personal-access-tokens')
.then(response => {
  console.log(response.data);
});
```

POST /oauth/personal-access-tokens

这个路由会创建一个新的私人访问令牌，该路由要求传入两个参数：令牌的 `name` 和需要分配到这个令牌的 `scopes`：

```
const data = {
  name: 'Token Name',
  scopes: []
};

axios.post('/oauth/personal-access-tokens', data)
.then(response => {
  console.log(response.data.accessToken);
})
.catch (response => {
  // List errors on response...
});
```

DELETE /oauth/personal-access-tokens/{token-id}

这个路由可以用于删除私人访问令牌：

```
axios.delete('/oauth/personal-access-tokens/' + tokenId);
```

路由保护

通过中间件

Passport 提供了一个认证 guard 用于验证输入请求的访问令牌，当你使用 `passport` 驱动配置好 `api` guard 后，只需要在所有路由上指定需要传入有效访问令牌的 `auth:api` 中间件即可：

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

传递访问令牌

调用被 Passport 保护的路由时，应用 API 的消费者需要在请求的 `Authorization` 头中指定它们的访问令牌作为 `Bearer` 令牌。例如：

```
$response = $client->request('GET', '/api/user', [
    'headers' => [
        'Accept' => 'application/json',
        'Authorization' => 'Bearer '.$accessToken,
    ],
]);
```

令牌作用域

定义作用域

作用域（Scope）允许 API 客户端在请求账户授权的时候请求特定的权限集合。例如，如果你在构建一个电子商务应用，不是所有的 API 消费者都需要下订单的能力，取而代之地，你可以让这些消费者只请求访问订单物流状态的权限，换句话说，作用域允许你的应用用户限制第三方应用自身可以执行的操作。

你可以在 `AuthServiceProvider` 的 `boot` 方法中使用 `Passport::tokensCan` 方法定义 API 的作用域。`tokensCan` 方法接收作用域名称数组和作用域描述，作用域描述可以是任何你想要在授权通过页面展示给用户的东西：

```
use Laravel\Passport\Passport;

Passport::tokensCan([
    'place-orders' => 'Place orders',
    'check-status' => 'Check order status',
]);
```

分配作用域到令牌

请求授权码

当使用授权码请求访问令牌时，消费者应该指定他们期望的作用域作为 `scope` 查询字符串参数，`scope` 参数是通过空格分隔的作用域列表：

```
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://your-app.com/oauth/authorize?'.$query);
});
```

颁发私人访问令牌

如果你使用 `User` 模型的 `createToken` 方法颁发私人访问令牌，可以传递期望的作用域数组作为该方法的第二个参数：

```
$token = $user->createToken('My Token', ['place-orders'])->accessToken;
```

检查作用域

Passport 提供了两个可用于验证输入请求是否经过已发放作用域的令牌认证的中间件。开始使用之前，添加如下中间件到 `app/Http/Kernel.php` 文件的 `$routeMiddleware` 属性：

```
'scopes' => \Laravel\Passport\Http\Middleware\CheckScopes::class,
'scope' => \Laravel\Passport\Http\Middleware\CheckForAnyScope::class,
```

检查所有作用域

`scopes` 中间件会分配给一个用于验证输入请求的访问令牌拥有所有列出作用域的路由：

```
Route::get('/orders', function () {
    // Access token has both "check-status" and "place-orders" scopes...
});
```

```
) -> middleware('scopes:check-status,place-orders');
```

检查任意作用域

`scope` 中间件会分配给一个用于验证输入请求的访问令牌拥有至少一个列出作用域的路由:

```
Route::get('/orders', function () {
    // Access token has either "check-status" or "place-orders" scope...
}) -> middleware('scope:check-status,place-orders');
```

检查令牌实例上的作用域

当一个访问令牌认证过的请求进入应用后, 你仍然可以使用经过认证的 `User` 实例上的 `tokenCan` 方法来检查这个令牌是否拥有给定作用域:

```
use Illuminate\Http\Request;

Route::get('/orders', function (Request $request) {
    if ($request->user()->tokenCan('place-orders')) {
        //
    }
});
```

使用 JavaScript 消费 API

构建 API 时, 能够从你的 JavaScript 应用消费你自己的 API 非常有用。这种 API 开发方式允许你自己的应用消费你和其他人分享的同一个 API, 这个 API 可以被你的 Web 应用消费, 也可以被你的移动应用消费, 还可以被第三方应用消费, 以及任何你可能发布在多个包管理器上的 SDK 消费。

通常, 如果你想要从你的 JavaScript 应用消费自己的 API, 需要手动发送访问令牌到应用并在应用的每一个请求中传递它。不过, Passport 提供了一个中间件用于处理这一操作。你所需要做的只是添加这个中间件 `CreateFreshApiToken` 到 `web` 中间件组:

```
'web' => [
    // Other middleware...
    \Laravel\Passport\Http\Middleware\CreateFreshApiToken::class,
],
```

这个 Passport 中间件将会附加 `laravel_token` Cookie 到输出响应, 这个 Cookie 包含加密过的 JWT, Passport 将使用这个 JWT 来认证来自 JavaScript 应用的 API 请求, 现在, 你可以发送请求到应用的 API, 而不必显示传递访问令牌:

```
axios.get('/user')
    .then(response => {
        console.log(response.data);
    });
```

使用这种认证方法时, Axios 会自动发送 `X-CSRF-TOKEN` 头, 此外, Laravel 默认 JavaScript 脚手架引入的 Axios 还会发送 `X-Requested-With` 头。不过你需要确保在 `HTML meta 标签` 中引入了 CSRF 令牌:

```
window.axios.defaults.headers.common = {
    'X-Requested-With': 'XMLHttpRequest',
};
```

注: 如果你使用的是其它 JavaScript 框架, 需要确保每个请求都被配置为发送这两个请求头。

事件

Passport 会在颁发访问令牌和刷新令牌时触发事件, 你可以使用这些事件来处理或撤销数据库中的其它访问令牌, 你可以在应用的 `EventServiceProvider` 中添加监听器到这些事件:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Laravel\Passport\Events\AccessTokenCreated' => [
        'App\Listeners\RevokeOldTokens',
    ],
    'Laravel\Passport\Events\RefreshTokenCreated' => [
        'App\Listeners\PruneOldTokens',
    ],
];
```

测试

Passport 的 `actingAs` 方法可用于指定当前认证用户及其作用域, 传递给 `actingAs` 方法的第一个参数是用户实例, 第二个参数是授权给用户令牌的作用域数组:

```
public function testServerCreation()
```

```
{
    Passport::actingAs(
        factory(User::class)->create(),
        ['create-servers']
    );

    $response = $this->post('/api/create-server');

    $response->assertStatus(200);
}
```

用户授权

简介

除了提供开箱即用的认证服务之外，Laravel 还提供了一个简单的方式来管理授权逻辑以便控制对资源的访问权限。和认证一样，在 Laravel 中实现授权很简单，主要有两种方式：Gate 和 Policy。

可以将 Gate 和 Policy 分别看作路由和控制器，Gate 提供了简单的基于闭包的方式进行授权，而 Policy 和控制器一样，对特定模型或资源上的复杂授权逻辑进行分组，本着由简入繁的思路，我们首先来看 Gate，然后再看 Policy。

不要将 Gate 和 Policy 看作互斥的东西，实际上，在大多数应用中我们会混合使用它们，这很有必要，因为 Gate 通常用于与模型或资源无关的权限，比如访问管理后台，与之相反，Policy 则用于对指定模型或资源的动作进行授权。

Gate

编写 Gate

Gate 是用于判断用户是否有权进行某项操作的闭包，通常使用 Gate 门面定义在 `App\Providers\AuthServiceProvider` 类中。Gate 总是接收用户实例作为第一个参数，还可以接收相关的 Eloquent 模型实例作为额外参数：

```
/**
 * 注册任意认证/授权服务.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Gate::define('update-post', function ($user, $post) {
        return $user->id == $post->user_id;
    });
}
```

Gate 还可以通过使用 `Class@method` 风格的回调字符串定义，和控制器一样：

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Gate::define('update-post', 'PostPolicy@update');
}
```

资源 Gate

你还可以使用 `resource` 方法一次定义多个 Gate 权限：

```
Gate::resource('posts', 'PostPolicy');
```

上面的写法等价于下面的 Gate 定义：

```
Gate::define('posts.view', 'PostPolicy@view');
Gate::define('posts.create', 'PostPolicy@create');
Gate::define('posts.update', 'PostPolicy@update');
Gate::define('posts.delete', 'PostPolicy@delete');
```

默认情况下, `view`、`create`、`update` 和 `delete` 权限会被定义, 你也可以通过传递一个数组作为第三个参数到 `resource` 方法来覆盖或添加其他权限。这个数组的键就是权限名称, 而对应的键值就是权限方法的名称。例如, 下面的代码将会创建两个新的 Gate 定义 —

— `posts.image` 和 `posts.photo`:

```
Gate::resource('posts', 'PostPolicy', [
    'image' => 'updateImage',
    'photo' => 'updatePhoto',
]);
```

授权动作

要使用 Gate 授权某个动作, 可以使用 `allows` 或 `denies` 方法, 需要注意的是你可以不传用户实例到这些方法, Laravel 会自动将用户实例 (当前用户) 传递到 Gate 闭包:

```
if (Gate::allows('update-post', $post)) {
    // 当前用户可以更新文章...
}

if (Gate::denies('update-post', $post)) {
    // 当前用户不能更新文章...
}
```

注: 这种情况下, 对于未登录用户所有权限校验都会返回 `false`。

如果你想要判断指定用户 (非当前用户) 是否有权进行某项操作, 可以使用 Gate 门面上的 `forUser` 方法:

```
if (Gate::forUser($user)->allows('update-post', $post)) {
    // 当前用户可以更新文章...
}

if (Gate::forUser($user)->denies('update-post', $post)) {
    // 当前用户不能更新文章...
}
```

创建 Policy

生成 Policy 类

Policy (策略) 是用于组织基于特定模型或资源的授权逻辑类, 例如, 如果你开发的是一个博客应用, 可以有一个 `Post` 模型和与之对应的 `PostPolicy` 来授权用户创建或更新博客的动作。

我们使用 Artisan 命令 `make:policy` 来生成一个 Policy 类, 生成的 Policy 类位于 `app/Policies` 目录下, 如果这个目录之前不存在, Laravel 会自动为我们创建:

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

`make:policy` 命令会生成一个空的 Policy 类, 如果你想要生成一个包含基本 CRUD 策略方法的 Policy 类, 在执行该命令的时候可以通过 `--model` 指定相应模型:

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

注: 所有策略类都通过服务容器进行解析, 这样在策略类的构造函数中就可以通过类型提示进行依赖注入。

注册 Policy 类

Policy 类创建之后, 需要注册到容器。Laravel 自带的 `AuthServiceProvider` 包含了一个 `policies` 属性来映射 Eloquent 模型及与之对应的 Policy 类。注册 Policy 将会告知 Laravel 在授权给定模型动作时使用哪一个策略类:

```
<?php

namespace App\Providers;

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

class AuthServiceProvider extends ServiceProvider
{
    /**
     * 应用的策略映射.
     *
     * @var array
     */
    protected $policies = [
        Post::class => PostPolicy::class,
    ];
}
```

```

 * @translator laravelacademy.org
 */
protected $policies = [
    Post::class => PostPolicy::class,
];

/**
 * 注册任意认证/授权服务.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    /**
}
}

```

编写 Policy

Policy 方法

Policy 类被注册后，还要为每个授权动作添加方法，例如，我们为用户更新 Post 实例这一动作在 PostPolicy 中定义一个 update 方法。update 方法会接收一个 User 实例和一个 Post 实例作为参数，并且返回 true 或 false 以表明该用户是否有权限对给定 Post 进行更新。因此，在这个例子中，我们验证用户的 id 和文章对应的 user_id 是否匹配：

```

<?php

namespace App\Policies;

use App\User;
use App\Post;

class PostPolicy
{
    /**
     * 判断给定文章是否可以被用户更新.
     *
     * @param \App\User $user
     * @param \App\Post $post
     * @return bool
     * @translator laravelacademy.org
     */
    public function update(User $user, Post $post)
    {
        return $user->id === $post->user_id;
    }
}

```

你可以继续在 Policy 类中为授权的权限定义更多需要的方法，例如，你可以定义 view 或者 delete 等方法来授权多个 Post 动作，方法名不限。注：如果你在使用 Artisan 命令生成策略类的时候使用了 --model 选项，那么策略类中就会包含了 view、create、update 和 delete 授权动作方法。

不带模型的方法

有些策略方法只接收当前认证的用户，并不接收授权的模型实例作为参数，这种用法在授权 create 动作的时候很常见。例如，创建一篇博客的时候，你可能想要检查当前用户是否有权创建新博客。

当定义不接收模型实例的策略方法时，例如 create 方法，可以这么做：

```

/**
 * 判断当前用户是否可以创建文章.
 *
 * @param \App\User $user
 * @return bool
 */
public function create(User $user)
{
    /**
}

```

策略过滤器

对特定用户，你可能想要在一个策略方法中对其授权所有权限，比如后台管理员。要实现这个功能，需要在 Policy 类中定义一个 `before` 方法，`before` 方法会在 Policy 类的所有其他方法执行前执行，从而确保在其他策略方法调用前执行其中的逻辑：

```
public function before($user, $ability)
{
    if ($user->isSuperAdmin()) {
        return true;
    }
}
```

如果你想要禁止所有授权，可以在 `before` 方法中返回 `false`。如果返回 `null`，该授权会落入策略方法。

注：如果 Policy 类没有包含与待检查权限名称相匹配的授权方法时，该 Policy 类的 `before` 方法将不会被调用。

使用 Policy 授权动作

通过 User 模型

Laravel 自带的 `User` 模型提供了两个方法用于授权动作：`can` 和 `cant`。`can` 方法接收你想要授权的动作和对应的模型作为参数。例如，下面的例子我们判断用户是否被授权更新给定的 `Post` 模型：

```
if ($user->can('update', $post)) {
    //
}
```

如果给定模型对应的策略已经注册，则 `can` 方法会自动调用相应的策略并返回布尔结果。如果给定模型没有任何策略被注册，`can` 方法将会尝试调用与动作名称相匹配的 Gate 闭包。

不依赖模型的动作

有些动作比如 `create` 并不需要依赖给定模型实例，在这些场景中，可以传递一个类名到 `can` 方法，这个类名会在进行授权的时候用于判断使用哪一个策略：

```
use App\Post;

if ($user->can('create', Post::class)) {
    // Executes the "create" method on the relevant policy...
}
```

通过中间件

Laravel 提供了一个可以在请求到达路由或控制器之前进行授权的中间件——`Illuminate\Auth\Middleware\Authorize`，默认情况下，这个中间件在 `App\Http\Kernel` 类中被分配了一个 `can` 别名，下面我们来探究如何使用 `can` 中间件授权用户更新博客文章动作：

```
use App\Post;

Route::put('/post/{post}', function (Post $post) {
    // The current user may update the post...
})->middleware('can:update,post');
```

在这个例子中，我们传递了两个参数给 `can` 中间件，第一个是我们想要授权的动作名称，第二个是我们想要传递给策略方法的路由参数。在这个例子中，由于我们使用了 `隐式模型绑定`，`Post` 模型将会被传递给策略方法，如果没有对用户进行给定动作的授权，中间件将会生成并返回一个状态码为 `403` 的 HTTP 响应。

不依赖模型的动作

同样，对那些不需要传入模型实例的动作如 `create`，需要传递类名到中间件，类名将会在授权动作的时候用于判断使用哪个策略：

```
Route::post('/post', function () {
    // The current user may create posts...
})->middleware('can:create,App\Post');
```

通过控制器辅助函数

除了提供给 `User` 模型的辅助函数，Laravel 还为继承自 `App\Http\Controllers\Controller` 基类的所有控制器提供了 `authorize` 方法，和 `can` 方法类似，该方法接收你想要授权的动作名称以及相应模型实例作为参数，如果动作没有被授权，`authorize` 方法将会抛出 `Illuminate\Auth\Access\AuthorizationException`，Laravel 默认异常处理器将会将其转化为状态码为 `403` 的 HTTP 响应：

```
<?php

namespace App\Http\Controllers;

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

class PostController extends Controller
{
    /**
     *
     */
}
```

```

 * 更新给定博客文章.
 *
 * @param Request $request
 * @param Post $post
 * @return Response
 * @translator laravelacademy.org
 */
public function update(Request $request, Post $post)
{
    $this->authorize('update', $post);

    // The current user can update the blog post...
}

```

不依赖模型的动作

和之前讨论的一样，类似 `create` 这样的动作不需要传入模型实例参数，在这些场景中，可以传递类名给 `authorize` 方法，该类名将会在授权动作时判断使用哪个策略：

```

/**
 * 创建一篇新的博客文章.
 *
 * @param Request $request
 * @return Response
 */
public function create(Request $request)
{
    $this->authorize('create', Post::class);

    // The current user can create blog posts...
}

```

通过 Blade 模板

编写 Blade 模板的时候，你可能想要在用户被授权特定动作的情况下才显示对应的视图模板部分，例如，你可能想要在用户被授权更新权限的情况下才显示更新表单。在这种情况下，你可以使用 `@can` 和 `@cannot` 指令：

```

@can('update', $post)
    <!-- 当前用户可以更新文章 -->
@elsecan('create', App\Post::class)
    <!-- 当前用户可以创建新文章 -->
@endcan

@cannot('update', $post)
    <!-- 当前用户不能更新文章 -->
@elsecannot('create', App\Post::class)
    <!-- 当前用户不能创建新文章 -->
@endcannot

```

这种写法可看作是 `@if` 和 `@unless` 语句的缩写，上面的 `@can` 和 `@cannot` 语句与下面的语句等价：

```

@if (Auth::user()->can('update', $post))
    <!-- 当前用户可以更新文章 -->
@endif

@unless (Auth::user()->can('update', $post))
    <!-- 当前用户不能更新文章 -->
@endunless

```

不依赖模型的动作

和其它授权方法一样，如果授权动作不需要传入模型实例的情况下可以传递类名给 `@can` 和 `@cannot` 指令：

```

@can('create', App\Post::class)
    <!-- 当前用户可以创建文章 -->
@endcan

@cannot('create', App\Post::class)

```

```
<!-- 当前用户不能创建文章 -->
@endcannot
```

加密

简介

Laravel 的加密器使用 OpenSSL 来提供 AES-256 和 AES-128 加密。强烈建议使用 Laravel 自带的加密设置，不要尝试推出自己“土生土长”的加密算法。所有 Laravel 加密过的值都使用消息授权码（MAC）进行签名以便底层值一经加密就不能修改。

配置

在使用 Laravel 的加密器之前，必须在配置文件 `config/app.php` 中设置 `key` 选项为 32 位随机字符串。可以使用 `php artisan key:generate` 命令来生成这个 key，该 Artisan 命令会使用 PHP 的安全随机字节生成器来构建 key 的值。如果这个值没有被设置，所有 Laravel 加密过的值都是不安全的。

使用加密器

加密

你可以使用辅助函数 `encrypt` 对数据进行加密，所有加密值都使用 OpenSSL 和 AES-256-CBC 密码（cipher）进行加密。此外，所有加密值都通过一个消息认证码（MAC）来进行签名以防止对加密字符串的任何修改。

```
<?php

namespace App\Http\Controllers;

use App\User;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * 存储用户安全信息.
     *
     * @param Request $request
     * @param int $id
     * @return Response
     */
    public function storeSecret(Request $request, $id)
    {
        $user = User::findOrFail($id);

        $user->fill([
            'secret' => encrypt($request->secret)
        ])->save();
    }
}
```

不进行序列化的加密

加密值在加密期间都会经过序列化函数 `serialize` 进行处理，从而允许对对象和数组的加密。因此，非 PHP 客户端接收的加密数据需要进行 `unserialize` 反序列化。如果你想要在加密和解密数据时不进行序列化操作，可以使用 `Crypt` 门面提供的 `encryptString` 和 `decryptString` 方法：

```
use Illuminate\Support\Facades\Crypt;

$encrypted = Crypt::encryptString('Hello world. ');

$decrypted = Crypt::decryptString($encrypted);
```

解密

你可以使用辅助函数 `decrypt` 对加密数据进行解密。如果该值不能被解密，例如 MAC 无效，将会抛出一个 `Illuminate\Contracts\Encryption\DecryptException` 异常：

```
use Illuminate\Contracts\Encryption\DecryptException;

try {
    $decrypted = decrypt($encryptedValue);
} catch (DecryptException $e) {
    //
```

```
}
```

示例

我们在 `routes/web.php` 定义一个路由进行简单测试：

```
Route::get('/test/crypt', function () {
    $raw_str = encrypt('Laravel 学院');
    $decrypted_str = decrypt($raw_str);
    dd(['after_encrypt' => $raw_str, 'after_decrypt' => $decrypted_str]);
});
```

在浏览器中访问 `http://blog.test/test/crypt` 页面输出如下：

[]

```
array:2 [▼
  "after_encrypt" =>
  "eyJpdiI6In1VdEg3RTFFd3BkZXU4SnUralUrWHc9PSIsInZhHVlIjoiNFVPYldXZHNuYWJSRm9vK2R1R1ZKM2N5dmh0NjlcL2ZXMk04aWt5dxQwd1U9IiwibWFjIjoiMGE1Yjc1NDQyYWE1MDF1ZWM0NTc0MTU0N2QzMGU2ZDQ4NWUxYzUwNTB1MWW2MTA0NmIxMmRlNzAxYmNjYjE1YiJ9 ◀"
  "after_decrypt" => " Laravel学院"
]
```

哈希

简介

Laravel 的 `Hash` 门面为存储用户密码提供了安全的 Bcrypt 和 Argon2 哈希算法。如果你正在使用 Laravel 应用自带的 `LoginController` 和 `RegisterController` 控制器，它们将会自动在注册和认证时使用默认的 Bcrypt 算法。

注：Bcrypt 是散列密码的绝佳选择，因为其「工作因子」是可调整的，这意味着随着硬件功能的提升，生成哈希所花费的时间也会增加。

配置

应用默认的哈希驱动配置在配置文件 `config/hashing.php` 中，目前支持两个驱动：`Bcrypt` 和 `Argon2`。

注：Argon2 驱动要求 PHP 7.2.0 或更高版本。

基本使用

可以调用 `Hash` 门面上的 `make` 方法对存储密码进行哈希：

```
<?php

namespace App\Http\Controllers;

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

class UpdatePasswordController extends Controller
{
    /**
     * 更新用户密码.
     *
     * @param Request $request
     * @return Response
     */
    public function update(Request $request)
    {
        // 验证新密码长度...

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

调整 Bcrypt 工作因子

如果你使用的是 Bcrypt 算法的话，`make` 方法允许你通过 `rounds` 选项管理算法的工作因子；不过，默认配置对大部分应用而言都是可接受的：

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

调整 Argon2 工作因子

如果你使用的是 Argon2 算法的话，`make` 方法允许你使用 `memory`、`time` 以及 `threads` 选项来管理算法的工作因子；不过，默认配置对大部分应用而言都是可接受的：

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

注：更多关于这些选项的信息，请查看 [PHP 官方文档](#)。

通过哈希验证密码

`check` 方法允许你验证给定原生字符串和给定哈希是否相等，不过，如果你在使用 Laravel 自带的 [LoginController](#)（详见[登录认证](#)文档），就不需要再直接使用 `check` 方法，因为这个控制器自动调用了该方法：

```
if (Hash::check('plain-text', $hashedPassword)) {
    // 密码匹配...
}
```

检查密码是否需要被重新哈希

`needsRehash` 方法允许你判断哈希计算器使用的工作因子在上次密码被哈希后是否发生改变：

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

注：想了解更多关于哈希算法的原理以及 Laravel 底层密码加密校验实现可参考这篇文章 <http://laravelacademy.org/post/4764.html>

重置密码

简介

想要快速实现该功能？只需要在新安装的 Laravel 应用下运行 `php artisan make:auth`（如果你已经执行过此命令，可忽略），然后在浏览器中访问 <http://your-app.test/register> 或者其他分配给应用的 URL，该命令会生成用户登录注册所需的所有东西，包括密码重置！

大多数 Web 应用都提供了为用户重置密码的功能，Laravel 也不例外，Laravel 提供了用于发送密码重置链接及实现密码重置逻辑的便捷方法，而不需要你在每个应用中自己重复实现。

注：在使用 Laravel 提供的密码重置功能之前，`User` 模型必须使用了 [Illuminate\Notifications\Notifiable trait](#)。

数据库相关

开始之前，先验证 `App\User` 模型实现了 [Illuminate\Contracts\Auth\CanResetPassword](#) 契约。当然，Laravel 自带的 `App\User` 模型已经实现了该接口，并使用 [Illuminate\Auth\Passwords\CanResetPassword trait](#) 来包含实现该接口需要的方法。

生成重置令牌表迁移

接下来，用来存储密码重置令牌的表必须被创建，Laravel 已经自带了这张表的迁移，就存放在 `database/migrations` 目录。所以，你所要做的仅仅是运行迁移：

```
php artisan migrate
```

这张表就是 `password_resets`：

Field	Type	Length	Unsigned	Zerofill	Binary	Allow Null	Key	Default	Extra	Encoding	Collation	Comment
email	VARCHAR	255					M...		None	UTF-8 Unicode	utf8mb4_unicode_ci	
token	VARCHAR	255							None	UTF-8 Unicode	utf8mb4_unicode_ci	
created_at	TIMESTAMP							NULL	None			

路由

Laravel 自带了 `Auth\ForgotPasswordController` 和 `Auth\ResetPasswordController` 控制器（这两个控制器类会通过 `php artisan make:auth` 命令自动生成），分别用于发送密码重置链接邮件和重置用户密码功能。重置密码所需的路由都已经通过 `make:auth` 命令自动生成了：

```
php artisan make:auth
```

对应路由定义在 `Illuminate\Routing\Router` 的 `auth` 方法中：

```
/**
 * Register the typical authentication routes for an application.
 *
 * @return void
 */
public function auth()
{
    // Authentication Routes...
    $this->get('login', ['action' => 'Auth\LoginController@showLoginForm'])->name('login');
    $this->post('login', ['action' => 'Auth\LoginController@login']);
    $this->post('logout', ['action' => 'Auth\LoginController@logout'])->name('logout');

    // Registration Routes...
    $this->get('register', ['action' => 'Auth\RegisterController@showRegistrationForm'])->name('register');
    $this->post('register', ['action' => 'Auth\RegisterController@register']);

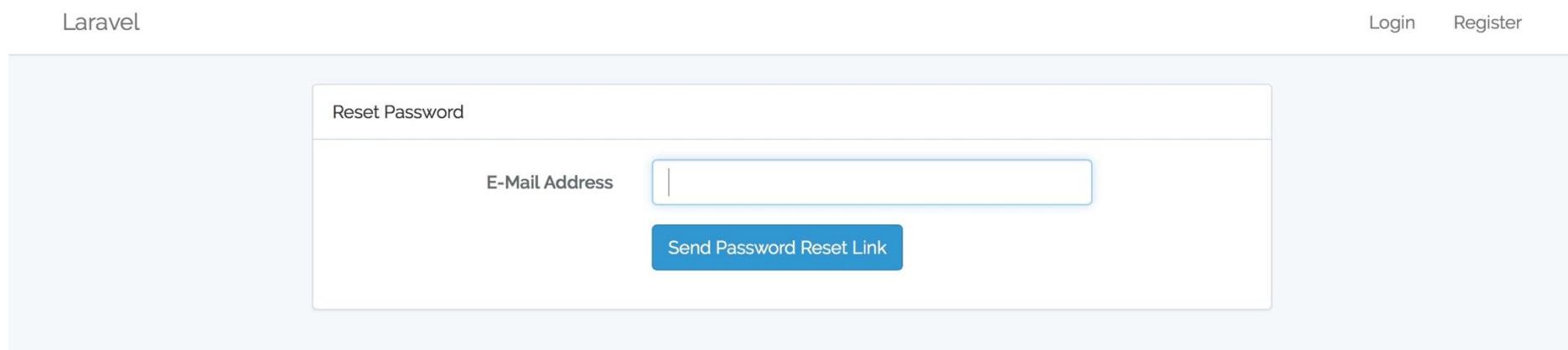
    // Password Reset Routes...
    $this->get('password/reset', ['action' => 'Auth\ForgotPasswordController@showLinkRequestForm'])->name('password.request');
    $this->post('password/email', ['action' => 'Auth\ForgotPasswordController@sendResetLinkEmail'])->name('password.email');
    $this->get('password/reset/{token}', ['action' => 'Auth\ResetPasswordController@showResetForm'])->name('password.reset');
    $this->post('password/reset', ['action' => 'Auth\ResetPasswordController@reset']);
}
```

视图

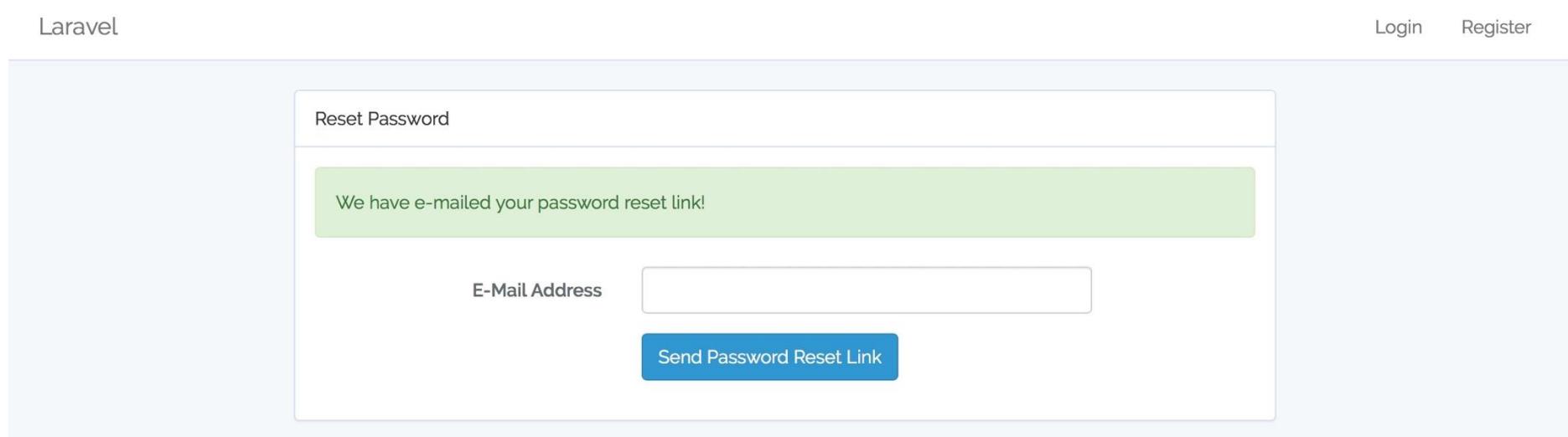
和路由一样，重置密码所需的视图文件也通过 `make:auth` 命令一并生成了，这些视图文件位于 `resources/views/auth/passwords` 目录下，你可以按照所需对生成的文件进行相应修改。

重置密码

定义好重置用户密码路由和视图后，只需要在浏览器中通过 `/password/reset` 访问这个入口路由。框架自带的 `ForgotPasswordController` 已经包含了发送密码重置链接邮件的逻辑，`ResetPasswordController` 包含了重置用户密码的逻辑：



输入注册邮箱，点击发送密码重置链接，就会发送密码重置链接到该邮箱：



打开邮箱会收到这样一封重置密码邮件：



Hello!

You are receiving this email because we received a password reset request for your account.

[Reset Password](#)

If you did not request a password reset, no further action is required.

Regards,
Laravel

If you're having trouble clicking the "Reset Password" button, copy and paste the URL below into your web browser:

<http://localhost/password/reset/2c52e04f18cf459140144a0bbef2e098e1623e9111615b8d24fdbb0410b6f23>

点击重置密码按钮，即可进入重置密码页面：

填写表单提交之后即可重置密码。

密码被重置后，用户将会自动登录到应用并重定向到 `/home`。你可以通过定义 `ResetPasswordController` 的 `redirectTo` 属性来自定义密码重置成功后的跳转链接：

```
protected $redirectTo = '/dashboard';
```

注：默认情况下，密码重置令牌一小时内有效，你可以通过修改 `config/auth.php` 文件中的选项 `expire` 来改变有效时间。

自定义

自定义认证 Guard

在配置文件 `auth.php` 中，可以配置多个“guards”，以便用于实现基于多用户表的独立认证，你可以通过重写内置的 `ResetPasswordController` 控制器上的 `guard` 方法来使用你所选择的 `guard`，该方法将会返回一个 `guard` 实例：

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

protected function guard()
{
    return Auth::guard('guard-name');
}
```

自定义密码 broker

在配置文件 `auth.php` 中，可以配置多个密码，以便用于重置多个用户表的密码 `broker`，同样，可以通过重写自带的 `ForgotPasswordController` 和 `ResetPasswordController` 控制器中的 `broker` 方法来使用你所选择的 `broker`：

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

/**
```

```

 * 获取密码重置期间所使用的 broker.
 *
 * @return PasswordBroker
 * @translator laravelacademy.org
 */
protected function broker()
{
    return Password::broker('name');
}

```

自定义密码重置邮件

你可以很方便地编辑发送密码重置链接给用户的通知类实现自定义密码重置邮件，要实现这一功能，需要重写 `User` 模型上的 `sendPasswordResetNotification` 方法，在这个方法中，可以使用任何你所喜欢的通知类发送通知，该方法接收的第一个参数是密码重置 `$token`:

```

/**
 * 发送密码重置通知.
 *
 * @param string $token
 * @return void
 */
public function sendPasswordResetNotification($token)
{
    $this->notify(new ResetPasswordNotification($token));
}

```

九、进阶系列

Artisan 控制台

简介

Artisan 是 Laravel 自带的命令行接口，它为我们在开发过程中提供了很多有用的命令。想要查看所有可用的 Artisan 命令，可使用 `list` 命令：

```
php artisan list
```

每个命令都可以用 `help` 指令显示命令描述及命令参数和选项。想要查看帮助界面，只需要在命令前加上 `help` 就可以了：

```
php artisan help migrate
```

Laravel REPL

所有的 Laravel 应用都提供了 Tinker —— 一个由 `PsySH` 扩展包驱动的 REPL（Read-Eval-Print Loop，即终端命令行“读取-求值-输出”循环工具）。Tinker 允许你通过命令行与整个 Laravel 应用进行交互，包括 Eloquent ORM、任务、事件等等。要进入 Tinker 环境，运行 `tinker` 命令即可：

```
php artisan tinker
```

注：关于 Tinker 的使用方法可以查看这篇教程[使用 Php Artisan Tinker 来调试你的 Laravel](#)

编写命令

除了 Artisan 提供的系统命令之外，还可以编写自己的命令。自定义命令通常存放在 `app\Console\Commands` 目录下；当然，你也可以自己选择存放位置，只要该命令类可以被 Composer 自动加载即可。

生成命令

要创建一个新命令，你可以使用 Artisan 命令 `make:command`，该命令会在 `app\Console\Commands` 目录下创建一个新的命令类。如果该目录不存在，不用担心，它将会在你首次运行 Artisan 命令 `make:command` 时被创建。生成的命令将会包含默认的属性设置以及所有命令都共有的方法：

```
php artisan make:command SendEmails
```

命令结构

命令生成以后，需要填写该类的 `signature` 和 `description` 属性，这两个属性在调用 `list` 显示命令的时候会被用到。`handle` 方法在命令执行时被调用，你可以将所有命令逻辑都放在这个方法里面。

注：为了更好地实现代码复用，最佳实践是保持控制台命令的轻量并让它们延迟到应用服务中完成任务。在下面的例子中，我们注入了一个服务类来完成发送邮件这样的“繁重”任务。

下面让我们来看一个例子，注意我们可以在命令类的构造函数中注入任何依赖，Laravel 服务容器将会在构造函数中自动注入所有依赖类型提示：

```
<?php
```

```

namespace App\Console\Commands;

use App\User;
use App\DripEmailer;
use Illuminate\Console\Command;

class SendEmails extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     * @translator laravelacademy.org
     */
    protected $signature = 'email:send {user}';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Send drip e-mails to a user';

    /**
     * The drip e-mail service.
     *
     * @var DripEmailer
     */
    protected $drip;

    /**
     * Create a new command instance.
     *
     * @param DripEmailer $drip
     * @return void
     */
    public function __construct(DripEmailer $drip)
    {
        parent::__construct();

        $this->drip = $drip;
    }

    /**
     * Execute the console command.
     *
     * @return mixed
     */
    public function handle()
    {
        $this->drip->send(User::find($this->argument('user')));
    }
}

```

闭包命令

基于闭包的命令和闭包路由之于控制器一样，为以类的方式定义控制台命令提供了可选方案，在 `app\Console\Kernel.php` 文件的 `commands` 方法中，Laravel 加载了 `routes/console.php` 文件：

```

/**
 * 为应用注册基于闭包的命令.
 *
 * @return void
 */
protected function commands()
{
    require base_path('routes/console.php');
}

```

尽管这个文件没有定义 HTTP 路由，但是它定义了基于控制台的应用入口（和路由作用一样），在这个文件中，你可以使用 `Artisan::command` 方法定义所有基于闭包的路由。`command` 方法接收两个参数——`命令标识` 和接收命令参数和选项的闭包：

```
Artisan::command('build {project}', function ($project) {
```

```
$this->info("Building {$project}!");
});
```

该闭包被绑定到底层命令实例，所以你可以像在完整的命令类中一样访问所有辅助方法。

将上面的代码拷贝到 `routes/console.php` 文件后就可以在终端调用了：

```
sunqiangdeMacBook-Pro:laravel55 sunqiang$ php artisan build project
Building project!
```

类型提示依赖

除了接收命令参数和选项外，闭包命令还可以类型提示服务容器之外解析的额外依赖：

```
use App\User;
use App\DripEmailer;

Artisan::command('email:send {user}', function (DripEmailer $drip, $user) {
    $drip->send(User::find($user));
});
```

闭包命令描述

定义基于闭包的命令时，可以使用 `describe` 方法来添加命令描述，这个描述将会在运行 `php artisan list` 或 `php artisan help` 命令时显示：

```
Artisan::command('build {project}', function ($project) {
    $this->info("Building {$project}!");
})->describe('Build the project');
```

定义期望输入

编写控制台命令的时候，通常通过参数和选项收集用户输入，Laravel 使这项操作变得很方便：在命令中使用 `signature` 属性来定义我们期望的用户输入。`signature` 属性通过一个优雅的、路由风格的语法允许你定义命令的名称、参数以及选项。

参数

所有用户提供的参数和选项都包含在花括号里，下面这个例子定义的命令要求用户输入必选参数 `user`：

```
/**
 * 控制台命令名称
 *
 * @var string
 */
protected $signature = 'email:send {user}';
```

你还可以让该参数可选并定义默认的可选参数值：

```
// 选项参数...
email:send {user?}
// 带默认值的选项参数...
email:send {user=foo}
```

选项

选项，和参数一样，是用户输入的另一种格式，不同之处在于选项前面有两个短划线（`--`），有两种类型的选项：接收值和不接收值的。不接收值的选项一般用作布尔开关。我们来看一个这种类型的选项：

```
/**
 * 控制台命令名称
 *
 * @var string
 */
protected $signature = 'email:send {user} {--queue}';
```

在本例中，`--queue` 开关在调用 Artisan 命令的时候被指定。如果 `--queue` 被传递，对应开关值是 `true`，否则其值是 `false`：

```
php artisan email:send 1 --queue
```

带值的选项

接下来，我们来看一个带值的选项，如果用户必须为选项指定值，需要通过`=`进行分配：

```
/**
 * 控制台命令名称
 *
 * @var string
 */
protected $signature = 'email:send {user} {--queue=}';
```

```
 */
protected $signature = 'email:send {user} {--queue=}';
```

在这个例子中，用户可以通过这样的方式传值：

```
php artisan email:send 1 --queue=default
```

还可以给选项分配默认值，如果用户没有传递值给选项，将会使用默认值：

```
email:send {user} {--queue=default}
```

选项简写

如果想要为命令选项分配一个简写，可以在选项前指定并使用分隔符 | 将简写和完整选项名分开：

```
email:send {user} {--Q|queue}
```

输入数组

如果你想要定义参数和选项以便指定输入数组，可以使用字符 *，首先，让我们看一个指定数组参数的例子：

```
email:send {user*}
```

调用这个方法时，`user` 参数会顺序传递到命令行，例如，下面的命令会设置 `user` 的值为 `['foo', 'bar']`：

```
php artisan email:send foo bar
```

定义一个期望输入数组的选项时，每个传递给命令的选项值都应该加上选项名前缀：

```
email:send {user} {--id=*}
```

```
php artisan email:send --id=1 --id=2
```

输入描述

你可以通过冒号将参数和描述进行分隔的方式分配描述到输入参数和选项，如果你需要一些空间来定义命令，可以通过换行来定义命令：

```
/**
 * 控制台命令名称
 *
 * @var string
 * @translator laravelacademy.org
 */
protected $signature = 'email:send
    {user : The ID of the user}
    {--queue= : Whether the job should be queued}';
```

命令 I/O

获取输入

在命令被执行的时候，很明显，你需要访问命令获取的参数和选项的值。使用 `argument` 和 `option` 方法即可实现：

```
/**
 * 执行控制台命令。
 *
 * @return mixed
 */
public function handle()
{
    $userId = $this->argument('user');

    //
}
```

如果需要以数组方式返回所有参数的值，调用 `arguments` 方法：

```
$arguments = $this->arguments();
```

选项值和参数值的获取一样简单，使用 `option` 方法，要以数组方式返回所有选项值，可以调用 `options` 方法：

```
// 获取指定选项...
$queueName = $this->option('queue');

// 获取所有选项...
```

```
$options = $this->options();
```

如果参数或选项不存在，返回 `null`。

输入提示

除了显示输出之外，你可能还要在命令执行期间要用户提供输入。`ask` 方法将会使用给定问题提示用户，接收输入，然后返回用户输入到命令：

```
/**
 * 执行控制台命令
 *
 * @return mixed
 */
public function handle() {
    $name = $this->ask('What is your name?');
}
```

`secret` 方法和 `ask` 方法类似，但用户输入在终端对他们而言是不可见的，这个方法在问用户一些敏感信息如密码时很有用：

```
$password = $this->secret('What is the password?');
```

让用户确认

如果你需要让用户确认信息，可以使用 `confirm` 方法，默认情况下，该方法返回 `false`，如果用户输入 `y`，则该方法返回 `true`：

```
if ($this->confirm('Do you wish to continue? [y|N]')) {
    //
}
```

自动完成

`anticipate` 方法可用于为可能的选项提供自动完成功能，用户仍然可以选择答案，而不管这些选择：

```
$name = $this->anticipate('What is your name?', ['Taylor', 'Dayle']);
```

给用户提供选择

如果你需要给用户预定义的选择，可以使用 `choice` 方法。用户选择答案的索引，但是返回给你的是答案的值。如果用户什么都没选的话你可以设置默认返回的值：

```
$name = $this->choice('What is your name?', ['Taylor', 'Dayle'], $defaultIndex);
```

编写输出

要将输出发送到控制台，使用 `line`, `info`, `comment`, `question` 和 `error` 方法，每个方法都会使用相应的 ANSI 颜色以作标识。例如，要显示一条信息消息给用户， 使用 `info` 方法在终端显示为绿色：

```
/**
 * 执行控制台命令
 *
 * @return mixed
 */
public function handle() {
    $this->info('Display this on the screen');
}
```

要显示一条错误消息，使用 `error` 方法。错误消息文本通常是红色：

```
$this->error('Something went wrong!');
```

如果你想要显示原生输出，可以使用 `line` 方法，该方法输出的字符不带颜色：

```
$this->line('Display this on the screen');
```

表格布局

`table` 方法使输出多行/列格式的数据变得简单，只需要将头和行传递给该方法，宽度和高度将基于给定数据自动计算：

```
$headers = ['Name', 'Email'];
$users = App\User::all(['name', 'email'])->toArray();
$this->table($headers, $users);
```

表格布局输出如下：

```
sunqiangdeMacBook-Pro:laravel55 sunqiang$ php artisan test
+-----+-----+
| Name | Email           |
+-----+-----+
| 学院君 | yaojinbu@163.com |
+-----+-----+
```

进度条

对需要较长时间运行的任务，显示进度指示器很有用，使用该输出对象，我们可以开始、前进以及停止该进度条。在开始进度时你必须定义步数，然后每走一步进度条前进一格：

```
Artisan::command('test', function () {
    $users = App\User::all();

    $bar = $this->output->createProgressBar(count($users));
    foreach ($users as $user) {
        // $this->performTask($user);

        sleep(3); // 模拟任务执行
        $bar->advance();
    }

    $bar->finish();
    $this->info('task finished!');
});
```

进度条输出如下：

```
sunqiangdeMacBook-Pro:laravel55 sunqiang$ php artisan test
1/1 [██████████] 100%task finished!
```

想要了解更多，查看 [Symfony 进度条组件文档](#)。

注册命令

由于 Console Kernel 的 `commands` 方法会调用 `load` 方法，所有 `app\Console\Commands` 目录下的命令都会通过 Artisan 自动注册。实际上，你可以额外调用 `load` 方法来遍历其他目录下的 Artisan 命令：

```
/**
 * Register the commands for the application.
 *
 * @return void
 */
protected function commands()
{
    $this->load(__DIR__.'/Commands');
    $this->load(__DIR__.'/MoreCommands');

    // ...
}
```

此外，还可以在 `app\Console\Kernel.php` 类的 `$commands` 属性中通过手动添加类名的方式来注册命令。当 Artisan 启动的时候，该属性中列出的命令将会被服务容器解析并通过 Artisan 注册：

```
protected $commands = [
    Commands\SendEmails::class
];
```

通过代码调用命令

有时候你可能希望在 CLI 之外执行 Artisan 命令，比如，你可能希望在路由或控制器中触发 Artisan 命令，这可以使用 Artisan 门面上的 `call` 方法来实现。`call` 方法接收被执行的命令名称作为第一个参数，命令参数数组作为第二个参数，退出码会被返回：

```
Route::get('/foo', function () {
    $exitCode = Artisan::call('email:send', [
        'user' => 1, '--queue' => 'default'
    ]);
});
```

使用 `Artisan` 门面上的 `queue` 方法，你甚至可以将 `Artisan` 命令放到队列中，这样它们就可以通过后台的队列工作者来处理。在使用此方法之前，确保你配置好了队列并且运行了队列监听器：

```
Route::get('/foo', function () {
    Artisan::queue('email:send', [
        'user' => 1, '--queue' => 'default'
    ]);
});
```

还可以指定 `Artisan` 命令被分发到的连接或队列：

```
Artisan::queue('email:send', [
    'user' => 1, '--queue' => 'default'
])->onConnection('redis')->onQueue('commands');
```

传递数组值

如果命令定义了接收数组的选项，可以传递数组值到该选项：

```
Route::get('/foo', function () {
    $exitCode = Artisan::call('email:send', [
        'user' => 1, '--id' => [5, 13]
    ]);
});
```

传递布尔值

如果你需要指定不接收字符串的选项值，例如 `migrate:refresh` 命令上的 `--force` 标识，可以传递布尔值 `true` 或 `false`：

```
$exitCode = Artisan::call('migrate:refresh', [
    '--force' => true,
]);
```

通过其他命令调用命令

有时候你希望从一个已存在的 `Artisan` 命令中调用其它命令。你可以通过使用 `call` 方法来实现这一目的。`call` 方法接收命令名称和数组形式的命令参数：

```
/**
 * 执行控制台命令
 *
 * @return mixed
 */
public function handle() {
    $this->call('email:send', [
        'user' => 1, '--queue' => 'default'
    ]);
}
```

如果你想要调用其它控制台命令并阻止其所有输出，可以使用 `callSilent` 方法。`callSilent` 方法和 `call` 方法用法一致：

```
$this->callSilent('email:send', [
    'user' => 1, '--queue' => 'default'
]);
```

广播

简介

在很多现代 Web 应用中，Web 套接字（`WebSockets`）被用于实现实时更新的用户接口。当一些数据在服务器上被更新，通常一条消息通过 `Websocket` 连接被发送给客户端处理。这为我们提供了一个更强大的、更有效的选择来持续拉取应用的更新。

为帮助你构建这样的应用，Laravel 让通过 `Websocket` 连接广播事件变得简单。广播 Laravel 事件允许你在服务端和客户端 `JavaScript` 框架之间共享同一事件名。

注：在深入了解事件广播之前，确保你已经阅读并理解 Laravel 事件与监听器相关文档。

配置

应用的所有事件广播配置选项都存放在 `config/broadcasting.php` 配置文件中。Laravel 开箱支持多种广播驱动：`Pusher`、`Redis` 以及一个服务于本地开发和调试的 `log` 驱动。此外，还提供了一个 `null` 驱动用于完全禁止事件广播。每一个驱动在 `config/broadcasting.php` 配置文件中都有一个配置示例。

广播服务提供者

在广播任意事件之前，首先需要注册 `App\Providers\BroadcastServiceProvider`。在新安装的 Laravel 应用中，你只需要取消 `config/app.php` 配置文件中 `providers` 数组内对应服务提供者之前的注释即可。该提供者允许你注册广播授权路由和回调。

CSRF 令牌

Laravel Echo 需要访问当前 Session 的 CSRF 令牌（token），如果有效的话，Echo 将会从 JavaScript 变量 `Laravel.csrfToken` 中获取令牌。如果你运行过 Artisan 命令 `make:auth` 的话，该对象定义在 `resources/views/layouts/app.blade.php` 布局文件中。如果你没有使用这个布局，你可以在应用的 HTML 元素 `head` 中定义这样一个 `meta` 标签：

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

驱动预备知识

Pusher

如果你准备通过 Pusher 广播事件，需要使用 Composer 包管理器安装对应的 Pusher PHP SDK：

```
composer require pusher/pusher-php-server "~3.0"
```

接下来，你需要在 `config/broadcasting.php` 配置文件中配置你的 Pusher 证书。一个配置好的 Pusher 示例已经包含在这个文件中，你可以按照这个模板进行修改，指定自己的 Pusher key、secret 和应用 ID 即可。`config/broadcasting.php` 文件的 `pusher` 配置还允许你指定额外的被 Pusher 支持的 `options`，例如 `cluster`：

```
'options' => [
    'cluster' => 'eu',
    'encrypted' => true
],
```

使用 Pusher 和 Laravel Echo 的时候，需要在 `resources/assets/js/bootstrap.js` 文件中安装某个 Echo 实例的时候指定 `pusher` 作为期望的广播：

```
import Echo from "laravel-echo"

window.Pusher = require('pusher-js');

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: 'your-pusher-key'
});
```

Redis

如果你使用 Redis 广播，需要安装 Predis 库：

```
composer require predis/predis
```

Redis 广播使用 Redis 的 pub/sub 功能进行广播；不过，你需要将其和能够接受 Redis 消息的 Websocket 服务器进行配对以便将消息广播到 Websocket 频道。

当 Redis 广播发布事件时，事件将会被发布到指定的频道上，传递的数据是一个 JSON 格式的字符串，其中包含了事件名称、数据明细 `data`、以及生成事件 socket ID 的用户。

Socket.IO

如果你想配对 Redis 广播和 Socket.IO 服务器，则需要在应用的 HTML 元素 `head` 中引入 Socket.IO JavaScript 库。当 Socket.IO 服务器启动后，会自动通过一个标准的 URL 来暴露客户端 JavaScript 库，例如，如果你在同一个域名下运行 Socket.IO 和 Web 应用，可以这样访问客户端 JavaScript 库：

```
<script src="//{{ Request::getHost() }}:6001/socket.io/socket.io.js"></script>
```

接下来，你需要使用 `socket.io` 连接器和 `host` 来实例化 Echo：

```
import Echo from "laravel-echo"

window.Echo = new Echo({
    broadcaster: 'socket.io',
    host: window.location.hostname + ':6001'
});
```

最后，需要运行一个与之兼容的 Socket.IO 服务器。Laravel 并未内置一个 Socket.IO 服务器实现，不过，这里有一个第三方实现的 Socket.IO 驱动：[tlaverdure/laravel-echo-server](#)。

队列预备知识

在开始介绍广播事件之前，还需要配置并运行一个队列监听器。所有事件广播都通过队列任务来完成以便应用的响应时间不受影响。

概念概览

Laravel 的事件广播允许你使用基于驱动的 WebSocket 将服务器端事件广播到客户端 JavaScript 应用。目前，Laravel 使用 Pusher 和 Redis 驱动，这些事件可以通过 JavaScript 包 Laravel Echo 在客户端被轻松消费。

事件通过“频道”进行广播，这些频道可以是公共的，也可以是私有的，应用的任何访问者都可以不经认证和授权注册到一个公共的频道，不过，想要注册到私有频道，用户必须经过认证和授权才能监听该频道。

示例应用

在深入了解每个事件广播组件之前，让我们先通过一个电商网站作为示例对整体有个大致的了解。这里我们不会讨论 Pusher 和 Laravel Echo 的配置细节，这些将会放在本文档的后续部分进行讨论。

在我们的应用中，假设我们有一个页面允许用户查看订单的物流状态，我们还假设当应用进行订单状态更新处理时会触发一个 `ShippingStatusUpdated` 事件：

```
event(new ShippingStatusUpdated($update));
```

ShouldBroadcast 接口

当用户查看某个订单时，我们不希望他们必须刷新页面来查看更新状态。取而代之地，我们希望在创建时将更新广播到应用。因此，我们需要标记 `ShippingStatusUpdated` 事件实现 `ShouldBroadcast` 接口，这样，Laravel 就会在触发事件时广播该事件：

```
<?php

namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class ShippingStatusUpdated implements ShouldBroadcast
{
    /**
     * 物流状态更新信息.
     *
     * @var string
     */
    public $update;
}
```

`ShouldBroadcast` 接口要求事件类定义一个 `broadcastOn` 方法，该方法会返回事件将要广播的频道。事件类生成时一个空的方法存根已经定义，我们所要做的只是填充其细节。我们只想要订单的创建者才能够察看状态更新，所以我们将这个事件广播在一个与订单绑定的私有频道上：

```
/**
 * 获取事件广播的频道.
 *
 * @return array
 */
public function broadcastOn()
{
    return new PrivateChannel('order.'.$this->update->order_id);
}
```

授权频道

记住，用户必须经过授权之后才能监听私有频道。我们可以在 `routes/channels.php` 文件中定义频道授权规则。在本例中，我们需要验证任意试图监听 `order.1` 频道的用户确实是订单的创建者：

```
Broadcast::channel('order.{${orderId}}', function ($user, ${orderId}) {
    return $user->id === Order::findOrNew(${orderId})->user_id;
});
```

`channel` 方法接收两个参数：频道的名称以及返回 `true` 或 `false` 以表明用户是否被授权可以监听频道的回调。

所有授权回调都接收当前认证用户作为第一个参数以及任意额外通配符参数作为随后参数，在本例中，我们使用 `{orderId}` 占位符标识频道名称的 ID 部分是一个通配符。

监听事件广播

接下来要做的就是在 JavaScript 中监听事件。我们可以使用 Laravel Echo 来完成这一工作。首先，我们使用 `private` 方法订阅到私有频道。然后，我们使用 `listen` 方法监听 `ShippingStatusUpdated` 事件。默认情况下，所有事件的公共属性都会包含在广播事件中：

```
Echo.private('order.${orderId}')
    .listen('ShippingStatusUpdated', (e) => {
        console.log(e.update);
    });
});
```

定义广播事件

接下来我们来分解上面的示例应用。

要告诉 Laravel 给定事件应该被广播，需要在事件类上实现 `Illuminate\Contracts\Broadcasting\ShouldBroadcast` 接口，这个接口已经在 Laravel 框架生成的事件类中导入了，你只需要将其添加到事件即可。

`ShouldBroadcast` 接口要求你实现一个方法：`broadcastOn`。该方法应该返回一个事件广播频道或频道数组。这些频道必须是 `Channel`、`PrivateChannel` 或 `PresenceChannel` 的实例，`Channel` 频道表示任意用户都可以订阅的公共频道，而 `PrivateChannels` 或 `PresenceChannels` 则代表需要进行 `频道授权` 的私有频道：

```
<?php

namespace App\Events;

use Illuminate\Broadcasting\Channel;
```

```

use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class ServerCreated implements ShouldBroadcast
{
    use SerializesModels;

    public $user;

    /**
     * 创建一个新的事件实例.
     *
     * @return void
     * @translator laravelacademy.org
     */
    public function __construct(User $user)
    {
        $this->user = $user;
    }

    /**
     * 获取事件广播的频道.
     *
     * @return Channel|array
     */
    public function broadcastOn()
    {
        return new PrivateChannel('user.' . $this->user->id);
    }
}

```

然后，你只需要正常触发这个事件即可。一旦事件被触发，队列任务会自动通过指定广播驱动广播该事件。

广播名称

默认情况下，Laravel 会使用事件的类名来广播事件，不过，你可以通过在事件中定义 `broadcastAs` 方法来自定义广播名称：

```

/**
 * 事件的广播名称.
 *
 * @return string
 */
public function broadcastAs()
{
    return 'server.created';
}

```

如果你使用了 `broadcastAs` 方法来广播事件，需要确保在注册监听器的时候带上了 `.` 前缀字符。这将会告知 Echo 不要在事件之前添加应用的命名空间：

```

.listen('.server.created', function (e) {
    ...
});

```

广播数据

如果某个事件被广播，其所有的 `public` 属性都会按照事件负载（payload）自动序列化和广播，从而允许你从 JavaScript 中访问所有 `public` 数据，举个例子，如果你的事件有一个单独的包含 Eloquent 模型的 `$user` 属性，广播负载定义如下：

```

{
    "user": {
        "id": 1,
        "name": "Patrick Stewart"
        ...
    }
}

```

不过，如果你希望对广播负载有更加细粒度的控制，可以添加 `broadcastWith` 方法到事件，该方法会返回你想要通过事件广播的数组数据：

```

/**

```

```
* 获取广播数据
*
* @return array
*/
public function broadcastWith() {
    return ['id' => $this->user->id];
}
```

广播队列

默认情况下，每个广播事件都会被推送到配置文件 `queue.php` 中指定的默认队列连接对应的默认队列中，你可以通过在事件类上定义一个 `broadcastQueue` 属性来自定义广播使用的队列。该属性需要指定广播时你想要使用的队列名称：

```
/**
 * 事件被推送的队列名称.
 *
 * @var string
 */
public $broadcastQueue = 'your-queue-name';
```

如果你想要使用 `sync` 队列而不是默认的队列驱动来广播事件，可以实现 `ShouldBroadcastNow` 接口来取代 `ShouldBroadcast`：

```
<?php

use Illuminate\Contracts\Broadcasting\ShouldBroadcastNow;

class ShippingStatusUpdated implements ShouldBroadcastNow
{
    //
}
```

广播条件

有时候你想要在指定条件为 `true` 的前提下才广播事件，可以通过添加 `broadcastWhen` 方法到事件类来定义这些条件：

```
/**
 * 判定事件是否广播
 *
 * @return bool
 */
public function broadcastWhen()
{
    return $this->value > 100;
}
```

授权频道

私有频道要求你授权当前认证用户可以监听的频道。这可以通过向 Laravel 发送包含频道名称的 HTTP 请求然后让应用判断该用户是否可以监听频道来实现。使用 `Laravel Echo` 的时候，授权订阅私有频道的 HTTP 请求会自动发送，不过，你也需要定义相应路由来响应这些请求。

定义授权路由

庆幸的是，在 Laravel 中定义响应频道授权请求的路由很简单，在 Laravel 自带的 `BroadcastServiceProvider` 中，你可以看到 `Broadcast::routes` 方法的调用，该方法会注册 `/broadcasting/auth` 路由来处理授权请求：

```
Broadcast::routes();
```

`Broadcast::routes` 方法将会自动将路由放置到 `web` 中间件组，不过，你也可以传递一个路由属性数组到这个方法以便自定义分配的属性：

```
Broadcast::routes($attributes);
```

定义授权回调

接下来，我们需要定义执行频道授权的逻辑，这可以通过应用自带的 `routes/channels.php` 文件来完成。在这个方法中，你可以使用 `Broadcast::channel` 方法来注册频道授权回调：

```
Broadcast::channel('order.{orderId}', function ($user, $orderId) {
    return $user->id === Order::findOrNew($orderId)->user_id;
});
```

`channel` 方法接收两个参数：频道名称和返回 `true` 或 `false` 以标识用户是否授权可以监听该频道的回调。

所有授权回调都接收当前认证用户作为第一个参数以及任意额外通配符参数作为其他参数。在本例中，我们使用占位符 `{orderId}` 来标识频道名称的 ID 部分是一个通配符。

授权回调模型绑定

和 HTTP 路由一样，频道路由也可以使用隐式和显式的路由模型绑定，例如，我们可以直接接收一个真正的 `Order` 模型实例，而不是字符串或数字格式的订单 ID：

```
use App\Order;

Broadcast::channel('order.{order}', function ($user, Order $order) {
    return $user->id === $order->user_id;
});
```

定义频道类

如果应用会消费很多不同的频道，和路由文件一样，`routes/channels.php` 文件会变得越来越臃肿，所以，除了通过闭包定义授权频道之外，我们还可以使用频道类。要生成一个频道类，可以使用 `make:channel` 命令，该命令会将生成的新频道类放到 `App/Broadcasting` 目录下：

```
php artisan make:channel OrderChannel
```

接下来在 `routes/channels.php` 中注册这个频道：

```
use App\Broadcasting\OrderChannel;

Broadcast::channel('order.{order}', OrderChannel::class);
```

最后，我们在频道类的 `join` 方法中定义授权逻辑，`join` 方法里的逻辑和定义在频道授权闭包里的逻辑一样。当然，你也可以使用频道模型绑定：

```
<?php

namespace App\Broadcasting;

use App\User;
use App\Order;

class OrderChannel
{
    /**
     * Create a new channel instance.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Authenticate the user's access to the channel.
     *
     * @param \App\User $user
     * @param \App\Order $order
     * @return array|bool
     */
    public function join(User $user, Order $order)
    {
        return $user->id === $order->user_id;
    }
}
```

注：和 Laravel 中很多其他类一样，频道类可以通过服务容器自动解析，所以，你可以在频道的构造函数对其进行依赖注入的类型提示。

广播事件

定义好事件并标记其实现 `ShouldBroadcast` 接口后，你所要做的就是使用 `event` 方法触发该事件。事件分发器将会关注事件是否实现了 `ShouldBroadcast` 接口，如果是的话就将其推送到广播队列中：

```
event(new ShippingStatusUpdated($update));
```

只广播给其他人

构建使用事件广播的应用时，你还可以使用 `broadcast` 函数替代 `event` 函数，和 `event` 函数一样，`broadcast` 函数将事件分发到服务器端监听器：

```
broadcast(new ShippingStatusUpdated($update));
```

不过，`broadcast` 函数还暴露了 `toOthers` 方法以便允许你从广播接收者中排除当前用户：

```
broadcast(new ShippingStatusUpdated($update))->toOthers();
```

为了更好地理解 `toOthers` 方法，我们先假设有一个任务列表应用，在这个应用中，用户可以通过输入任务名称创建一个新的任务，而要创建一个任务，应用需要发送请求到 `/task`，在这里，会广播任务创建并返回一个 JSON 格式的新任务。当你的 JavaScript 应用从服务端接收到响应后，会直接将这个新任务插入到任务列表：

```
axios.post('/task', task)
  .then((response) => {
    this.tasks.push(response.data);
  });
}
```

不过，还记得吗？我们还广播了任务创建，如果你的 JavaScript 应用正在监听这个事件以便添加任务到任务列表，就会在列表中出现重复任务：一个来自服务端，一个来自广播。你可以通过使用 `toOthers` 方法来解决这个问题，该方法告知广播不要将事件广播给当前用户。

注：事件必须使用了 `Illuminate\Broadcasting\InteractsWithSockets` trait 以便调用 `toOthers` 方法。

配置

当你初始化 Laravel Echo 实例的时候，需要给连接分配一个 socket ID。如果你使用的是 Vue 和 Axios，这个 socket ID 会以 `x-Socket-ID` 头的方式自动添加到每个输出请求。当你调用 `toOthers` 方法时，Laravel 会从请求头中解析这个 socket ID 并告知广播不要广播到带有这个 socket ID 的连接。

如果你没有使用 Vue 和 Axios，则需要手动配置 JavaScript 应用发送 `x-Socket-ID` 请求头。你可以使用 `Echo.socketId` 方法获取这个 socket ID：

```
var socketId = Echo.socketId();
```

接收广播

安装 Laravel Echo

Laravel Echo 是一个 JavaScript 库，有了它之后，订阅频道监听 Laravel 广播的事件将变得轻而易举。你可以通过 NPM 包管理器安装 Echo，在本例中，我们还会安装 `pusher-js` 包，因为我们将会使用 Pusher 进行广播：

```
npm install --save laravel-echo pusher-js
```

安装好 Echo 之后，就可以在应用的 JavaScript 中创建一个新的 Echo 实例，做这件事的最佳位置当然是在 Laravel 自带的 `resources/assets/js/bootstrap.js` 文件的底部：

```
import Echo from "laravel-echo"

window.Echo = new Echo({
  broadcaster: 'pusher',
  key: 'your-pusher-key'
});
```

创建一个使用 `pusher` 连接器的 Echo 实例时，还可以指定一个 `cluster` 以及连接是否需要加密：

```
window.Echo = new Echo({
  broadcaster: 'pusher',
  key: 'your-pusher-key',
  cluster: 'eu',
  encrypted: true
});
```

监听事件

安装并初始化 Echo 之后，就可以开始监听事件广播了。首先，使用 `channel` 方法获取一个频道实例，然后调用 `listen` 方法监听指定事件：

```
Echo.channel('orders')
  .listen('OrderShipped', (e) => {
    console.log(e.order.name);
  });
}
```

如果你想要监听一个私有频道上的事件，可以使用 `private` 方法，你仍然可以在其后调用 `listen` 方法在单个频道监听多个事件：

```
Echo.private('orders')
  .listen(...)
  .listen(...)
  .listen(...);
```

离开频道

要离开一个频道，可以在 Echo 实例上调用 `leave` 方法：

```
Echo.leave('orders');
```

命名空间

你可能已经注意到在上述例子中我们并没有指定事件类的完整命名空间，这是因为 Echo 会默认事件都位于 `App\Events` 命名空间下。不过，你可以在实例化 Echo 的时候通过传递配置选项 `namespace` 来配置根命名空间：

```
window.Echo = new Echo({
```

```
broadcaster: 'pusher',
key: 'your-pusher-key',
namespace: 'App.Other.Namespace'
});
```

另外，使用 Echo 订阅事件的时候可以在事件类前面加上`.Namespace.Event.Class` 前缀，这样你就可以指定完整的类名了：

```
Echo.channel('orders')
.listen('.Namespace.Event.Class', (e) => {
    //
});
```

存在频道（Presence Channel）

存在频道构建于私有频道之上，并且提供了额外功能：获知谁订阅了频道。基于这一点，我们可以构建强大的、协作的应用功能，例如当其他用户访问同一个页面时通知当前用户。

授权存在频道

所有存在频道同时也是私有频道，因此，用户必须被[授权访问权限](#)。不过，当定义存在频道的授权回调时，如果用户被授权加入频道不要返回`true`，取而代之地，你应该返回关于该用户的数据数组。

授权回调返回的数据在 JavaScript 应用的存在频道事件监听器中使用，如果用户没有被授权加入存在频道，应该返回`false`或`null`：

```
Broadcast::channel('chat.{roomId}', function ($user, $roomId) {
    if ($user->canJoinRoom($roomId)) {
        return ['id' => $user->id, 'name' => $user->name];
    }
});
```

加入存在频道

要加入存在频道，可以使用 Echo 的`join`方法，`join`方法会返回一个`PresenceChannel`实现，并暴露`listen`方法，从而允许你注册到`here`、`joining`和`leaving`事件：

```
Echo.join(`chat.${roomId}`)
.here((users) => {
    //
})
.joining((user) => {
    console.log(user.name);
})
.leaving((user) => {
    console.log(user.name);
});
```

`here` 回调会在频道加入成功后立即执行，并接收一个包含所有其他订阅该频道的用户信息数组。`joining`方法会在新用户加入频道时执行，`leaving`方法则在用户离开频道时执行。

广播到存在频道

存在频道可以像公共或私有频道一样接收事件，以聊天室为例，我们可能想要广播`NewMessage`事件到房间的存在频道，要实现这个功能，需要从事件的`broadcastOn`方法返回`PresenceChannel`实例：

```
/**
 * 获取事件广播频道.
 *
 * @return Channel|array
 * @translator laravelacademy.org
 */
public function broadcastOn()
{
    return new PresenceChannel('room.'.$this->message->room_id);
}
```

和公共或私有频道一样，存在频道事件可以使用`broadcast`函数进行广播。和其他事件一样，你可以使用`toOthers`方法从所有接收广播的用户中排除当前用户：

```
broadcast(new NewMessage($message));
broadcast(new NewMessage($message))->toOthers();
```

你可以通过 Echo 的`listen`方法监听加入事件：

```
Echo.join(`chat.${roomId}`)
.here(...)
.joining(...)
.leaving(...)
```

```
.listen('NewMessage', (e) => {
    //
});
```

客户端事件

有时候你可能想要广播事件到其他连接到的客户端，而不经过 Laravel 应用，这在处理“输入”通知这种事情时尤其有用，比如告知应用的用户其他用户正在给定屏幕输入信息。要广播客户端事件，可以使用 Echo 的 `whisper` 方法：

```
Echo.channel('chat')
    .whisper('typing', {
        name: this.user.name
    });
```

要监听客户端事件，可以使用 `listenForWhisper` 方法：

```
Echo.channel('chat')
    .listenForWhisper('typing', (e) => {
        console.log(e.name);
    });
```

通知

通过配对事件广播和通知，JavaScript 应用可以在事件发生时无需刷新当前页面接收新的通知。在此之前，确保你已经通读广播通知频道文档。配置好使用广播频道的通知后，可以通过使用 Echo 的 `notification` 方法监听广播事件，记住，频道名称应该和接收通知的类名保持一致：

```
Echo.private(`App.User.${userId}`)
    .notification((notification) => {
        console.log(notification.type);
    });
```

在这个例子中，所有通过 `broadcast` 频道发送给 `App\User` 实例的通知都会被这个回调接收。Laravel 框架内置的 `BroadcastServiceProvider` 中包含了一个针对 `App.User.{id}` 频道的频道授权回调。

缓存

配置

Laravel 为不同的缓存系统提供了统一的 API。缓存配置位于 `config/cache.php`。在该文件中你可以指定在应用中默认使用哪个缓存驱动。Laravel 开箱支持主流的缓存后端如 `Memcached` 和 `Redis` 等。

缓存配置文件还包含其他文档化的选项，确保仔细阅读这些选项。默认情况下，Laravel 被配置成使用文件缓存（`file` 驱动），这会将序列化数据和缓存对象存储到文件系统。对于大型应用，建议使用内存缓存如 `Memcached` 或 `APC`，你甚至可以为同一驱动配置多个缓存配置。

驱动预备知识

数据库

使用 `database` 缓存驱动时，你需要设置一张表存储缓存项。下面是该表的 `Schema` 声明：

```
Schema::create('cache', function($table) {
    $table->string('key')->unique();
    $table->text('value');
    $table->integer('expiration');
});
```

注：你还可以使用 Artisan 命令 `php artisan cache:table` 通过相应的 `schema` 生成迁移。

Memcached

使用 Memcached 缓存要求安装了 `Memcached PECL 包`，即 PHP Memcached 扩展。你可以在配置文件 `config/cache.php` 中列出所有 Memcached 服务器：

```
'memcached' => [
    [
        'host' => '127.0.0.1',
        'port' => 11211,
        'weight' => 100
    ],
],
```

你还可以设置 `host` 选项为 UNIX socket 路径，如果你这样做，`port` 选项应该置为 0：

```
'memcached' => [
    [
        'host' => '/var/run/memcached/memcached.sock',
        'port' => 0,
        'weight' => 100
    ],
],
```

```
],
]
```

Redis

使用 Laravel 的 Redis 缓存之前，你需要通过 Composer 安装 [predis/predis](#) 包 (~1.0)。想要了解更多关于 Redis 的配置，查看 Laravel 的 [Redis 文档](#)。

缓存使用

获取缓存实例

[Illuminate\Contracts\Cache\Factory](#) 和 [Illuminate\Contracts\Cache\Repository](#) 契约提供了访问 Laravel 缓存服务的方法。[Factory](#) 契约提供了所有访问应用定义的缓存驱动的方法。[Repository](#) 契约通常是应用中 `cache` 配置文件中指定的默认缓存驱动的一个实现。不过，你还可以使用 `Cache` 门面，这也是我们在整个文档中使用的方式，`Cache` 门面提供了简单方便的方式对底层 Laravel 缓存契约实现进行访问：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

class UserController extends Controller
{
    /**
     * 显示应用所有用户列表.
     *
     * @return Response
     */
    public function index()
    {
        $value = Cache::get('key');

        //
    }
}
```

访问多个缓存存储

使用 `Cache` 门面，你可以使用 `store` 方法访问不同的缓存存储器，传入 `store` 方法的键就是 `cache` 配置文件中 `stores` 配置数组里列出的相应的存储器：

```
$value = Cache::store('file')->get('foo');
Cache::store('redis')->put('bar', 'baz', 10);
```

从缓存中获取数据

`Cache` 门面的 `get` 方法用于从缓存中获取缓存项，如果缓存项不存在，返回 `null`。如果需要的话你可以传递第二个参数到 `get` 方法指定缓存项不存在时返回的自定义默认值：

```
$value = Cache::get('key');
$value = Cache::get('key', 'default');
```

你甚至可以传递一个闭包作为默认值，如果缓存项不存在的话闭包的结果将会被返回。传递闭包允许你可以从数据库或其它外部服务获取默认值：

```
$value = Cache::get('key', function() {
    return DB::table(...)->get();
});
```

检查缓存项是否存在

`has` 方法用于判断缓存项是否存在，如果值为 `null` 或 `false` 该方法会返回 `false`：

```
if (Cache::has('key')) {
    //
}
```

数值增加/减少

`increment` 和 `decrement` 方法可用于调整缓存中的整型数值。这两个方法都可以接收第二个参数来指明缓存项数值增加和减少的数目：

```
Cache::increment('key');
Cache::increment('key', $amount);
Cache::decrement('key');
Cache::decrement('key', $amount);
```

获取&存储

有时候你可能想要获取缓存项，但如果请求的缓存项不存在时给它存储一个默认值。例如，你可能想要从缓存中获取所有用户，或者如果它们不存在的话，从数据库获取它们并将其添加到缓存中，你可以通过使用 `Cache::remember` 方法实现：

```
$value = Cache::remember('users', $minutes, function() {
    return DB::table('users')->get();
});
```

如果缓存项不存在，传递给 `remember` 方法的闭包被执行并且将结果放到缓存中。
你还可以使用 `rememberForever` 方法从缓存中获取数据或者将其永久存储起来：

```
$value = Cache::rememberForever('users', function() {
    return DB::table('users')->get();
});
```

获取&删除

如果你需要从缓存中获取缓存项然后删除，你可以使用 `pull` 方法，和 `get` 方法一样，如果缓存项不存在的话返回 `null`：

```
$value = Cache::pull('key');
```

在缓存中存储数据

你可以使用 `Cache` 门面上的 `put` 方法在缓存中存储数据。当你在缓存中存储数据的时候，需要指定数据被缓存的时间（分钟数）：

```
Cache::put('key', 'value', $minutes);
```

除了传递缓存项失效时间，你还可以传递一个代表缓存项有效时间的 PHP `Datetime` 实例：

```
$expiresAt = Carbon::now()->addMinutes(10);
Cache::put('key', 'value', $expiresAt);
```

缓存不存在时存储数据

`add` 方法只会在缓存项不存在的情况下添加数据到缓存，如果数据被成功添加到缓存返回 `true`，否则，返回 `false`：

```
Cache::add('key', 'value', $minutes);
```

永久存储数据

`forever` 方法用于持久化存储数据到缓存，这些值必须通过 `forget` 方法手动从缓存中移除：

```
Cache::forever('key', 'value');
```

注：如果你使用的是 Memcached 驱动，当缓存数据达到上限后永久存储的数据就会被移除。

从缓存中移除数据

你可以使用 `Cache` 门面上的 `forget` 方法从缓存中移除缓存项数据：

```
Cache::forget('key');
```

还可以使用 `flush` 方法清除所有缓存：

```
Cache::flush();
```

注：清除缓存并不管什么缓存键前缀，而是从缓存系统中移除所有数据，所以在使用这个方法时如果其他应用与本应用有共享缓存时需要格外注意。

缓存辅助函数

除了使用 `Cache` 门面或 `缓存契约`，还可以使用全局的 `cache` 函数来通过缓存获取和存储数据。当带有一个字符串参数的 `cache` 函数被调用时，会返回给定键对应的缓存值（取值）：

```
$value = cache('key');
```

如果你提供了键值对数组和一个过期时间给该函数，则会在指定的有效期内存储缓存值（存储）：

```
cache(['key' => 'value'], $minutes);
cache(['key' => 'value'], Carbon::now()->addSeconds(10));
```

测试调用 `cache` 函数时，可以像测试门面一样使用 `Cache::shouldReceive` 方法。

缓存标签

注：缓存标签目前不支持 `file` 或 `database` 缓存驱动，此外，当使用多标签的缓存被设置为永久存储时，使用 `memcached` 驱动的缓存有着最佳性能表现，因为 Memcached 会自动清除陈旧记录。

存储被打上标签的缓存项

缓存标签允许你给相关缓存项打上同一个标签以便于后续清除这些缓存值，被打上标签的缓存可以通过传递一个被排序的标签数组来访问。例如，我们可以通过以下方式在添加缓存的时候设置标签：

```
Cache::tags(['people', 'artists'])->put('John', $john, $minutes);
Cache::tags(['people', 'authors'])->put('Anne', $anne, $minutes);
```

访问被打上标签的缓存项

要获取被打上标签的缓存项，传递同样的有序标签数组到 `tags` 方法然后使用你想要获取的 key 来调用 `get` 方法：

```
$john = Cache::tags(['people', 'artists'])->get('John');
$anne = Cache::tags(['people', 'authors'])->get('Anne');
```

移除被打上标签的数据项

你可以同时清除被打上同一标签/标签列表的所有缓存项，例如，以下语句会移除被打上 `people` 或 `authors` 标签的所有缓存：

```
Cache::tags(['people', 'authors'])->flush();
```

这样，上面设置的 `Anne` 和 `John` 缓存项都会从缓存中移除。

相反，以下语句只移除被打上 `authors` 标签的语句，所以只有 `Anne` 会被移除而 `John` 不会：

```
Cache::tags('authors')->flush();
```

添加自定义缓存驱动

编写驱动

要创建自定义的缓存驱动，首先需要实现 `Illuminate\Contracts\Cache\Store` 契约，所以，我们的 MongoDB 缓存实现看起来会像这样子：

```
<?php

namespace App\Extensions;

use Illuminate\Contracts\Cache\Store;

class MongoStore implements Store
{
    public function get($key) {}
    public function many(array $keys);
    public function put($key, $value, $minutes) {}
    public function putMany(array $values, $minutes);
    public function increment($key, $value = 1) {}
    public function decrement($key, $value = 1) {}
    public function forever($key, $value) {}
    public function forget($key) {}
    public function flush() {}
    public function getPrefix() {}
}
```

我们只需要使用一个 MongoDB 连接来实现其中的每一个方法，想要看如何实现每个方法的示例，可以参考 Laravel 底层源码 `Illuminate\Cache\MemcachedStore`，实现完成后，我们就可以完成自定义驱动注册：

```
Cache::extend('mongo', function($app) {
    return Cache::repository(new MongoStore());
});
```

注：如果你在担心将自定义缓存驱动代码放到哪，可以在 `app` 目录下创建一个 `Extensions` 命名空间。不过，记住 Laravel 并没有一个严格的应用目录结构，你可以基于你的需要自由的组织目录结构。

注册驱动

要通过 Laravel 注册自定义的缓存驱动，可以使用 `cache` 门面上的 `extend` 方法。对 `Cache::extend` 的调用可以在 Laravel 自带的 `App\Providers\AppServiceProvider` 提供的 `boot` 方法中完成，或者，你也可以创建自己的服务提供者来存放扩展——只是别忘了在配置文件 `config/app.php` 中注册服务提供者到 `providers` 数组：

```
<?php

namespace App\Providers;

use App\Extensions\MongoStore;
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\ServiceProvider;

class CacheServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    * @translator laravelacademy.org
    */
```

```

public function boot()
{
    Cache::extend('mongo', function($app) {
        return Cache::repository(new MongoStore());
    });
}

/**
 * Register bindings in the container.
 *
 * @return void
 */
public function register()
{
    //
}
}

```

传递给 `extend` 方法的第一个参数是驱动名称。该值对应配置文件 `config/cache.php` 中的 `driver` 选项。第二个参数是返回 `Illuminate\Cache\Repository` 实例的闭包。该闭包中被传入一个 `$app` 实例，也就是服务容器的一个实例。

扩展被注册后，只需简单更新配置文件 `config/cache.php` 的 `driver` 选项为自定义扩展名称即可。

缓存事件

要在每次缓存操作时执行代码，你可以监听缓存触发的事件，通常，你可以将这些缓存处理器代码放到 `EventServiceProvider` 中：

```

/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Cache\Events\CacheHit' => [
        'App\Listeners\LogCacheHit',
    ],
    'Illuminate\Cache\Events\CacheMissed' => [
        'App\Listeners\LogCacheMissed',
    ],
    'Illuminate\Cache\Events\KeyForgotten' => [
        'App\Listeners\LogKeyForgotten',
    ],
    'Illuminate\Cache\Events\KeyWritten' => [
        'App\Listeners\LogKeyWritten',
    ],
];

```

集合

简介

`Illuminate\Support\Collection` 类为处理数组数据提供了流式、方便的封装。例如，查看下面的代码，我们使用辅助函数 `collect` 创建一个新的集合实例，为每一个元素运行 `strtoupper` 函数，然后移除所有空元素：

```

$collection = collect(['taylor', 'abigail', null])->map(function ($name) {
    return strtoupper($name);
})->reject(function ($name) {
    return empty($name);
});

```

正如你所看到的，`Collection` 类允许你使用方法链对底层数组执行匹配和移除操作，通常，每个 `Collection` 方法都会返回一个新的 `Collection` 实例。

创建集合

正如上面所提到的，辅助函数 `collect` 为给定数组返回一个新的 `Illuminate\Support\Collection` 实例，所以，创建集合很简单：

```
$collection = collect([1, 2, 3]);
```

注：默认情况下，`Eloquent` 查询的结果总是返回 `Collection` 实例。

扩展集合

集合是“macroable”的，这意味着我们可以在运行时动态添加方法到 `Collection` 类，例如，下面的代码添加了 `toUpper` 方法到 `Collection` 类：

```
use Illuminate\Support\Str;

Collection::macro('toUpper', function () {
    return $this->map(function ($value) {
        return Str::upper($value);
    });
});

$collection = collect(['first', 'second']);

$upper = $collection->toUpper();

// ['FIRST', 'SECOND']
```

通常，我们需要在 [服务提供者](#) 中声明集合宏。

集合方法

本文档接下来的部分将会介绍 `Collection` 类上每一个有效的方法，所有这些方法都可以以方法链的方式流式操作底层数组。此外，几乎每个方法返回一个新的 `Collection` 实例，从而允许你在必要的时候保持原来的集合备份。

方法列表

all()

`all` 方法简单返回集合表示的底层数组：

```
collect([1, 2, 3])->all();
// [1, 2, 3]
```

avg()

`avg` 方法返回所有集合项的平均值：

```
$average = collect([('foo' => 10), ['foo' => 10], ['foo' => 20], ['foo' => 40])->avg('foo');

// 20

$average = collect([1, 1, 2, 4])->avg();

// 2
```

average()

`avg` 方法的别名。

chunk()

`chunk` 方法将一个集合分割成多个小尺寸的小集合：

```
$collection = collect([1, 2, 3, 4, 5, 6, 7]);
$chunks = $collection->chunk(4);
$chunks->toArray();
// [[1, 2, 3, 4], [5, 6, 7]]
```

当处理栅栏系统如 `Bootstrap` 时该方法在 [视图](#) 中尤其有用，建设你有一个想要显示在栅栏中的 `Eloquent` 模型集合：

```
@foreach ($products->chunk(3) as $chunk)
    <div class="row">
        @foreach ($chunk as $product)
            <div class="col-xs-4">{{ $product->name }}</div>
        @endforeach
    </div>
@endforeach
```

collapse()

`collapse` 方法将一个多维数组集合收缩成一个一维数组：

```
$collection = collect([[1, 2, 3], [4, 5, 6], [7, 8, 9]]);
$collapsed = $collection->collapse();
$collapsed->all();
// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

combine()

`combine` 方法可以将一个集合的键和另一个数组或集合的值连接起来：

```
$collection = collect(['name', 'age']);

$combined = $collection->combine(['George', 29]);
```

```
$combined->all();
// ['name' => 'George', 'age' => 29]
```

concat()

`concat` 方法可用于追加给定数组或集合数据到集合末尾:

```
$collection = collect(['John Doe']);

$concatenated = $collection->concat(['Jane Doe'])->concat(['name' => 'Johnny Doe']);

$concatenated->all();

// ['John Doe', 'Jane Doe', 'Johnny Doe']
```

contains()

`contains` 方法判断集合是否包含一个给定项:

```
$collection = collect(['name' => 'Desk', 'price' => 100]);

$collection->contains('Desk');
// true

$collection->contains('New York');
// false
```

你还可以传递一个键值对到 `contains` 方法, 这将会判断给定键值对是否存在子集中:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
]);

$collection->contains('product', 'Bookcase');
// false
```

最后, 你还可以传递一个回调到 `contains` 方法来执行自己的真实测试:

```
$collection = collect([1, 2, 3, 4, 5]);
$collection->contains(function ($key, $value) {
    return $value > 5;
});
// false
```

`contains` 方法在检查值的时候使用「宽松」比较, 这意味着一个包含整型值的字符串和同样的整型值是相等的 (例如, '`1`' 和 `1` 相等)。要想进行严格比较, 可以使用 `containsStrict` 方法。

containsStrict()

这个方法和 `contains` 方法签名一样, 不同之处在于所有值都是「严格」比较。

count()

`count` 方法返回集合中所有项的总数:

```
$collection = collect([1, 2, 3, 4]);
$collection->count();
// 4
```

crossJoin()

`crossJoin` 方法会在给定数组或集合之间交叉组合集合值, 然后返回所有可能排列组合的笛卡尔积:

```
$collection = collect([1, 2]);

$matrix = $collection->crossJoin(['a', 'b']);

$matrix->all();

/*
[
    [1, 'a'],
    [1, 'b'],
    [2, 'a'],
    [2, 'b'],
]
*/

$collection = collect([1, 2]);

$matrix = $collection->crossJoin(['a', 'b'], ['I', 'II']);

$matrix->all();

/*
[
    [1, 'a', 'I'],
    [1, 'a', 'II'],
    [1, 'b', 'I'],
    [1, 'b', 'II'],
    [2, 'a', 'I'],
    [2, 'a', 'II'],
    [2, 'b', 'I'],
    [2, 'b', 'II'],
]
*/
```

```
[1, 'a', 'I'],
[1, 'a', 'II'],
[1, 'b', 'I'],
[1, 'b', 'II'],
[2, 'a', 'I'],
[2, 'a', 'II'],
[2, 'b', 'I'],
[2, 'b', 'II'],
]
*/
```

dd()

`dd` 方法会打印集合项并结束脚本执行:

```
$collection = collect(['John Doe', 'Jane Doe']);

$collection->dd();

/*
Collection {
    #items: array:2 [
        0 => "John Doe"
        1 => "Jane Doe"
    ]
}
*/
```

如果你不想要终止脚本执行, 可以使用 `dump` 方法来替代。

dump()

`dump` 方法会打印集合项而不终止脚本执行:

```
$collection = collect(['John Doe', 'Jane Doe']);

$collection->dump();

/*
Collection {
    #items: array:2 [
        0 => "John Doe"
        1 => "Jane Doe"
    ]
}
*/
```

如果你想要在打印集合之后终止脚本执行, 可以使用 `dd` 方法来替代。

diff()

`diff` 方法将集合和另一个集合或原生 PHP 数组以基于值的方式作比较, 这个方法会返回存在于原来集合而不存在于给定集合的值:

```
$collection = collect([1, 2, 3, 4, 5]);
$diff = $collection->diff([2, 4, 6, 8]);
$diff->all();
// [1, 3, 5]
```

diffAssoc()

`diffAssoc` 方法会基于键值将一个集合和另一个集合或原生 PHP 数组进行比较。该方法返回只存在于第一个集合中的键值对:

```
$collection = collect([
    'color' => 'orange',
    'type' => 'fruit',
    'remain' => 6
]);

$diff = $collection->diffAssoc([
    'color' => 'yellow',
    'type' => 'fruit',
    'remain' => 3,
    'used' => 6
]);

$diff->all();
// ['color' => 'orange', 'remain' => 6]
```

diffKeys()

`diffKeys` 方法会基于键将一个集合和另一个集合或原生 PHP 数组进行比较。该方法会返回只存在于第一个集合的键值对:

```
$collection = collect([
    'one' => 10,
```

```
'two' => 20,
'three' => 30,
'four' => 40,
'five' => 50,
]);

$diff = $collection->diffKeys([
    'two' => 2,
    'four' => 4,
    'six' => 6,
    'eight' => 8,
]);
$diff->all();
// ['one' => 10, 'three' => 30, 'five' => 50]
```

each()

`each` 方法迭代集合中的数据项并传递每个数据项到给定回调:

```
$collection = $collection->each(function ($item, $key) {
    //
});
```

如果你想要终止对数据项的迭代, 可以从回调返回 `false`:

```
$collection = $collection->each(function ($item, $key) {
    if /* some condition */ {
        return false;
    }
});
```

eachSpread()

`eachSpread` 方法会迭代集合项, 传递每个嵌套数据项值到给定集合:

```
$collection = collect([['John Doe', 35], ['Jane Doe', 33]]);

$collection->eachSpread(function ($name, $age) {
    //
});
```

你可以通过从回调中返回 `false` 来停止对集合项的迭代:

```
$collection->eachSpread(function ($name, $age) {
    return false;
});
```

every()

`every` 方法可以用于验证集合的所有元素能够通过给定的真理测试:

```
collect([1, 2, 3, 4])->every(function ($value, $key) {
    return $value > 2;
});

// false
```

except()

`except` 方法返回集合中除了指定键的所有集合项:

```
$collection = collect(['product_id' => 1, 'price' => 100, 'discount' => false]);

$filtered = $collection->except(['price', 'discount']);

$filtered->all();

// ['product_id' => 1]
```

与 `except` 相对的是 `only` 方法。

filter()

`filter` 方法通过给定回调过滤集合, 只有通过给定真理测试的数据项才会保留下:

```
$collection = collect([1, 2, 3, 4]);

$filtered = $collection->filter(function ($value, $key) {
    return $value > 2;
});

$filtered->all();

// [3, 4]
```

如果没有提供回调, 那么集合中所有等价于 `false` 的项都会被移除:

```
$collection = collect([1, 2, 3, null, false, '', 0, []]);  
  
$collection->filter()->all();  
  
// [1, 2, 3]
```

和 `filter` 相对的方法是 `reject`。

first()

`first` 方法返回通过真理测试集合的第一个元素:

```
collect([1, 2, 3, 4])->first(function ($value, $key) {  
    return $value > 2;  
});  
// 3
```

你还可以调用不带参数的 `first` 方法来获取集合的第一个元素, 如果集合是空的, 返回 `null`:

```
collect([1, 2, 3, 4])->first();  
// 1
```

firstWhere()

`firstWhere` 方法会返回集合中的第一个元素, 包含键值对:

```
$collection = collect([  
    ['name' => 'Regena', 'age' => 12],  
    ['name' => 'Linda', 'age' => 14],  
    ['name' => 'Diego', 'age' => 23],  
    ['name' => 'Linda', 'age' => 84],  
]);  
  
$collection->firstWhere('name', 'Linda');  
  
// ['name' => 'Linda', 'age' => 14]
```

还可以调用带操作符的 `firstWhere` 方法:

```
$collection->firstWhere('age', '>=', 18);  
  
// ['name' => 'Diego', 'age' => 23]
```

flatMap()

`flatMap` 方法会迭代集合并传递每个值到给定回调, 该回调可以自由编辑数据项并将其返回, 最后形成一个经过编辑的新集合。然后, 这个数组在层级维度被扁平化:

```
$collection = collect([  
    ['name' => 'Sally'],  
    ['school' => 'Arkansas'],  
    ['age' => 28]  
]);  
  
$flattened = $collection->flatMap(function ($values) {  
    return array_map('strtoupper', $values);  
});  
  
$flattened->all();  
  
// ['name' => 'SALLY', 'school' => 'ARKANSAS', 'age' => '28'];
```

flatten()

`flatten` 方法将多维度的集合变成一维的:

```
$collection = collect(['name' => 'taylor', 'languages' => ['php', 'javascript']]);  
  
$flattened = $collection->flatten();  
  
$flattened->all();  
  
// ['taylor', 'php', 'javascript'];
```

还可以选择性传入深度参数:

```
$collection = collect([  
    'Apple' => [  
        ['name' => 'iPhone 6S', 'brand' => 'Apple'],  
    ],  
    'Samsung' => [  
        ['name' => 'Galaxy S7', 'brand' => 'Samsung']  
    ],  
]);
```

```
$products = $collection->flatten(1);

$products->values()->all();

/*
[
    ['name' => 'iPhone 6S', 'brand' => 'Apple'],
    ['name' => 'Galaxy S7', 'brand' => 'Samsung'],
]
*/
```

在本例中，调用不提供深度的 `flatten` 方法也会对嵌套数组进行扁平化处理，返回结果是 `['iPhone 6S', 'Apple', 'Galaxy S7', 'Samsung']`。提供深度允许你严格设置被扁平化的数组层级。

flip()

`flip` 方法将集合的键值做交换：

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$flipped = $collection->flip();

$flipped->all();

// ['taylor' => 'name', 'laravel' => 'framework']
```

forget()

`forget` 方法通过键从集合中移除数据项：

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$collection->forget('name');

$collection->all();

// ['framework' => 'laravel']
```

注：不同于大多数其他的集合方法，`forget` 不返回新的修改过的集合；它只修改所调用的集合。

forPage()

`forPage` 方法返回新的包含给定页数数据项的集合。该方法接收页码数作为第一个参数，每页显示数据项数作为第二个参数：

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9]);

$chunk = $collection->forPage(2, 3);

$chunk->all();

// [4, 5, 6]
```

get()

`get` 方法返回给定键的数据项，如果对应键不存在，返回 `null`：

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('name');

// taylor
```

你可以选择传递默认值作为第二个参数：

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('foo', 'default-value');

// default-value
```

你甚至可以传递回调作为默认值，如果给定键不存在的话回调的结果将会返回：

```
$collection->get('email', function () {
    return 'default-value';
});

// default-value
```

groupBy()

`groupBy` 方法通过给定键分组集合数据项：

```
$collection = collect([
    ['account_id' => 'account-x10', 'product' => 'Chair'],
    ['account_id' => 'account-x10', 'product' => 'Bookcase'],
    ['account_id' => 'account-x11', 'product' => 'Desk'],
```

```
]);
$grouped = $collection->groupBy('account_id');
$grouped->toArray();
/*
[
    'account-x10' => [
        ['account_id' => 'account-x10', 'product' => 'Chair'],
        ['account_id' => 'account-x10', 'product' => 'Bookcase'],
    ],
    'account-x11' => [
        ['account_id' => 'account-x11', 'product' => 'Desk'],
    ],
]
*/

```

除了传递字符串 `key`, 还可以传递一个回调, 回调应该返回分组后的值:

```
$grouped = $collection->groupBy(function ($item, $key) {
    return substr($item['account_id'], -3);
});
$grouped->toArray();
/*
[
    'x10' => [
        ['account_id' => 'account-x10', 'product' => 'Chair'],
        ['account_id' => 'account-x10', 'product' => 'Bookcase'],
    ],
    'x11' => [
        ['account_id' => 'account-x11', 'product' => 'Desk'],
    ],
]
*/

```

多个分组条件可以以一个数组的方式传递, 每个数组元素都会应用到多维数组中的对应层级:

```
$data = new Collection([
    10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1', 'Role_3']],
    20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1', 'Role_2']],
    30 => ['user' => 3, 'skill' => 2, 'roles' => ['Role_1']],
    40 => ['user' => 4, 'skill' => 2, 'roles' => ['Role_2']],
]);
$result = $data->groupBy([
    'skill',
    function ($item) {
        return $item['roles'];
    },
], $preserveKeys = true);
/*
[
    1 => [
        'Role_1' => [
            10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1', 'Role_3']],
            20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1', 'Role_2']],
        ],
        'Role_2' => [
            20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1', 'Role_2']],
        ],
        'Role_3' => [
            10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1', 'Role_3']],
        ],
    ],
    2 => [
        'Role_1' => [
            30 => ['user' => 3, 'skill' => 2, 'roles' => ['Role_1']],
        ],
        'Role_2' => [
            40 => ['user' => 4, 'skill' => 2, 'roles' => ['Role_2']],
        ],
    ],
]
*/

```

```
    ],
],
*/
```

has()

`has` 方法判断给定键是否在集合中存在:

```
$collection = collect(['account_id' => 1, 'product' => 'Desk']);

$collection->has('email');

// false
```

implode()

`implode` 方法连接集合中的数据项。其参数取决于集合中数据项的类型。如果集合包含数组或对象，应该传递你想要连接的属性键，以及你想要放在值之间的“粘合”字符串:

```
$collection = collect([
    ['account_id' => 1, 'product' => 'Desk'],
    ['account_id' => 2, 'product' => 'Chair'],
]);

$collection->implode('product', ', ');

// Desk, Chair
```

如果集合包含简单的字符串或数值，只需要传递“粘合”字符串作为唯一参数到该方法:

```
collect([1, 2, 3, 4, 5])->implode('-');

// '1-2-3-4-5'
```

intersect()

`intersect` 方法返回两个集合的交集，结果集合将保留原来集合的键:

```
$collection = collect(['Desk', 'Sofa', 'Chair']);

$intersect = $collection->intersect(['Desk', 'Chair', 'Bookcase']);

$intersect->all();

// [0 => 'Desk', 2 => 'Chair']
```

intersectByKeys()

`intersectByKeys` 方法会从原生集合中移除任意没有在给定数组或集合中出现的键:

```
$collection = collect([
    'serial' => 'UX301', 'type' => 'screen', 'year' => 2009
]);

$intersect = $collection->intersectByKeys([
    'reference' => 'UX404', 'type' => 'tab', 'year' => 2011
]);

$intersect->all();

// ['type' => 'screen', 'year' => 2009]
```

isEmpty()

如果集合为空的话 `isEmpty` 方法返回 `true`；否则返回 `false`:

```
collect([])->isEmpty();
// true
```

isNotEmpty()

如果集合不为空的话 `isNotEmpty` 方法返回 `true`；否则返回 `false`:

```
collect([])->isNotEmpty();
// false
```

keyBy()

`keyBy` 方法将指定键的值作为集合的键，如果多个数据项拥有同一个键，只有最后一个会出现在新集合里面:

```
$collection = collect([
    ['product_id' => 'prod-100', 'name' => 'desk'],
    ['product_id' => 'prod-200', 'name' => 'chair'],
]);

$keyed = $collection->keyBy('product_id');
```

```
$keyed->all();

/*
[
    'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
]
*/
```

你还可以传递自己的回调到该方法，该回调将会返回经过处理的键的值作为新的集合键：

```
$keyed = $collection->keyBy(function ($item) {
    return strtoupper($item['product_id']);
});

$keyed->all();

/*
[
    'PROD-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'PROD-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
]
*/
```

keys()

`keys` 方法返回所有集合的键：

```
$collection = collect([
    'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$keys = $collection->keys();

$keys->all();

// ['prod-100', 'prod-200']
```

last()

`last` 方法返回通过真理测试的集合的最后一个元素：

```
collect([1, 2, 3, 4])->last(function ($value, $key) {
    return $value < 3;
});
// 2
```

还可以调用无参的 `last` 方法来获取集合的最后一个元素。如果集合为空，返回 `null`：

```
collect([1, 2, 3, 4])->last();
// 4
```

macro()

静态 `macro()` 方法允许你在运行时添加方法到 `Collection` 类，更多细节可以查看[扩展集合](#)部分文档。

make()

静态 `make` 方法会创建一个新的集合实例，细节可查看[创建集合](#)部分文档。

map()

`map` 方法遍历集合并传递每个值给给定回调。该回调可以修改数据项并返回，从而生成一个新的经过修改的集合：

```
$collection = collect([1, 2, 3, 4, 5]);

$multiplied = $collection->map(function ($item, $key) {
    return $item * 2;
});

$multiplied->all();

// [2, 4, 6, 8, 10]
```

注：和大多数集合方法一样，`map` 返回新的集合实例；它并不修改所调用的实例。如果你想要改变原来的集合，使用 `transform` 方法。

mapInto()

`mapInto()` 方法会迭代集合，通过传递值到构造器来为给定类创建新的实例：

```
class Currency
{
    /**
     * Create a new currency instance.
     *
```

```

 * @param string $code
 * @return void
 */
function __construct(string $code)
{
    $this->code = $code;
}

$collection = collect(['USD', 'EUR', 'GBP']);

$currencies = $collection->mapInto(Currency::class);

$currencies->all();

// [Currency('USD'), Currency('EUR'), Currency('GBP')]

```

mapSpread()

`mapSpread` 方法会迭代集合项，传递每个嵌套集合项值到给定回调。在回调中我们可以修改集合项并将其返回，从而通过修改的值组合成一个新的集合：

```

$collection = collect([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]);

$chunks = $collection->chunk(2);

$sequence = $chunks->mapSpread(function ($odd, $even) {
    return $odd + $even;
});

$sequence->all();

// [1, 5, 9, 13, 17]

```

mapToGroups()

`mapToGroups` 方法会通过给定回调对集合项进行分组，回调会返回包含单个键值对的关联数组，从而将分组后的值组合成一个新的集合：

```

$collection = collect([
    [
        'name' => 'John Doe',
        'department' => 'Sales',
    ],
    [
        'name' => 'Jane Doe',
        'department' => 'Sales',
    ],
    [
        'name' => 'Johnny Doe',
        'department' => 'Marketing',
    ]
]);

$grouped = $collection->mapToGroups(function ($item, $key) {
    return [$item['department'] => $item['name']];
});

$grouped->toArray();

/*
[
    'Sales' => ['John Doe', 'Jane Doe'],
    'Marketing' => ['Johnny Doe'],
]
*/

$grouped->get('Sales')->all();

// ['John Doe', 'Jane Doe']

```

mapWithKeys()

`mapWithKeys` 方法对集合进行迭代并传递每个值到给定回调，该回调会返回包含键值对的关联数组：

```

$collection = collect([
    [
        'name' => 'John',
        'department' => 'Sales',
        'email' => 'john@example.com'
    ]
]);

```

```

    ],
    [
        'name' => 'Jane',
        'department' => 'Marketing',
        'email' => 'jane@example.com'
    ]
);

$keyed = $collection->mapWithKeys(function ($item) {
    return [$item['email'] => $item['name']];
});

$keyed->all();

/*
[
    'john@example.com' => 'John',
    'jane@example.com' => 'Jane',
]
*/

```

max()

`max` 方法返回集合中给定键的最大值:

```

$max = collect(['foo' => 10, ['foo' => 20]])->max('foo');

// 20

$max = collect([1, 2, 3, 4, 5])->max();

// 5

```

median()

`median` 方法会返回给定键的中位数:

```

$median = collect(['foo' => 10, ['foo' => 10], ['foo' => 20], ['foo' => 40])->median('foo');

// 15

$median = collect([1, 1, 2, 4])->median();

// 1.5

```

merge()

`merge` 方法合并给定数组到集合。该数组中的任何字符串键匹配集合中的字符串键的将会重写集合中的值:

```

$collection = collect(['product_id' => 1, 'name' => 'Desk']);

$merged = $collection->merge(['price' => 100, 'discount' => false]);

$merged->all();

// ['product_id' => 1, 'name' => 'Desk', 'price' => 100, 'discount' => false]

```

如果给定数组的键是数字，数组的值将会附加到集合后面:

```

$collection = collect(['Desk', 'Chair']);

$merged = $collection->merge(['Bookcase', 'Door']);

$merged->all();

// ['Desk', 'Chair', 'Bookcase', 'Door']

```

min()

`min` 方法返回集合中给定键的最小值:

```

$min = collect(['foo' => 10, ['foo' => 20])->min('foo');

// 10

$min = collect([1, 2, 3, 4, 5])->min();

// 1

```

mode()

`mode` 方法会返回给定键的众数:

```
$mode = collect(['foo' => 10, 'foo' => 10, 'foo' => 20, 'foo' => 40])->mode('foo');

// [10]

$mode = collect([1, 1, 2, 4])->mode();

// [1]
```

nth()

`nth` 方法组合集合中第 `n`-th 个元素创建一个新的集合:

```
$collection = collect(['a', 'b', 'c', 'd', 'e', 'f']);

$collection->nth(4);

// ['a', 'e']
```

还可以传递一个 `offset` (偏移位置) 作为第二个参数:

```
$collection->nth(4, 1);

// ['b', 'f']
```

only()

`only` 方法返回集合中指定键的集合项:

```
$collection = collect(['product_id' => 1, 'name' => 'Desk', 'price' => 100, 'discount' => false]);

$filtered = $collection->only(['product_id', 'name']);

$filtered->all();

// ['product_id' => 1, 'name' => 'Desk']
```

与 `only` 方法相对的是 `except` 方法。

pad()

`pad` 方法将给定值填充数组直到达到指定的最大长度。该方法和 PHP 函数 `array_pad` 类似。

如果你想要把数据填充到左侧，需要指定一个负值长度，如果指定长度绝对值小于等于数组长度那么将不会做任何填充:

```
$collection = collect(['A', 'B', 'C']);

$filtered = $collection->pad(5, 0);

$filtered->all();

// ['A', 'B', 'C', 0, 0]

$filtered = $collection->pad(-5, 0);

$filtered->all();

// [0, 0, 'A', 'B', 'C']
```

partition()

`partition` 方法可以和 PHP 函数 `list` 一起使用，从而将通过真理测试和没通过的分割开来:

```
$collection = collect([1, 2, 3, 4, 5, 6]);

list($underThree, $aboveThree) = $collection->partition(function ($i) {
    return $i < 3;
});
```

pipe()

`pipe` 方法传递集合到给定回调并返回结果:

```
$collection = collect([1, 2, 3]);

$piped = $collection->pipe(function ($collection) {
    return $collection->sum();
});

// 6
```

pluck()

`pluck` 方法为给定键获取所有集合值:

```
$collection = collect([
    'product_id' => 'prod-100', 'name' => 'Desk',
```

```

['product_id' => 'prod-200', 'name' => 'Chair'],
);

$plucked = $collection->pluck('name');

$plucked->all();

// ['Desk', 'Chair']

```

还可以指定你想要结果集合如何设置键：

```

$plucked = $collection->pluck('name', 'product_id');

$plucked->all();

// ['prod-100' => 'Desk', 'prod-200' => 'Chair']

```

如果存在重复键，最后一个匹配的元素将会插入集合：

```

$collection = collect([
    ['brand' => 'Tesla', 'color' => 'red'],
    ['brand' => 'Pagani', 'color' => 'white'],
    ['brand' => 'Tesla', 'color' => 'black'],
    ['brand' => 'Pagani', 'color' => 'orange'],
]);

$plucked = $collection->pluck('color', 'brand');

$plucked->all();

// ['Tesla' => 'black', 'Pagani' => 'orange']

```

pop()

`pop` 方法移除并返回集合中最后面的数据项：

```

$collection = collect([1, 2, 3, 4, 5]);

$collection->pop();

// 5

$collection->all();

// [1, 2, 3, 4]

```

prepend()

`prepend` 方法添加数据项到集合开头：

```

$collection = collect([1, 2, 3, 4, 5]);

$collection->prepend(0);

$collection->all();

// [0, 1, 2, 3, 4, 5]

```

你还可以传递第二个参数到该方法用于设置前置项的键：

```

$collection = collect(['one' => 1, 'two' => 2]);

$collection->prepend(0, 'zero');

$collection->all();

// ['zero' => 0, 'one' => 1, 'two' => 2]

```

pull()

`pull` 方法通过键从集合中移除并返回数据项：

```

$collection = collect(['product_id' => 'prod-100', 'name' => 'Desk']);

$collection->pull('name');

// 'Desk'

$collection->all();

```

```
// ['product_id' => 'prod-100']

push()
push 方法附加数据项到集合结尾:

$collection = collect([1, 2, 3, 4]);

$collection->push(5);

$collection->all();

// [1, 2, 3, 4, 5]
```

```
put()
put 方法在集合中设置给定键和值:

$collection = collect(['product_id' => 1, 'name' => 'Desk']);

$collection->put('price', 100);

$collection->all();

// ['product_id' => 1, 'name' => 'Desk', 'price' => 100]
```

```
random()
random 方法从集合中返回随机数据项:

$collection = collect([1, 2, 3, 4, 5]);

$collection->random();

// 4 - (retrieved randomly)
```

你可以传递一个整型数据到 random 函数来指定返回的数据数目，如果该整型数值大于 1，将会返回一个集合：

```
$random = $collection->random(3);

$random->all();

// [2, 4, 5] - (retrieved randomly)
```

```
reduce()
reduce 方法用于减少集合到单个值，传递每个迭代结果到子迭代:
```

```
$collection = collect([1, 2, 3]);

$total = $collection->reduce(function ($carry, $item) {
    return $carry + $item;
});

// 6
```

在第一次迭代时 \$carry 的值是 null；不过，你可以通过传递第二个参数到 reduce 来指定其初始值：

```
$collection->reduce(function ($carry, $item) {
    return $carry + $item;
}, 4);

// 10
```

```
reject()
reject 方法使用给定回调过滤集合，该回调应该为所有它想要从结果集合中移除的数据项返回 true:
```

```
$collection = collect([1, 2, 3, 4]);
$filtered = $collection->reject(function ($value, $key) {
    return $value > 2;
});

$filtered->all();

// [1, 2]
```

和 reject 方法相对的方法是 filter 方法。

```
reverse()
reverse 方法将集合数据项的顺序颠倒:
```

```
$collection = collect(['a', 'b', 'c', 'd', 'e']);

$reversed = $collection->reverse();

$reversed->all();
```

```
/*
[
    4 => 'e',
    3 => 'd',
    2 => 'c',
    1 => 'b',
    0 => 'a',
]
*/
```

search()

`search` 方法为给定值查询集合，如果找到的话返回对应的键，如果没找到，则返回 `false`:

```
$collection = collect([2, 4, 6, 8]);

$collection->search(4);

// 1
```

上面的搜索使用的是「宽松」比较，要使用「严格」比较，传递 `true` 作为第二个参数到该方法:

```
$collection->search('4', true);
// false
```

此外，你还可以传递自己的回调来搜索通过真理测试的第一个数据项:

```
$collection->search(function ($item, $key) {
    return $item > 5;
});
// 2
```

shift()

`shift` 方法从集合中移除并返回第一个数据项:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->shift();

// 1

$collection->all();

// [2, 3, 4, 5]
```

shuffle()

`shuffle` 方法随机打乱集合中的数据项:

```
$collection = collect([1, 2, 3, 4, 5]);

$shuffled = $collection->shuffle();

$shuffled->all();

// [3, 2, 5, 1, 4] // (随机生成)
```

slice()

`slice` 方法从给定索引开始返回集合的一个切片:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

$slice = $collection->slice(4);

$slice->all();

// [5, 6, 7, 8, 9, 10]
```

如果你想要限制返回切片的尺寸，将尺寸值作为第二个参数传递到该方法:

```
$slice = $collection->slice(4, 2);

$slice->all();

// [5, 6]
```

返回的切片有新的、数字化索引的键，如果你想要保持原有的键，可以使用 `values` 方法对它们进行重新索引。

sort()

`sort` 方法对集合进行排序，排序后的集合保持原来的数组键，在本例中我们使用 `values` 方法重置键为连续编号索引:

```
$collection = collect([5, 3, 1, 2, 4]);
```

```
$sorted = $collection->sort();
$sorted->values()->all();
// [1, 2, 3, 4, 5]
```

如果你需要更加高级的排序，你可以使用自己的算法传递一个回调给 `sort` 方法。参考 PHP 官方文档关于 `uasort` 的说明，`sort` 方法底层正是调用了该方法。

注：要为嵌套集合和对象排序，查看 `sortBy` 和 `sortByDesc` 方法。

sortBy()

`sortBy` 方法通过给定键对集合进行排序，排序后的集合保持原有数组索引，在本例中，使用 `values` 方法重置键为连续索引：

```
$collection = collect([
    ['name' => 'Desk', 'price' => 200],
    ['name' => 'Chair', 'price' => 100],
    ['name' => 'Bookcase', 'price' => 150],
]);
$sorted = $collection->sortBy('price');
$sorted->values()->all();
/*
[
    ['name' => 'Chair', 'price' => 100],
    ['name' => 'Bookcase', 'price' => 150],
    ['name' => 'Desk', 'price' => 200],
]
*/
```

你还可以传递自己的回调来判断如何排序集合的值：

```
$collection = collect([
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]);
$sorted = $collection->sortBy(function ($product, $key) {
    return count($product['colors']);
});
$sorted->values()->all();
/*
[
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]
*/
```

sortByDesc()

该方法和 `sortBy` 用法相同，不同之处在于按照相反顺序进行排序。

sortKeys()

`sortKeys` 方法通过底层关联数组的键对集合进行排序：

```
$collection = collect([
    'id' => 22345,
    'first' => 'John',
    'last' => 'Doe',
]);
$sorted = $collection->sortKeys();
$sorted->all();
/*
[
    'first' => 'John',
    'id' => 22345,
    'last' => 'Doe',
]
*/
```

sortKeysDesc()

该方法和 `sortKeys` 方法方法签名相同，但是排序顺序与其相反。

splice()

`splice` 方法从给定位置开始移除并返回数据项切片：

```
$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2);

$chunk->all();

// [3, 4, 5]

$collection->all();

// [1, 2]
```

你可以传递参数来限制返回组块的大小：

```
$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2, 1);

$chunk->all();

// [3]

$collection->all();

// [1, 2, 4, 5]
```

此外，你可以传递第三个包含新的数据项的参数来替代从集合中移除的数据项：

```
$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2, 1, [10, 11]);

$chunk->all();

// [3]

$collection->all();

// [1, 2, 10, 11, 4, 5]
```

split()

`split` 方法通过给定数值对集合进行分组：

```
$collection = collect([1, 2, 3, 4, 5]);

$groups = $collection->split(3);

$groups->toArray();

// [[1, 2], [3, 4], [5]]
```

sum()

`sum` 方法返回集合中所有数据项的和：

```
collect([1, 2, 3, 4, 5])->sum();

// 15
```

如果集合包含嵌套数组或对象，应该传递一个键用于判断对哪些值进行求和运算：

```
$collection = collect([
    ['name' => 'JavaScript: The Good Parts', 'pages' => 176],
    ['name' => 'JavaScript: The Definitive Guide', 'pages' => 1096],
]);

$collection->sum('pages');

// 1272
```

此外，你还可以传递自己的回调来判断对哪些值进行求和：

```
$collection = collect([
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]);
```

```
]);
$collection->sum(function ($product) {
    return count($product['colors']);
});
// 6
```

take()

`take` 方法使用指定数目的数据项返回一个新的集合:

```
$collection = collect([0, 1, 2, 3, 4, 5]);
$chunk = $collection->take(3);
$chunk->all();
// [0, 1, 2]
```

你还可以传递负数的方式从集合末尾开始获取指定数目的数据项:

```
$collection = collect([0, 1, 2, 3, 4, 5]);
$chunk = $collection->take(-2);
$chunk->all();
// [4, 5]
```

tap()

`tap` 方法会传递集合到给定回调，从而允许你在指定入口进入集合并对集合项进行处理而不影响集合本身:

```
collect([2, 4, 3, 1, 5])
    ->sort()
    ->tap(function ($collection) {
        Log::debug('Values after sorting', $collection->values()->toArray());
    })
    ->shift();

// 1
```

times()

通过静态 `times()` 方法可以通过调用指定次数的回调创建一个新的集合:

```
$collection = Collection::times(10, function ($number) {
    return $number * 9;
});

$collection->all();
// [9, 18, 27, 36, 45, 54, 63, 72, 81, 90]
```

该方法在和工厂方法一起创建 `Eloquent` 模型时很有用:

```
$categories = Collection::times(3, function ($number) {
    return factory(Category::class)->create(['name' => 'Category #' . $number]);
});

$categories->all();

/*
[
    ['id' => 1, 'name' => 'Category #1'],
    ['id' => 2, 'name' => 'Category #2'],
    ['id' => 3, 'name' => 'Category #3'],
]
*/
```

toArray()

`toArray` 方法将集合转化为一个原生的 PHP 数组。如果集合的值是 `Eloquent` 模型，该模型也会被转化为数组:

```
$collection = collect(['name' => 'Desk', 'price' => 200]);

$collection->toArray();

/*
[
    ['name' => 'Desk', 'price' => 200],
]
```

```
*/
```

注: `toArray` 还将所有嵌套对象转化为数组。如果你想要获取底层数组, 使用 `all` 方法。

toJson()

`toJson` 方法将集合转化为 JSON:

```
$collection = collect(['name' => 'Desk', 'price' => 200]);  
  
$collection->toJson();  
  
// '{"name": "Desk", "price": 200}'
```

transform()

`transform` 方法迭代集合并对集合中每个数据项调用给定回调。集合中的数据项将会被替换成从回调中返回的值:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$collection->transform(function ($item, $key) {  
    return $item * 2;  
});  
  
$collection->all();  
  
// [2, 4, 6, 8, 10]
```

注意: 不同于大多数其它集合方法, `transform` 修改集合本身, 如果你想要创建一个新的集合, 使用 `map` 方法。

union()

`union` 方法添加给定数组到集合, 如果给定数组包含已经在原来集合中存在的键, 原生集合的值会被保留:

```
$collection = collect([1 => ['a'], 2 => ['b']]);  
  
$union = $collection->union([3 => ['c'], 1 => ['b']]);  
  
$union->all();  
  
// [1 => ['a'], 2 => ['b'], 3 => ['c']]
```

unique()

`unique` 方法返回集合中所有的唯一数据项, 返回的集合保持原来的数组键, 在本例中我们使用 `values` 方法重置这些键为连续的数字索引:

```
$collection = collect([1, 1, 2, 2, 3, 4, 2]);  
  
$unique = $collection->unique();  
  
$unique->values()->all();  
  
// [1, 2, 3, 4]
```

处理嵌套数组或对象时, 可以指定用于判断唯一的键:

```
$collection = collect([  
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],  
    ['name' => 'iPhone 5', 'brand' => 'Apple', 'type' => 'phone'],  
    ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],  
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],  
    ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],  
]);  
  
$unique = $collection->unique('brand');  
  
$unique->values()->all();  
  
/*  
 * [  
 *     ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],  
 *     ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],  
 * ]  
 */
```

你还可以指定自己的回调用于判断数据项唯一性:

```
$unique = $collection->unique(function ($item) {  
    return $item['brand'].$item['type'];  
});  
  
$unique->values()->all();  
  
/*
```

```
[  
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],  
    ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],  
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],  
    ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],  
]  
*/
```

`unique` 方法在检查数据项值的时候使用「宽松」比较，也就是说一个整型字符串和整型数值被看作是相等的，如果要「严格」比较可以使 `uniqueStrict` 方法。

uniqueStrict()

该方法和 `unique` 方法签名一样，不同之处在于所有值都是「严格」比较。

unless()

`unless` 方法会执行给定回调，除非传递到该方法的第一个参数等于 `true`:

```
$collection = collect([1, 2, 3]);  
  
$collection->unless(true, function ($collection) {  
    return $collection->push(4);  
});  
  
$collection->unless(false, function ($collection) {  
    return $collection->push(5);  
});  
  
$collection->all();  
  
// [1, 2, 3, 5]
```

与 `unless` 相对的方法是 `when`。

unwrap()

静态 `unwrap` 方法会从给定值中返回集合项:

```
Collection::unwrap(collect('John Doe'));  
  
// ['John Doe']  
  
Collection::unwrap(['John Doe']);  
  
// ['John Doe']  
  
Collection::unwrap('John Doe');  
  
// 'John Doe'
```

values()

`values` 方法通过将集合键重置为连续整型数字的方式返回新的集合:

```
$collection = collect([  
    10 => ['product' => 'Desk', 'price' => 200],  
    11 => ['product' => 'Desk', 'price' => 200]  
]);  
  
$values = $collection->values();  
  
$values->all();  
  
/*  
 *  
 * 0 => ['product' => 'Desk', 'price' => 200],  
 * 1 => ['product' => 'Desk', 'price' => 200],  
 */
```

when()

`when` 方法在传入的第一个参数执行结果为 `true` 时执行给定回调:

```
$collection = collect([1, 2, 3]);  
  
$collection->when(true, function ($collection) {  
    return $collection->push(4);  
});  
  
$collection->when(false, function ($collection) {  
    return $collection->push(5);  
});
```

```
$collection->all();
```

```
// [1, 2, 3, 4]
```

与 `when` 方法相对的是 `unless`。

`where()`

`where` 方法通过给定键值对过滤集合：

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->where('price', 100);

$filtered->all();

/*
[
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Door', 'price' => 100],
]
*/
```

检查数据项值时 `where` 方法使用「宽松」比较，也就是说整型字符串和整型数组是等价的。使用 `whereStrict` 方法使用「严格」比较进行过滤。

`whereStrict()`

该方法和 `where` 用法签名一样，不同之处在于，所有值都使用「严格」比较。

`whereIn()`

`whereIn` 方法通过包含在给定数组中的键值对集合进行过滤：

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereIn('price', [150, 200]);

$filtered->all();

/*
[
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Desk', 'price' => 200],
]
*/
```

`whereIn` 方法在检查数据项值的时候使用「宽松」比较，要使用「严格」比较可以使用 `whereInStrict` 方法。

`whereInStrict()`

该方法和 `whereIn` 方法签名相同，不同之处在于 `whereInStrict` 在比较值的时候使用「严格」比较。

`whereInstanceOf()`

`whereInstanceOf` 方法通过给定类的类型过滤集合：

```
$collection = collect([
    new User,
    new User,
    new Post,
]);

return $collection->whereInstanceOf(User::class);
```

`whereNotIn()`

`whereNotIn` 方法通过给定键值过滤不在给定数组中的集合数据项：

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereNotIn('price', [150, 200]);

$filtered->all();
```

```
/*
[
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Door', 'price' => 100],
]
*/
```

`whereNotIn` 方法在检查集合项值的时候使用「宽松」比较，也就是说整型字符串和整型数值被看作是相等的。要想进行严格过滤可以使用 `whereNotInStrict` 方法。

`whereNotInStrict()`

该方法和 `whereNotIn` 方法签名一样，不同之处在于所有值都使用「严格」比较。

`wrap()`

静态 `wrap` 方法会将给定值封装到集合中：

```
$collection = Collection::wrap('John Doe');

$collection->all();

// ['John Doe']

$collection = Collection::wrap(['John Doe']);

$collection->all();

// ['John Doe']

$collection = Collection::wrap(collect('John Doe'));

$collection->all();

// ['John Doe']
```

`zip()`

`zip` 方法在与集合的值对应的索引处合并给定数组的值：

```
$collection = collect(['Chair', 'Desk']);

$zipped = $collection->zip([100, 200]);

$zipped->all();

// [['Chair', 100], ['Desk', 200]]
```

高阶消息传递

集合还支持“高阶消息传递”，也就是在集合上执行通用的功能，支持高阶消息传递的方法包括：`average`、`avg`、`contains`、`each`、`every`、`filter`、`first`、`map`、`partition`、`reject`、`sortBy`、`sortByDesc`、`sum` 和 `unique`。

每个高阶消息传递都可以在集合实例上以动态属性的方式访问，例如，我们使用 `each` 高阶消息传递来在集合的每个对象上调用一个方法：

```
$users = User::where('votes', '>', 500)->get();

$users->each->markAsVip();
```

类似的，我们可以使用 `sum` 高阶消息传递来聚合用户集合的投票总数：

```
$users = User::where('group', 'Development')->get();

return $users->sum->votes;
```

事件

简介

Laravel 事件提供了简单的观察者模式实现，允许你订阅和监听应用中的事件。事件类通常存放在 `app/Events` 目录，监听器存放 在 `app/Listeners`。如果你在应用中没有看到这些目录，不要担心，它们会在你使用 Artisan 命令生成事件和监听器的时候自动创建。事件为应用功能模块解耦提供了行之有效的解决办法，因为单个事件可以有多个监听器而这些监听器之间并不相互依赖。例如，你可能想要在每次订单发送时给用户发送一个 Slack 通知，有了事件之后，你大可不必将订单处理代码和 Slack 通知代码耦合在一起，而只需要简单触发一个可以被监听器接收并处理为 Slack 通知的 `OrderShipped` 事件即可。

注册事件/监听器

Laravel 自带的 `EventServiceProvider` 为事件监听器注册提供了方便之所。其中的 `listen` 属性包含了事件（键）和对应监听器（值）数组。如果应用需要，你可以添加多个事件到该数组。下面让我们添加一个 `OrderShipped` 事件：

```
/**
 * 应用的事件监听器映射.
 *
 * @var array
 * @translator laravelacademy.org
 */
protected $listen = [
    'App\Events\OrderShipped' => [
        'App\Listeners\SendShipmentNotification',
    ],
];
```

生成事件/监听器类

当然，手动为每个事件和监听器创建文件是很笨重的，取而代之地，我们只需简单添加监听器和事件到 `EventServiceProvider` 然后运行 `event:generate` 命令。该命令将会生成罗列在 `EventServiceProvider` 中的所有事件和监听器。当然，已存在的事件和监听器不会被重复创建：

```
php artisan event:generate
```

手动注册事件

通常，我们需要通过 `EventServiceProvider` 的 `$listen` 数组注册事件，此外，你还可以在 `EventServiceProvider` 的 `boot` 方法中手动注册基于闭包的事件：

```
/**
 * 注册应用的其它事件.
 *
 * @return void
 */
public function boot()
{
    parent::boot();

    Event::listen('event.name', function ($foo, $bar) {
        //
    });
}
```

通配符事件监听器

你甚至还可以使用通配符^{*}来注册监听器，这样就可以通过同一个监听器捕获多个事件。通配符监听器接收整个事件数据数组作为参数：

```
$events->listen('event.*', function ($eventName, array $data) {
    //
});
```

定义事件

事件类是一个处理与事件相关的简单数据容器，例如，假设我们生成的 `OrderShipped` 事件接收一个 `Eloquent ORM` 对象：

```
<?php

namespace App\Events;

use App\Order;
use Illuminate\Queue\SerializesModels;

class OrderShipped
{
    use SerializesModels;

    public $order;

    /**
     * 创建一个新的事件实例.
     *
     * @param Order $order
     * @return void
     */
}
```

```
public function __construct(Order $order)
{
    $this->order = $order;
}
}
```

正如你所看到的，该事件类不包含任何特定逻辑，只是一个存放被购买的 `Order` 对象的容器，如果事件对象被序列化的话，事件使用的 `SerializesModels` trait 将会使用 PHP 的 `serialize` 函数序列化所有 Eloquent 模型。

定义监听器

接下来，让我们看看示例事件的监听器，事件监听器在 `handle` 方法中接收事件实例，`event:generate` 命令将会自动在 `handle` 方法中导入相应的事件类和类型提示事件。在 `handle` 方法内，你可以执行任何需要的逻辑以响应事件：

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;

class SendShipmentNotification
{
    /**
     * 创建事件监听器.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * 处理事件.
     *
     * @param OrderShipped $event
     * @return void
     */
    public function handle(OrderShipped $event)
    {
        // 使用 $event->order 发访问订单...
    }
}
```

注：事件监听器还可以在构造器中类型提示任何需要的依赖，所有事件监听器通过 `服务容器` 解析，所以依赖会自动注入。

停止事件继续往下传播

有时候，你希望停止事件被传播到其它监听器，你可以通过从监听器的 `handle` 方法中返回 `false` 来实现。

事件监听器队列

如果监听器将要执行耗时任务比如发送邮件或者发送 HTTP 请求，那么将监听器放到队列是一个不错的选择。在队列化监听器之前，确保已经配置好队列并且在服务器或本地环境启动一个队列监听器。

要指定某个监听器需要放到队列，只需要让监听器类实现 `ShouldQueue` 接口即可，通过 Artisan 命令 `event:generate` 生成的监听器类已经将这个接口导入当前命名空间，所有你可以直接拿来使用：

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    //
}
```

就是这么简单！当这个监听器被调用的时候，将会使用 Laravel 的 `队列系统` 通过事件分发器自动推送到队列。如果通过队列执行监听器的时候没有抛出任何异常，队列任务会在执行完成后被自动删除。

自定义队列连接 & 队列名称

如果你想要自定义事件监听器使用的队列连接和队列名称，可以在监听器类中定义 `$connection` 和 `$queue` 属性：

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    /**
     * 任务将被推送到的连接名称.
     *
     * @var string|null
     */
    public $connection = 'sqe';

    /**
     * 任务将被推送到的连接名称.
     *
     * @var string|null
     */
    public $queue = 'listeners';
}
```

手动访问队列

如果你需要手动访问底层队列任务的 `delete` 和 `release` 方法，在生成的监听器中，默认导入的 `Illuminate\Queue\InteractsWithQueue` trait 为这两个方法提供了访问权限：

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    public function handle(OrderShipped $event)
    {
        if (true) {
            $this->release(30);
        }
    }
}
```

处理失败任务

有时候队列中的事件监听器可能会执行失败。如果队列中的监听器任务执行时超出了队列进程定义的最大尝试次数，监听器上的 `failed` 方法会被调用，`failed` 方法接收事件实例和导致失败的异常：

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    public function handle(OrderShipped $event)
    {
        //
    }
}
```

```
public function failed(OrderShipped $event, $exception)
{
    //
}
}
```

分发事件

要分发一个事件，可以传递事件实例到辅助函数 `event`，这个辅助函数会分发事件到所有注册的监听器。由于辅助函数 `event` 全局有效，所以可以在应用的任何地方调用它：

```
<?php

namespace App\Http\Controllers;

use App\Order;
use App\Events\OrderShipped;
use App\Http\Controllers\Controller;

class OrderController extends Controller
{
    /**
     * 处理给定订单.
     *
     * @param int $orderId
     * @return Response
     */
    public function ship($orderId)
    {
        $order = Order::findOrFail($orderId);

        // 订单处理逻辑...

        event(new OrderShipped($order));
    }
}
```

注：测试的时候，只需要断言特定事件被分发，无需真正触发监听器，Laravel [自带的测试函数](#)让这一实现轻而易举。

事件订阅者

编写事件订阅者

事件订阅者是指那些在类本身中订阅多个事件的类，通过事件订阅者你可以在单个类中定义多个事件处理器。订阅者需要定义一个 `subscribe` 方法，该方法中传入一个事件分发器实例。你可以在给定的分发器中调用 `listen` 方法注册事件监听器：

```
<?php

namespace App\Listeners;

class UserEventSubscriber
{
    /**
     * 处理用户登录事件.
     *
     * @translator laravelacademy.org
     */
    public function onUserLogin($event) {}

    /**
     * 处理用户退出事件.
     */
    public function onUserLogout($event) {}

    /**
     * 为订阅者注册监听器.
     *
     * @param Illuminate\Events\Dispatcher $events
     */
}
```

```

public function subscribe($events)
{
    $events->listen(
        'Illuminate\Auth\Events\Login',
        'App\Listeners\UserEventSubscriber@onUserLogin'
    );

    $events->listen(
        'Illuminate\Auth\Events\Logout',
        'App\Listeners\UserEventSubscriber@onUserLogout'
    );
}

```

注册事件订阅者

编写好订阅者之后，就可以通过事件分发器对订阅者进行注册，你可以使用 `EventServiceProvider` 提供的 `$subscribe` 属性来注册订阅者。例如，让我们添加一个 `UserEventSubscriber`：

```

<?php

namespace App\Providers;

use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvider;

class EventServiceProvider extends ServiceProvider
{
    /**
     * 应用的事件监听器映射.
     *
     * @var array
     */
    protected $listen = [
        // ...
    ];

    /**
     * 要注册的订阅者类.
     *
     * @var array
     */
    protected $subscribe = [
        'App\Listeners\UserEventSubscriber',
    ];
}

```

文件存储

简介

Laravel 基于 Frank de Jonge 开发的 PHP 包 `Flysystem` 提供了强大的文件系统抽象层。Laravel 集成 Flysystem 以便使用不同驱动简化对文件系统的操作，这些驱动包括本地文件系统、Amazon S3 以及 Rackspace 云存储。此外，在这些存储选项之间切换非常简单，因为对不同系统而言，API 是一致的。

配置

文件系统配置文件位于 `config/filesystems.php`。在该文件中可以配置所有“磁盘”，每个磁盘描述了特定的存储驱动和存储路径。该配置文件为每种支持的驱动提供了示例配置，所以，简单编辑该配置来应用你的存储参数和认证信息即可：

```
'disks' => [
    'local' => [
        'driver' => 'local',
        'root' => storage_path('path: app'),
    ],
    'public' => [
        'driver' => 'local',
        'root' => storage_path('path: app/public'),
        'url' => env('key: APP_URL').'/storage',
        'visibility' => 'public',
    ],
    's3' => [
        'driver' => 's3',
        'key' => env('key: AWS_KEY'),
        'secret' => env('key: AWS_SECRET'),
        'region' => env('key: AWS_REGION'),
        'bucket' => env('key: AWS_BUCKET'),
    ],
],
```

当然，你想配置多少磁盘就配置多少，多个磁盘也可以共用同一个驱动。

公共磁盘

`public` 磁盘用于存储可以被公开访问的文件，默认情况下，`public` 磁盘使用 `local` 驱动并将文件存储在 `storage/app/public` 目录下，要让这些文件可以通过 Web 浏览器访问到，还需要创建一个软链 `public/storage` 指向 `storage/app/public`，这种方式可以将公开访问的文件保存在一个可以很容易被不同部署环境共享的目录，在使用零停机时间部署系统如 `Envoyer` 的时候尤其方便。

要创建这个软链，可以使用 Artisan 命令 `storage:link`：

```
php artisan storage:link
```

文件被存储并且软链已经被创建的情况下，就可以使用辅助函数 `asset` 创建一个指向该文件的 URL：

```
echo asset('storage/file.txt');
```

本地驱动

使用 `local` 驱动的时候，所有的文件操作都相对于定义在配置文件中的 `root` 目录，默认情况下，该值设置为 `storage/app` 目录，因此，下面的方法将会存储文件到 `storage/app/file.txt`：

```
Storage::disk('local')->put('file.txt', 'Contents');
```

驱动预备知识

Composer 包

在使用 SFTP、Amazon S3 或 Rackspace 驱动之前，需要通过 Composer 安装相应的包：

- SFTP: `league/flysystem-sftp ~1.0`
- Amazon S3: `league/flysystem-aws-s3-v3 ~1.0`
- Rackspace: `league/flysystem-rackspace ~1.0`

S3 驱动配置

S3 驱动配置信息位于配置文件 `config/filesystems.php`，该文件包含 S3 驱动的示例配置数组。你可以使用自己的 S3 配置和认证信息编辑该数组。为了方便起见，这些环境变量和 AWS CLI 的命名规范一致。

FTP 驱动配置

Laravel 的 Flysystem 集成支持 FTP 操作，不过，框架默认的配置文件 `filesystems.php` 并没有提供示例配置。如果你需要配置一个 FTP 文件系统，可以使用以下示例配置：

```
'ftp' => [
    'driver' => 'ftp',
    'host' => 'ftp.example.com',
    'username' => 'your-username',
    'password' => 'your-password',

    // Optional FTP Settings...
    // 'port' => 21,
    // 'root' => '',
    // 'passive' => true,
```

```
// 'ssl'      => true,
// 'timeout'  => 30,
],
```

SFTP 驱动配置

Laravel 的 Flysystem 集成支持 SFTP 操作，不过，框架默认的 `filesystems.php` 配置文件并没有为此提供一个示例配置。如果你需要配置针对 SFTP 的文件系统配置，可以使用下面的示例配置：

```
'sftp' => [
    'driver' => 'sftp',
    'host' => 'example.com',
    'username' => 'your-username',
    'password' => 'your-password',

    // Settings for SSH key based authentication...
    // 'privateKey' => '/path/to/privateKey',
    // 'password' => 'encryption-password',

    // Optional SFTP Settings...
    // 'port' => 22,
    // 'root' => '',
    // 'timeout' => 30,
],
```

Rackspace 驱动配置

Laravel 的 Flysystem 集成还支持 Rackspace 操作，同样，默认配置文件 `filesystems.php` 也没有提供对应的示例配置，如果你需要配置 Rackspace 文件系统，可以使用以下示例配置：

```
'rackspace' => [
    'driver'      => 'rackspace',
    'username'   => 'your-username',
    'key'        => 'your-key',
    'container'  => 'your-container',
    'endpoint'   => 'https://identity.api.rackspacecloud.com/v2.0/',
    'region'     => 'IAD',
    'url_type'   => 'publicURL',
],
```

获取硬盘实例

我们可以使用 `Storage` 门面和上面配置的任意磁盘进行交互，例如，可以使用该门面上的 `put` 方法来存储头像到默认磁盘，如果调用 `Storage` 门面上的方法而没有调用 `disk` 方法，则调用的方法会自动被传递到默认磁盘：

```
use Illuminate\Support\Facades\Storage;
Storage::put('avatars/1', $fileContents);
```

与多个磁盘进行交互时，可以使用 `Storage` 门面上的 `disk` 方法访问特定磁盘：

```
Storage::disk('s3')->put('avatars/1', $fileContents);
```

获取文件

`get` 方法用于获取给定文件的内容，该方法将会返回该文件的原生字符串内容。需要注意的是，所有文件路径都是相对于配置文件中指定的磁盘默认根目录：

```
$contents = Storage::get('file.jpg');
```

`exists` 方法用于判断给定文件是否存在与磁盘上：

```
$exists = Storage::disk('s3')->exists('file.jpg');
```

下载文件

`download` 方法可用于生成强制用户浏览器下载给定路径文件的响应。`download` 方法接收一个文件名作为第二个参数用于决定用户下载时看到的文件名。最后，你可以传递一个 HTTP 请求头数组作为该方法的第三个参数：

```
return Storage::download('file.jpg');

return Storage::download('file.jpg', $name, $headers);
```

文件 URL

使用 `local` 或 `s3` 驱动时，可以使用 `url` 方法获取给定文件的 URI。如果你使用的是 `local` 驱动，通常会在给定路径前加上 `/storage`，并返回该文件的相对 URL；如果使用的是 `s3` 或 `rackspace` 驱动，则会返回完整的远程 URL：

```
use Illuminate\Support\Facades\Storage;
```

```
$url = Storage::url('file1.jpg');
```

注：记住，如果你在使用 `local` 驱动，所有需要公开访问的文件都应该存放在 `storage/app/public` 目录下，此外，你还需要创建一个指向 `storage/app/public` 目录的软链接 `public/storage`。

临时 URL

对于使用 `s3` 或 `rackspace` 驱动存储文件的系统，可以使用 `temporaryUrl` 方法创建临时 URL 到给定文件，该方法接收一个路径参数和指定 URL 何时过期的 `DateTime` 实例：

```
$url = Storage::temporaryUrl(
    'file1.jpg', now()->addMinutes(5)
);
```

自定义本地主机 URL

如果你想要预定义使用 `local` 驱动磁盘存放文件的主机，可以添加 `url` 选项到磁盘配置数组：

```
'public' => [
    'driver' => 'local',
    'root' => storage_path('app/public'),
    'url' => env('APP_URL') . '/storage',
    'visibility' => 'public',
],
```

文件元信息

除了读写文件之外，Laravel 还可以提供文件本身的信息。例如，`size` 方法可用于以字节方式返回文件大小：

```
use Illuminate\Support\Facades\Storage;
$size = Storage::size('file1.jpg');
```

`lastModified` 方法以 UNIX 时间戳

格式返回文件最后一次修改时间：

```
$time = Storage::lastModified('file1.jpg');
```

存储文件

`put` 方法可用于存储原生文件内容到磁盘。此外，还可以传递一个 PHP 资源到 `put` 方法，该方法将会使用 Flysystem 底层的流支持。在处理大文件的时候推荐使用文件流：

```
use Illuminate\Support\Facades\Storage;

Storage::put('file.jpg', $contents);
Storage::put('file.jpg', $resource);
```

自动文件流

如果你想要 Laravel 自动将给定文件流输出到对应存储路径，可以使用 `putFile` 或 `putFileAs` 方法，该方法接收 `Illuminate\Http\File` 或 `Illuminate\Http\UploadedFile` 实例，然后自动将文件流保存到期望的路径：

```
use Illuminate\Http\File;
use Illuminate\Support\Facades\Storage;

// 自动计算文件名的 MD5 值...
Storage::putFile('photos', new File('/path/to/photo'));

// 手动指定文件名...
Storage::putFileAs('photos', new File('/path/to/photo'), 'photo.jpg');
```

这里有一些关于 `putFile` 方法的重要注意点，注意到我们只指定了目录名，默认情况下，`putFile` 方法会基于文件内容自动生成文件名。实现原理是对文件内容进行 MD5 哈希运算。`putFile` 方法会返回文件路径，包括文件名，以便于在数据库中进行存储。

`putFile` 和 `putFileAs` 方法还接收一个用于指定存储文件“可见度”的参数，这在你将文件存储到云存储（如 S3）平台并期望文件可以被公开访问时很有用：

```
Storage::putFile('photos', new File('/path/to/photo'), 'public');
```

添加内容到文件开头/结尾

`prepend` 和 `append` 方法允许你轻松插入内容到文件开头/结尾：

```
Storage::prepend('file.log', 'Prepended Text');
Storage::append('file.log', 'Appended Text');
```

拷贝 & 移动文件

`copy` 方法将磁盘中已存在的文件从一个地方拷贝到另一个地方，而 `move` 方法将磁盘中已存在的文件从一定地方移到到另一个地方：

```
Storage::copy('old/file1.jpg', 'new/file1.jpg');
Storage::move('old/file1.jpg', 'new/file1.jpg');
```

文件上传

在 Web 应用中，最常见的存储文件案例就是存储用户上传的文件，如用户头像、照片和文档等。Laravel 通过上传文件实例上提供的 `store` 方法让存储上传文件变得简单。你只需要传入上传文件保存的路径并调用 `store` 方法即可：

```
<?php

namespace App\Http\Controllers;

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

class UserAvatarController extends Controller
{
    /**
     * 更新用户头像.
     *
     * @param Request $request
     * @return Response
     * @translator laravelacademy.org
     */
    public function update(Request $request)
    {
        $path = $request->file('avatar')->store('avatars');

        return $path;
    }
}
```

这里有一些需要注意的重要事项，在这里我们只指定了目录名，而不是文件名。默认情况下，`store` 方法会基于文件内容自动生成文件名，这通过对文件内容进行 MD5 实现。`store` 方法会返回文件路径以便在数据库中保存文件路径和文件名。

你还可以调用 `Storage` 门面上的 `putFile` 方法来执行与上例同样的文件操作：

```
$path = Storage::putFile('avatars', $request->file('avatar'));
```

指定文件名

如果你不想要自动生成文件名，可以使用 `storeAs` 方法，该方法接收路径、文件名以及磁盘（可选）作为参数：

```
$path = $request->file('avatar')->storeAs(
    'avatars', $request->user()->id
);
```

当然，你还可以使用 `Storage` 门面上的 `putFileAs` 方法，该方法与上面的方法实现同样的操作：

```
$path = Storage::putFileAs(
    'avatars', $request->file('avatar'), $request->user()->id
);
```

指定磁盘

默认情况下，`store` 方法会使用默认的磁盘，如果你想要指定其它磁盘，传递磁盘名称作为 `store` 方法的第二个参数即可：

```
$path = $request->file('avatar')->store(
    'avatars/'.$request->user()->id, 's3'
);
```

文件可见度

在 Laravel 的 Flysystem 集成中，“可见度”是对不同平台上文件权限的抽象，文件可以被声明成 `public` 或 `private`，当文件被声明为 `public`，意味着文件可以被其他人访问。例如，使用 S3 时，可以获取 `public` 文件的 URL。

使用 `put` 方法设置文件的时候可以顺便设置可见度：

```
use Illuminate\Support\Facades\Storage;
Storage::put('file.jpg', $contents, 'public');
```

如果文件已经被存储，可见度可以通过 `getVisibility` 和 `setVisibility` 方法获取和设置：

```
$visibility = Storage::getVisibility('file.jpg');
Storage::setVisibility('file.jpg', 'public');
```

删除文件

`delete` 方法接收单个文件名或多个文件数组并将其从磁盘移除：

```
use Illuminate\Support\Facades\Storage;

Storage::delete('file.jpg');
Storage::delete(['file1.jpg', 'file2.jpg']);
```

如果需要的话你可以指定从哪个磁盘删除文件:

```
use Illuminate\Support\Facades\Storage;

Storage::disk('s3')->delete('folder_path/file_name.jpg');
```

目录

获取一个目录下的所有文件

`files` 方法返回给定目录下的所有文件数组，如果你想要获取给定目录下包含子目录的所有文件列表，可以使用 `allFiles` 方法:

```
use Illuminate\Support\Facades\Storage;

$files = Storage::files($directory);
$files = Storage::allFiles($directory);
```

获取一个目录下的所有子目录

`directories` 方法返回给定目录下所有目录数组，此外，可以使用 `allDirectories` 方法获取嵌套的所有子目录数组:

```
$directories = Storage::directories($directory);

// 递归...

$directories = Storage::allDirectories($directory);
```

创建目录

`makeDirectory` 方法将会创建给定目录，包含子目录（递归）：

```
Storage::makeDirectory($directory);
```

删除目录

最后，`deleteDirectory` 方法用于移除目录，包括该目录下的所有文件:

```
Storage::deleteDirectory($directory);
```

自定义文件系统

Laravel 的 Flysystem 集成开箱提供了多个“驱动”，不过，Flysystem 的功能并不止步于此，还为许多其他存储系统提供了适配器。通过使用这些额外的适配器你可以在自己的 Laravel 应用创建自定义的驱动。

为了设置自定义的文件系统，你需要一个 Flysystem 适配器，我们来添加一个社区维护的 Dropbox 适配器到项目中:

```
composer require spatie/flysystem-dropbox
```

接下来，需要创建一个服务提供者如 `DropboxServiceProvider`。在该提供者的 `boot` 方法中，你可以使用 `Storage` 门面的 `extend` 方法定义自定义驱动:

```
<?php

namespace App\Providers;

use Storage;
use League\Flysystem\Filesystem;
use Illuminate\Support\ServiceProvider;
use Spatie\Dropbox\Client as DropboxClient;
use Spatie\FlysystemDropbox\DropboxAdapter;

class DropboxServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Storage::extend('dropbox', function ($app, $config) {
            $client = new DropboxClient(
                $config['authorizationToken']
            );

            return new Filesystem(new DropboxAdapter($client));
        });
    }

    /**
     * Register bindings in the container.
     *
     */
}
```

```
* @return void
*/
public function register()
{
    //
}
}
```

`extend` 方法的第一个参数是驱动名称，第二个参数是获取 `$app` 和 `$config` 变量的闭包。该解析器闭包必须返回一个 `League\Flysystem\Filesystem` 实例。`$config` 变量包含了定义在配置文件 `config/filesystems.php` 中为特定磁盘定义的选项。创建好注册扩展的服务提供者后，就可以使用配置文件 `config/filesystem.php` 中的 `dropbox` 驱动了。

辅助函数

简介

Laravel 自带了一系列 PHP 辅助函数，很多被框架自身使用，如果你觉得方便的话也可以在代码中使用它们。

函数列表

数组 & 对象函数

`array_add()`

`array_add` 函数添加给定键值对到数组 —— 如果给定键不存在的话：

```
$array = array_add(['name' => 'Desk'], 'price', 100);
// ['name' => 'Desk', 'price' => 100]
```

`array_collapse()`

`array_collapse` 函数将多个数组合并成一个：

```
$array = array_collapse([[1, 2, 3], [4, 5, 6], [7, 8, 9]]);
// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`array_divide()`

`array_divide` 函数返回两个数组，一个包含原数组的所有键，另外一个包含原数组的所有值：

```
list($keys, $values) = array_divide(['name' => 'Desk']);
// $keys: ['name']
// $values: ['Desk']
```

`array_dot()`

`array_dot` 函数使用`."`号将多维数组转化为一维数组：

```
$array = ['products' => ['desk' => ['price' => 100]]];
$flattened = array_dot($array);
// ['products.desk.price' => 100]
```

`array_except()`

`array_except` 函数从数组中移除给定键值对：

```
$array = ['name' => 'Desk', 'price' => 100];
$array = array_except($array, ['price']);
// ['name' => 'Desk']
```

`array_first()`

`array_first` 函数返回通过测试数组的第一个元素：

```
$array = [100, 200, 300];

$value = array_first($array, function ($value, $key) {
    return $value >= 150;
});

// 200
```

默认值可以作为第三个参数传递给该方法，如果没有值通过测试的话返回默认值：

```
$value = array_first($array, $callback, $default);
```

`array_flatten()`

`array_flatten` 函数将多维数组转化为一维数组：

```
$array = ['name' => 'Joe', 'languages' => ['PHP', 'Ruby']];
$array = array_flatten($array);
```

```
// ['Joe', 'PHP', 'Ruby'];
```

array_forget()

`array_forget` 函数使用“.”号从嵌套数组中移除给定键值对:

```
$array = ['products' => ['desk' => ['price' => 100]]];  
  
array_forget($array, 'products.desk');  
  
// ['products' => []]
```

array_get()

`array_get` 方法使用“.”号从嵌套数组中获取值:

```
$array = ['products' => ['desk' => ['price' => 100]]];  
  
$value = array_get($array, 'products.desk.price');  
  
// ['price' => 100]
```

`array_get` 函数还接收一个默认值, 如果指定键不存在的话则返回该默认值:

```
$value = array_get($array, 'products.desk.discount', 0);  
  
// 0
```

array_has()

`array_has` 函数使用“.”检查给定数据项是否在数组中存在:

```
$array = ['product' => ['name' => 'desk', 'price' => 100]];  
  
$hasItem = array_has($array, 'product.name');  
  
// true  
  
$hasItems = array_has($array, ['product.price', 'product.discount']);  
  
// false
```

array_last()

`array_last` 函数返回通过过滤数组的最后一个元素:

```
$array = [100, 200, 300, 110];  
  
$value = array_last($array, function ($value, $key) {  
    return $value >= 150;  
});  
  
// 300
```

我们可以传递一个默认值作为第三个参数到该函数, 如果没有值通过真理测试的话该默认值被返回:

```
$last = array_last($array, $callback, $default);
```

array_only()

`array_only` 方法只从给定数组中返回指定键值对:

```
$array = ['name' => 'Desk', 'price' => 100, 'orders' => 10];  
  
$array = array_only($array, ['name', 'price']);  
  
// ['name' => 'Desk', 'price' => 100]
```

array_pluck()

`array_pluck` 方法从数组中返回给定键对应的键值对列表:

```
$array = [  
    ['developer' => ['id' => 1, 'name' => 'Taylor']],  
    ['developer' => ['id' => 2, 'name' => 'Abigail']],  
];  
  
$names = array_pluck($array, 'developer.name');  
  
// ['Taylor', 'Abigail']
```

你还可以指定返回结果的键:

```
$array = array_pluck($array, 'developer.name', 'developer.id');  
  
// [1 => 'Taylor', 2 => 'Abigail'];
```

arrayprepend()

`arrayprepend` 函数将数据项推入数组开头:

```
$array = ['one', 'two', 'three', 'four'];

$array = arrayprepend($array, 'zero');

// $array: ['zero', 'one', 'two', 'three', 'four']
```

如果需要的话还可以指定用于该值的键:

```
$array = ['price' => 100];

$array = arrayprepend($array, 'Desk', 'name');

// ['name' => 'Desk', 'price' => 100]
```

arraypull()

`arraypull` 函数从数组中返回并移除键值对:

```
$array = ['name' => 'Desk', 'price' => 100];

$name = arraypull($array, 'name');

// $name: Desk

// $array: ['price' => 100]
```

我们还可以传递默认值作为第三个参数到该函数，如果指定键不存在的话返回该值:

```
$value = arraypull($array, $key, $default);
```

arrayrandom()

`arrayrandom` 函数从数组中返回随机值:

```
$array = [1, 2, 3, 4, 5];

$random = arrayrandom($array);

// 4 - (retrieved randomly)
```

还可以指定返回的数据项数目作为可选的第二个参数，需要注意的是提供这个参数会返回一个数组，即使只返回一个数据项:

```
$items = arrayrandom($array, 2);

// [2, 5] - (retrieved randomly)
```

arrayset()

`arrayset` 函数用于在嵌套数组中使用“.”号设置值:

```
$array = ['products' => ['desk' => ['price' => 100]]];

arrayset($array, 'products.desk.price', 200);

// ['products' => ['desk' => ['price' => 200]]]
```

arraysort()

`arraysort` 函数通过值对数组进行排序:

```
$array = ['Desk', 'Table', 'Chair'];

$sorted = arraysort($array);

// ['Chair', 'Desk', 'Table']
```

还可以通过给定闭包的结果对数组进行排序:

```
$array = [
    ['name' => 'Desk'],
    ['name' => 'Table'],
    ['name' => 'Chair'],
];

$sorted = arrayvalues(arraysort($array, function ($value) {
    return $value['name'];
}));
```

/*

```
[ 'name' => 'Chair'],
```

```

        ['name' => 'Desk'],
        ['name' => 'Table'],
    ]
*/

```

array_sort_recursive()

`array_sort_recursive` 函数使用 `sort` 函数对数组进行递归排序:

```

$array = [
    ['Roman', 'Taylor', 'Li'],
    ['PHP', 'Ruby', 'JavaScript'],
];

$array = array_sort_recursive($array);

/*
[
    ['Li', 'Roman', 'Taylor'],
    ['JavaScript', 'PHP', 'Ruby'],
]
*/

```

array_where()

`array_where` 函数使用给定闭包对数组进行过滤:

```

$array = [100, '200', 300, '400', 500];

$array = array_where($array, function ($key, $value) {
    return is_string($value);
});

// [1 => 200, 3 => 400]

```

array_wrap()

`array_wrap` 函数将给定值包裹到数组中，如果给定值已经是数组则保持不变:

```

$string = 'Laravel';

$array = array_wrap($string);

// ['Laravel']

```

如果给定值是空的，则返回一个空数组:

```

$nothing = null;

$array = array_wrap($nothing);

// []

```

data_fill()

`data_fill` 函数使用「.」号以嵌套数组或对象的方式设置缺失值:

```

$data = ['products' => ['desk' => ['price' => 100]]];

data_fill($data, 'products.desk.price', 200);

// ['products' => ['desk' => ['price' => 100]]]

data_fill($data, 'products.desk.discount', 10);

// ['products' => ['desk' => ['price' => 100, 'discount' => 10]]]

```

该函数还接收「*」号作为通配符并填充相应目标:

```

$data = [
    'products' => [
        ['name' => 'Desk 1', 'price' => 100],
        ['name' => 'Desk 2'],
    ],
];

data_fill($data, 'products.*.price', 200);

/*
[
    'products' => [
        ['name' => 'Desk 1', 'price' => 100],

```

```

        ['name' => 'Desk 2', 'price' => 200],
    ],
]
*/

```

data_get()

`data_get` 函数使用「.」号从嵌套数组或对象中获取值：

```

$data = ['products' => ['desk' => ['price' => 100]]];

$price = data_get($data, 'products.desk.price');

// 100

```

`data_get` 函数还接收默认值，以便指定键不存在的情况下返回：

```

$discount = data_get($data, 'products.desk.discount', 0);

// 0

```

data_set()

`data_set` 函数使用「.」号设置嵌套数组或对象的值：

```

$data = ['products' => ['desk' => ['price' => 100]]];

data_set($data, 'products.desk.price', 200);

// ['products' => ['desk' => ['price' => 200]]]

```

该函数还接收通配符然后设置相应的目标值：

```

$data = [
    'products' => [
        ['name' => 'Desk 1', 'price' => 100],
        ['name' => 'Desk 2', 'price' => 150],
    ],
];

data_set($data, 'products.*.price', 200);

/*
[
    'products' => [
        ['name' => 'Desk 1', 'price' => 200],
        ['name' => 'Desk 2', 'price' => 200],
    ],
]
*/

```

默认情况下，任意已存在的值都会被覆盖，如果你想要只设置不存在的值，可以传递 `false` 作为第三个参数：

```

$data = ['products' => ['desk' => ['price' => 100]]];

data_set($data, 'products.desk.price', 200, false);

// ['products' => ['desk' => ['price' => 100]]]

```

head()

`head` 函数只是简单返回给定数组的第一个元素：

```

$array = [100, 200, 300];

$first = head($array);

// 100

```

last()

`last` 函数返回给定数组的最后一个元素：

```

$array = [100, 200, 300];

$last = last($array);

// 300

```

路径函数

app_path()

`app_path` 函数返回 `app` 目录的绝对路径，你还可以使用 `app_path` 函数为相对于 `app` 目录的给定文件生成绝对路径：

```
$path = app_path();
$path = app_path('Http\Controllers\Controller.php');
```

base_path()

`base_path` 函数返回项目根目录的绝对路径，你还可以使用 `base_path` 函数为相对于应用根目录的给定文件生成绝对路径：

```
$path = base_path();
$path = base_path('vendor/bin');
```

config_path()

`config_path` 函数返回应用配置目录 `config` 的绝对路径，还可以使用 `config_path` 函数在应用配置目录内为给定文件生成完整路径：

```
$path = config_path();
$path = config_path('app.php');
```

database_path()

`database_path` 函数返回应用数据库目录 `database` 的完整路径，还可以使用 `database_path` 函数在数据库目录内为给定文件生成完整路径：

```
$path = database_path();
$path = database_path('factories/UserFactory.php');
```

mix()

`mix` 函数返回带有版本号的 Mix 文件路径：

```
mix($file);
```

public_path()

`public_path` 函数返回 `public` 目录的绝对路径，还可以使用 `public_path` 函数在 `public` 目录下为给定文件生成完整路径：

```
$path = public_path();
$path = public_path('css/app.css');
```

resource_path()

`resource_path` 函数返回 `resources` 目录的绝对路径，还可以使用 `resources` 函数在 `resources` 目录下为给定文件生成完整路径：

```
$path = resource_path();
$path = resource_path('assets/sass/app.scss');
```

storage_path()

`storage_path` 函数返回 `storage` 目录的绝对路径，还可以使用 `storage_path` 函数在 `storage` 目录下为给定文件生成完整路径：

```
$path = storage_path();
$path = storage_path('app/file.txt');
```

字符串函数

-0

`_` 函数会使用 [本地化文件](#) 翻译给定翻译字符串或翻译键：

```
echo __('Welcome to our application');
echo __('messages.welcome');
```

如果给定翻译字符串或键不存在，`_` 函数将会返回给定值。所以，使用上面的例子，如果翻译键不存在的话 `_` 函数将会返回 `messages.welcome`。

camel_case()

`camel_case` 函数将给定字符串转化为符合驼峰式命名规则的字符串：

```
$camel = camel_case('foo_bar');
// fooBar
```

class_basename()

`class_basename` 返回给定类移除命名空间后的类名：

```
$class = class_basename('Foo\Bar\Baz');
// Baz
```

e0

`e` 函数在给定字符串上运行 `htmlentities` (`double_encode` 选项设置为 `false`)：

```
echo e('<html>foo</html>');
// &lt;html&ampgtfoo&lt;/html&ampgt
```

ends_with()

`ends_with` 函数判断给定字符串是否以给定值结尾：

```
$value = ends_with('This is my name', 'name');
```

```
// true
```

kebab_case()

`kebab_case` 函数将给定字符串转化为短划线分隔的字符串:

```
$converted = kebab_case('fooBar');
```

```
// foo-bar
```

preg_replace_array()

`preg_replace_array` 函数使用数组替换字符串序列中的给定模式:

```
$string = 'The event will take place between :start and :end';

$replaced = preg_replace_array('/:[a-z_]+/', ['8:30', '9:00'], $string);

// The event will take place between 8:30 and 9:00
```

snake_case()

`snake_case` 函数将给定字符串转化为下划线分隔的字符串:

```
$snake = snake_case('fooBar');
```

```
// foo_bar
```

starts_with()

`starts_with` 函数判断给定字符串是否以给定值开头:

```
$result = starts_with('This is my name', 'This');

// true
```

str_after()

`str_after` 函数返回字符串中给定值之后的所有字符:

```
$slice = str_after('This is my name', 'This is');

// ' my name'
```

str_before()

`str_before` 函数返回字符串给定值之前的所有字符:

```
$slice = str_before('This is my name', 'my name');

// 'This is '
```

str_contains()

`str_contains` 函数判断给定字符串是否包含给定值（大小写敏感）:

```
$contains = str_contains('This is my name', 'my');

// true
```

还可以传递数组值判断给定字符串是否包含数组中的任意值:

```
$contains = str_contains('This is my name', ['my', 'foo']);

// true
```

str_finish()

`str_finish` 函数添加给定值单个实例到字符串结尾 —— 如果原字符串不以给定值结尾的话:

```
$adjusted = str_finish('this/string', '/');

// this/string/

$adjusted = str_finish('this/string/', '/');

// this/string/
```

str_is()

`str_is` 函数判断给定字符串是否与给定模式匹配，星号可用于表示通配符:

```
$value = str_is('foo*', 'foobar');

// true

$value = str_is('baz*', 'foobar');

// false
```

str_limit()

`str_limit` 函数以指定长度截断字符串：

```
$truncated = str_limit('The quick brown fox jumps over the lazy dog', 20);

// The quick brown fox...
```

还可以传递第三个参数来改变字符串末尾字符：

```
$truncated = str_limit('The quick brown fox jumps over the lazy dog', 20, ' (...)');
```

// The quick brown fox (...)

Str::orderedUuid()

`Str::orderedUuid` 方法会生成一个「时间戳优先」的 UUID 以便更高效地存储到数据库索引字段：

```
use Illuminate\Support\Str;

return (string) Str::orderedUuid();
```

str_plural()

`str_plural` 函数将字符串转化为复数形式，该函数当前只支持英文：

```
$plural = str_plural('car');

// cars

$plural = str_plural('child');

// children
```

还可以传递整型数据作为第二个参数到该函数以获取字符串的单数或复数形式：

```
$plural = str_plural('child', 2);

// children

$plural = str_plural('child', 1);

// child
```

str_random()

`str_random` 函数通过指定长度生成随机字符串，该函数使用了 PHP 的 `random_bytes` 函数：

```
$string = str_random(40);
```

str_replace_array()

`str_replace_array` 函数使用数组在字符串序列中替换给定值：

```
$string = 'The event will take place between ? and ?';

$replaced = str_replace_array('?', ['8:30', '9:00'], $string);

// The event will take place between 8:30 and 9:00
```

str_replace_first()

`str_replace_first` 函数会替换字符串中第一次出现的值：

```
$replaced = str_replace_first('the', 'a', 'the quick brown fox jumps over the lazy dog');

// a quick brown fox jumps over the lazy dog
```

str_replace_last()

`str_replace_last` 函数会替换字符串中最后一次出现的值：

```
$replaced = str_replace_last('the', 'a', 'the quick brown fox jumps over the lazy dog');

// the quick brown fox jumps over a lazy dog
```

str_singular()

`str_singular` 函数将字符串转化为单数形式，该函数目前只支持英文：

```
$singular = str_singular('cars');

// car

$singular = str_singular('children');

// child
```

str_slug()

`str_slug` 函数将给定字符串生成 URL 友好的格式：

```
$title = str_slug("Laravel 5 Framework", "-");
```

```
// laravel-5-framework
```

str_start()

如果字符串没有以给定值开头的话 `str_start` 函数会将给定值添加到字符串最前面:

```
$adjusted = str_start('this/string', '/');
// /this/string

$adjusted = str_start('/this/string/', '/');
// /this/string
```

studly_case()

`studly_case` 函数将给定字符串转化为单词开头字母大写的格式:

```
$value = studly_case('foo_bar');
// FooBar
```

title_case()

`title_case` 函数将字符串转化为 `Title` 形式:

```
$title = title_case('a nice title uses the correct case');
// A Nice Title Uses The Correct Case
```

trans()

`trans` 函数使用 [本地文件](#) 翻译给定翻译键:

```
echo trans('messages.welcome');
```

如果指定翻译键不存在, `trans` 函数会返回给定键, 所以, 以上面的示例为例, 如果翻译键不存在, `trans` 函数会返回 `messages.welcome`。

trans_choice()

`trans_choice` 函数翻译带拐点的给定翻译键:

```
echo trans_choice('messages.notifications', $unreadCount);
```

如果指定的翻译键不存在, `trans_choice` 函数会将其返回。所以, 以上面的示例为例, 如果指定翻译键不存在 `trans_choice` 函数会返回 `messages.notifications`。

Str::uuid()

`Str::uuid` 方法会生成一个 UUID (版本 4) :

```
use Illuminate\Support\Str;
return (string) Str::uuid();
```

URL 函数

action()

`action` 函数为给定控制器动作生成 URL, 你不需要传递完整的命名空间到该控制器, 传递相对于命名空间 `App\Http\Controllers` 的类名即可:

```
$url = action('HomeController@index');
```

如果该方法接收路由参数, 你可以将其作为第二个参数传递进来:

```
$url = action('UserController@profile', ['id' => 1]);
```

asset()

`asset` 函数使用当前请求的 `scheme` (HTTP 或 HTTPS) 为前端资源生成一个 URL:

```
$url = asset('img/photo.jpg');
```

secure_asset()

`secure_asset` 函数使用 HTTPS 为前端资源生成一个 URL:

```
echo secure_asset('foo/bar.zip', $title, $attributes = []);
```

route()

`route` 函数为给定命名路由生成一个 URL:

```
$url = route('routeName');
```

如果该路由接收参数, 你可以将其作为第二个参数传递进来:

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

默认情况下, `route` 函数生成的是绝对 URL, 如果你想要生成一个相对 URL, 可以传递 `false` 作为第三个参数:

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

secure_url

`secure_url` 函数为给定路径生成完整的 HTTPS URL:

```
echo secure_url('user/profile');

echo secure_url('user/profile', [1]);
```

url()

`url` 函数为给定路径生成完整 URL:

```
echo url('user/profile');

echo url('user/profile', [1]);
```

如果没有提供路径，将会返回 `Illuminate\Routing\UrlGenerator` 实例:

```
echo url() ->current();
echo url() ->full();
echo url() ->previous();
```

其它函数

abort()

`abort` 函数会抛出一个被 `异常处理器` 渲染的 `HTTP 异常`:

```
abort(403);
```

还可以提供异常响应文本以及自定义响应头:

```
abort(403, 'Unauthorized.', $headers);
```

abort_if()

`abort_if` 函数在给定布尔表达式为 `true` 时抛出 `HTTP 异常`:

```
abort_if(! Auth::user() ->isAdmin(), 403);
```

和 `abort` 一样，你还可以传递异常响应文本作为第三个参数以及自定义响应头数组作为第四个参数。

abort_unless()

`abort_unless` 函数在给定布尔表达式为 `false` 时抛出 `HTTP 异常`:

```
abort_unless(Auth::user() ->isAdmin(), 403);
```

和 `abort` 一样，你还可以传递异常响应文本作为第三个参数以及自定义响应头数组作为第四个参数。

app()

`app` 函数返回 `服务容器` 实例:

```
$container = app();
```

还可以传递类或接口名从容器中解析它:

```
$api = app('HelpSpot\API');
```

auth()

`auth` 函数返回一个 `认证器` 实例，为方便起见你可以用其取代 `Auth` 门面:

```
$user = auth() ->user();
```

如果需要的话还可以指定你想要使用的 `guard` 实例:

```
$user = auth('admin') ->user();
```

back()

`back` 函数生成 `HTTP 重定向响应` 到用户前一个访问页面:

```
return back($status = 302, $headers = [], $fallback = false);

return back();
```

bcrypt()

`bcrypt` 函数使用 `Bcrypt` 对给定值进行 `哈希`，你可以用其替代 `Hash` 门面:

```
$password = bcrypt('my-secret-password');
```

broadcast()

`broadcast` 函数 `广播` 给定 `事件` 到监听器:

```
broadcast(new UserRegistered($user));
```

blank()

`blank` 函数返回给定值是否为空:

```
blank('');
blank(' ');
blank(null);
blank(collect());
```

```
// true
```

```
blank(0);
blank(true);
blank(false);

// false
```

与 `blank` 相对的是 `filled` 函数。

cache()

`cache` 函数可以用于从缓存中获取值，如果给定 `key` 在缓存中不存在，可选的默认值会被返回：

```
$value = cache('key');

$value = cache('key', 'default');
```

你可以通过传递数组键值对到函数来添加数据项到缓存。还需要传递缓存有效期（分钟数）：

```
cache(['key' => 'value'], 5);

cache(['key' => 'value'], now() -> addSeconds(10));
```

class_uses_recursive()

`class_uses_recursive` 函数某个类所使用的所有 trait，包括子类使用的：

```
$traits = class_uses_recursive(App\User::class);
```

collect()

`collect` 函数会根据提供的数据项创建一个集合：

```
$collection = collect(['taylor', 'abigail']);
```

config()

`config` 函数获取配置变量的值，配置值可以通过使用“.”号访问，包含文件名以及你想要访问的选项。如果配置选项不存在的话默认值将会被指定并返回：

```
$value = config('app.timezone');

$value = config('app.timezone', $default);
```

辅助函数 `config` 还可以用于在运行时通过传递键值对数组设置配置变量值：

```
config(['app.debug' => true]);
```

cookie()

`cookie` 函数可用于创建一个新的 `Cookie` 实例：

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

csrf_field()

`csrf_field` 函数生成一个包含 CSRF 令牌值的 HTML 隐藏字段，例如，使用 `Blade 语法` 示例如下：

```
{{ csrf_field() }}
```

csrf_token()

`csrf_token` 函数获取当前 CSRF 令牌的值：

```
$token = csrf_token();
```

dd()

`dd` 函数输出给定变量值并终止脚本执行：

```
dd($value);

dd($value1, $value2, $value3, ...);
```

如果你不想停止脚本的运行，可以使用 `dump` 函数。

decrypt()

`decrypt` 函数使用 Laravel 加密器对给定值进行解密：

```
$decrypted = decrypt($encrypted_value);
```

dispatch()

`dispatch` 函数推送一个新的任务到 Laravel 任务队列：

```
dispatch(new App\Jobs\SendEmails);
```

dispatch_now()

`dispatch_now` 函数会立即运行给定任务并返回 `handle` 方法处理结果：

```
$result = dispatch_now(new App\Jobs\SendEmails);
```

dump()

`dump` 函数会打印给定变量：

```
dump($value);

dump($value1, $value2, $value3, ...);
```

如果你想要在打印变量后终止脚本执行，可以使用 `dd` 函数替代之。

encrypt()

`encrypt` 函数使用 Laravel 加密器加密给定字符串：

```
$encrypted = encrypt($unencrypted_value);
```

env()

`env` 函数获取环境变量值或返回默认值：

```
$env = env('APP_ENV');
```

// 如果变量不存在返回默认值...

```
$env = env('APP_ENV', 'production');
```

注：如果你在开发过程中执行了 `config:cache` 命令，需要确保只在配置文件中调用了 `env`，一旦配置被缓存起来，`.env` 文件将不会被加载，因此所有对 `env` 函数的调用都会返回 `null`。

event()

`event` 函数分发给定事件到对应监听器：

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

factory()

`factory` 函数为给定类、名称和数量创建模型工厂构建器，可用于测试或数据填充：

```
$user = factory(App\User::class)->make();
```

filled()

`filled` 函数会返回给定值是否不为空：

```
filled(0);
filled(true);
filled(false);
```

// true

```
filled('');
filled(' ');
filled(null);
filled(collect());
```

// false

与 `filled` 相对的是 `blank` 函数。

info()

`info` 函数会记录信息到日志系统：

```
info('Some helpful information!');
```

还可以传递上下文数据数组到该函数：

```
info('User login attempt failed.', ['id' => $user->id]);
```

logger()

`logger` 函数可以用于记录 `debug` 级别的日志消息：

```
logger('Debug message');
```

同样，也可以传递上下文数据数组到该函数：

```
logger('User has logged in.', ['id' => $user->id]);
```

如果没有值传入该函数的话会返回日志实例：

```
logger()->error('You are not allowed here.');
```

method_field()

`method_field` 函数生成包含 HTTP 请求方法的 HTML `hidden` 表单字段，例如：

```
<form method="POST">
    {{ method_field('DELETE') }}
</form>
```

now()

`now` 函数为当前时间创建一个新的 `Illuminate\Support\Carbon` 实例：

```
$now = now();
```

old()

`old` 函数获取存放在一次性 Session 中上一次输入的值：

```
$value = old('value');
```

```
$value = old('value', 'default');
```

optional()

`optional` 函数接收任意参数并允许你访问对象上的属性或调用其方法。如果给定的对象为空，属性或方法调用返回 `null` 而不是出错：

```
return optional($user->address)->street;

{!! old('name', optional($user)->name) !!}
```

policy()

`policy` 函数为给定模型类获取对应策略实例：

```
$policy = policy(App\User::class);
```

redirect()

`redirect` 函数返回 HTTP 重定向响应，如果不带参数的话返回重定向器示例：

```
return redirect($to = null, $status = 302, $headers = [], $secure = null);

return redirect('/home');

return redirect()->route('route.name');
```

report()

`report` 函数会使用异常处理器的 `report` 方法报告异常：

```
report($e);
```

request()

`request` 函数返回当前请求实例或者获取一个输入项：

```
$request = request();

$value = request('key', $default);
```

rescue()

`rescue` 函数可以执行给定闭包并捕获执行过程中的所有异常。这些捕获的异常会发送给异常处理器的 `report` 方法，不过，请求会继续执行：

```
return rescue(function () {
    return $this->method();
});
```

还可以传递第二个参数到 `rescue` 函数，作为在执行闭包出现异常的情况下返回的默认值：

```
return rescue(function () {
    return $this->method();
}, false);

return rescue(function () {
    return $this->method();
}, function () {
    return $this->failure();
});
```

resolve()

`resolve` 函数使用服务容器将给定类或接口名解析为对应绑定实例：

```
$api = resolve('HelpSpot\API');
```

response()

`response` 函数创建一个响应实例或者获取响应工厂实例：

```
return response('Hello World', 200, $headers);

return response()->json(['foo' => 'bar'], 200, $headers);
```

retry()

`retry` 函数尝试执行给定回调直到达到最大执行次数，如果回调没有抛出异常，会返回对应的返回值。如果回调抛出了异常，会自动重试。如果超出最大执行次数，异常会被抛出：

```
return retry(5, function () {
    // Attempt 5 times while resting 100ms in between attempts...
}, 100);
```

session()

`session` 函数可以用于获取/设置 Session 值：

```
$value = session('key');
```

可以通过传递键值对数组到该函数的方式设置 Session 值：

```
session(['chairs' => 7, 'instruments' => 3]);
```

如果没有传入参数到 `session` 函数则返回 Session 存储器对象实例：

```
$value = session()->get('key');

session()->put('key', $value);
```

tap()

`tap` 函数接收两个参数：任意的 `$value` 和一个闭包。`$value` 会被传递到闭包然后通过 `tap` 函数返回。闭包返回值与函数返回值不相关：

```
$user = tap(User::first(), function ($user) {
    $user->name = 'taylor';

    $user->save();
});
```

如果没有传入闭包到 `tap` 函数，那么你可以调用给定 `$value` 上面的任意方法，调用方法的返回值永远都是 `$value`，不管在方法中定义的返回值是什么。例如，`Eloquent update` 方法通常返回一个整型，不过，我们可以通过 `tap` 函数强制该方法返回模型本身：

```
$user = tap($user)->update([
    'name' => $name,
    'email' => $email,
]);
```

today()

`today` 函数会为当前日期创建一个新的 `Illuminate\Support\Carbon` 实例：

```
$today = today();
```

throw_if()

`throw_if` 函数会在给定布尔表达式为 `true` 的情况下抛出给定异常：

```
throw_if(! Auth::user()->isAdmin(), AuthorizationException::class);

throw_if(
    ! Auth::user()->isAdmin(),
    AuthorizationException::class,
    'You are not allowed to access this page'
);
```

throw_unless()

`throw_unless` 函数会在给定布尔表达式为 `false` 的情况下抛出给定异常：

```
throw_unless(Auth::user()->isAdmin(), AuthorizationException::class);

throw_unless(
    Auth::user()->isAdmin(),
    AuthorizationException::class,
    'You are not allowed to access this page'
);
```

trait_uses_recursive()

`trait_uses_recursive` 函数会返回某个 trait 使用的所有 trait：

```
$traits = trait_uses_recursive(\Illuminate\Notifications\Notifiable::class);
```

transform()

`transform` 函数会在给定值不为空的情况下执行闭包并返回闭包结果：

```
$callback = function ($value) {
    return $value * 2;
};

$result = transform(5, $callback);

// 10
```

默认值或者闭包可以以第三个参数的方式传递给该函数，默认值在给定值为空的情况下返回：

```
$result = transform(null, $callback, 'The value is blank');

// The value is blank
```

validator()

`validator` 函数通过给定参数创建一个新的验证器实例，为方便起见可以使用它代替 `Validator` 门面：

```
$validator = validator($data, $rules, $messages);
```

value()

`value` 函数返回给定的值，不过，如果你传递一个闭包到该函数，该闭包将被执行并返回执行结果：

```
$result = value(true);

// true

$result = value(function () {
    return false;
});
```

```
// false

view()
view 函数获取一个视图实例:

return view('auth.login');
```

with()

with 函数返回给定的值，如果第二个参数是闭包，则返回闭包执行结果：

```
$callback = function ($value) {
    return (is_numeric($value)) ? $value * 2 : 0;
};

$result = with(5, $callback);

// 10

$result = with(null, $callback);

// 0

$result = with(5, null);

// 5
```

邮件

简介

Laravel 基于 [SwiftMailer](#) 库提供了一套干净、清爽的邮件 API。Laravel 为 SMTP、Mailgun、SparkPost、Amazon SES、PHP 的 mail 函数，以及 sendmail 提供了驱动，从而允许你快速通过本地或云服务发送邮件。

邮件驱动预备知识

基于 API 的驱动如 Mailgun 和 SparkPost 通常比 SMTP 服务器更简单、更快，所以如果可以的话，尽可能使用这些服务。所有的 API 驱动要求应用已经安装 Guzzle HTTP 库，你可以通过 Composer 包管理器来安装它：

```
composer require guzzlehttp/guzzle
```

Mailgun 驱动

要使用 Mailgun 驱动（Mailgun 前 10000 封邮件免费，后续收费），首先安装 Guzzle，然后在配置文件 config/mail.php 中设置 driver 选项为 mailgun。接下来，验证配置文件 config/services.php 包含如下选项：

```
'mailgun' => [
    'domain' => 'your-mailgun-domain',
    'secret' => 'your-mailgun-key',
],
```

SparkPost 驱动

要使用 SparkPost 驱动，首先安装 Guzzle，然后在配置文件 config/mail.php 中设置 driver 选项值为 sparkpost。接下来，验证配置文件 config/services.php 包含如下选项：

```
'sparkpost' => [
    'secret' => 'your-sparkpost-key',
],
```

SES 驱动

要使用 Amazon SES 驱动（收费），先安装 Amazon AWS 的 PHP SDK，你可以通过添加如下行到 composer.json 文件的 require 部分然后运行 composer update 命令来安装该库：

```
"aws/aws-sdk-php": "~3.0"
```

接下来，设置配置文件 config/mail.php 中的 driver 选项为 ses。然后，验证配置文件 config/services.php 包含如下选项：

```
'ses' => [
    'key' => 'your-ses-key',
    'secret' => 'your-ses-secret',
    'region' => 'ses-region', // e.g. us-east-1
],
```

学院君注：坏消息是以上驱动均不适用于国内开发者开发服务于国内用户的应用，所以基本可以忽略。

生成可邮寄类

在 Laravel 中，应用发送的每一封邮件都可以表示为“可邮寄”类，这些类都存放在 `app/Mail` 目录。如果没看到这个目录，别担心，它将会在你使用 `make:mail` 命令创建第一个可邮寄类时生成：

```
php artisan make:mail OrderShipped
```

默认生成的模板代码如下：

```
<?php

namespace App\Mail;

use ...

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * Create a new message instance.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Build the message.
     *
     * @return $this
     */
    public function build()
    {
        return $this->view( view: 'view.name' );
    }
}
```

编写可邮寄类

所有的可邮寄类配置都在 `build` 方法中完成，在这个方法中，你可以调用多个方法，例如 `from`、`subject`、`view` 和 `attach` 来配置邮件的内容和发送。

配置发件人

使用 `from` 方法

我们来看一下邮件发件人的配置，或者，换句话说，邮件来自于谁。有两种方式来配置发送者，第一种方式是在可邮寄类的 `build` 方法方法中调用 `from` 方法：

```
/**
 * 构建消息.
 *
 * @return $this
 */
public function build()
{
    return $this->from('example@example.com')
        ->view('emails.orders.shipped');
}
```

使用全局的 `from` 地址

不过，如果你的应用在所有邮件中都使用相同的发送地址，在每个生成的可邮寄类中都调用 `from` 方法就显得很累赘。取而代之地，你可以在配置文件 `config/mail.php` 中指定一个全局的发送地址，该地址可用于在所有可邮寄类中没有指定其它发送地址的场景下（即作为默认发件人）：

```
'from' => ['address' => 'example@example.com', 'name' => 'App Name'],
```

配置视图

你可以在可邮寄类的 `build` 方法中使用 `view` 方法来指定渲染邮件内容时使用哪个视图模板，由于每封邮件通常使用 **Blade 模板** 来渲染内容，所以你可以在构建邮件 HTML 时使用 Blade 模板引擎提供的所有功能：

```
/**
 * 构建消息.
 *
 * @return $this
 * @translator laravelacademy.org
 */
public function build()
{
    return $this->view('emails.orders.shipped');
}
```

注：你可以创建一个 `resources/views/emails` 目录来存放所有邮件模板，当然，你也可以将邮件模板放到 `resources/views` 目录下任意其它位置。

纯文本邮件

如果你想要定义一个纯文本格式的邮件，可以使用 `text` 方法。和 `view` 方法一样，`text` 方法接收一个用于渲染邮件内容的模板名，你既可以定义纯文本消息也可以定义 HTML 消息：

```
/**
 * 构建消息.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
        ->text('emails.orders.shipped_plain');
}
```

视图数据

通过公共属性

通常，我们需要传递一些数据到渲染邮件的 HTML 视图以供使用。有两种方式将数据传递到视图，第一种是可邮寄类的公共（public）属性在视图中自动生效，举个例子，我们将数据传递给可邮寄类的构造器并将数据设置给该类的公共属性：

```
<?php

namespace App\Mail;

use App\Order;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * 订单实例.
     *
     * @var Order
     */
    public $order;

    /**
     * 创建一个新的消息实例.
     *
     * @return void
     */
    public function __construct(Order $order)
    {
        $this->order = $order;
    }

}
```

```

 * 构建消息.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped');
}
}

```

数据被设置给公共属性后，将会在视图中自动生效，所以你可以像在 Blade 模板中访问其它数据一样访问它们：

```
<div>
    Price: {{ $order->price }}
</div>
```

通过 `with` 方法

如果你想要在数据发送到模板之前自定义邮件数据的格式，可以通过 `with` 方法手动传递数据到视图。一般情况下，你还是需要通过可邮寄类的构造器传递数据，不过，这次你需要设置数据为 `protected` 或 `private` 属性，这样，这些数据就不会在视图中自动生效。然后，当调用 `with` 方法时，传递数组数据到该方法以便数据在视图模板中生效：

```

<?php

namespace App\Mail;

use App\Order;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * 订单实例.
     *
     * @var Order
     */
    protected $order;

    /**
     * 创建一个新的实例.
     *
     * @return void
     */
    public function __construct(Order $order)
    {
        $this->order = $order;
    }

    /**
     * 构建消息.
     *
     * @return $this
     */
    public function build()
    {
        return $this->view('emails.orders.shipped')
            ->with([
                'orderName' => $this->order->name,
                'orderPrice' => $this->order->price,
            ]);
    }
}

```

数据通过 `with` 方法传递到视图后，将会在视图中自动生效，因此你也可以像在 Blade 模板访问其它数据一样访问传递过来的数据：

```
<div>
    Price: {{ $orderPrice }}
</div>
```

附件

要添加附件到邮件，可以在可邮寄类的 `build` 方法中使用 `attach` 方法。`attach` 方法接收完整的文件路径作为第一个参数：

```
/**
 * 构建消息.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
        ->attach('/path/to/file');
}
```

附加文件到消息时，还可以通过传递一个数组作为 `attach` 方法的第二个参数来指定文件显示名或 MIME 类型：

```
/**
 * 构建消息.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
        ->attach('/path/to/file', [
            'as' => 'name.pdf',
            'mime' => 'application/pdf',
        ]);
}
```

原生数据附件

`attachData` 方法可用于添加原生的字节字符串作为附件。如果你想在内存中生成 PDF，并且在不保存到磁盘的情况下将其添加到邮件作为附件，则可以使用该方法。`attachData` 方法接收数据字节作为第一个参数，文件名作为第二个参数，可选数组作为第三个参数：

```
/**
 * 构建消息.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
        ->attachData($this->pdf, 'name.pdf', [
            'mime' => 'application/pdf',
        ]);
}
```

内联附件

嵌套内联图片到邮件中通常是很笨重的，为此，Laravel 提供了便捷的方式附加图片到邮件并获取相应的 CID，要嵌入内联图片，在邮件视图中使用 `$message` 变量上的 `embed` 方法即可。Laravel 在所有邮件视图中注入 `$message` 变量并使其自动有效，所以你不用关心如何将这个变量手动传入视图：

```
<body>
    Here is an image:

    
</body>
```

注：`$message` 变量不适用于 markdown 消息。

嵌入原生数据附件

如果你已经有一个想要嵌入邮件模板的原生数据字符串，可以使用 `$message` 变量上的 `embedData` 方法：

```
<body>
    Here is an image from raw data:

    
</body>
```

自定义 SwiftMailer 消息

`Mailable` 基类上的 `withSwiftMessage` 方法允许你注册一个在发送消息之前可以被原生 `SwiftMailer` 消息实例调用的回调。这赋予了我们在发送消息前定制消息的机会：

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    $this->view('emails.orders.shipped');

    $this->withSwiftMessage(function ($message) {
        $message->getHeaders()
            ->addTextHeader('Custom-Header', 'HeaderValue');
    });
}
```

Markdown 邮件类

Markdown 邮件消息允许你在可邮寄类中利用预置模板和邮件通知组件。因为这些消息以 Markdown 格式编写，Laravel 还可以为它们渲染出高颜值、响应式的 HTML 模板，同时自动生成纯文本的副本。

生成 Markdown 邮件类

要生成带有相应 Markdown 模板的可邮寄类，可以在使用 Artisan 命令 `make:mail` 时带上 `--markdown` 选项：

```
php artisan make:mail OrderShipped --markdown=emails.orders.shipped
```

然后，配置可邮寄类的 `build` 方法时，使用 `markdown` 方法取代 `view` 方法。`markdown` 方法接收 Markdown 模板的名称和一个可选的在模板中生效的数据数组：

```
/**
 * 构建消息.
 *
 * @return $this
 */
public function build()
{
    return $this->from('example@example.com')
        ->markdown('emails.orders.shipped');
}
```

编写 Markdown 消息

Markdown 邮件类组合使用了 Blade 组件和 Markdown 语法，从而让你在不脱离 Laravel 预置组件的情况下轻松构建邮件消息：

```
@component('mail::message')
# Order Shipped

Your order has been shipped!

@component('mail::button', ['url' => $url])
View Order
@endcomponent

Thanks, <br>
{{ config('app.name') }}
@endcomponent
```

注：在编写 Markdown 邮件时不要使用多余的缩进，Markdown 解析器会将缩进渲染成代码区块。

按钮组件

按钮组件渲染一个居中的按钮链接。该组件接收两个参数，`url` 和可选的 `color`，支持的颜色有 `blue`，`green` 和 `red`。你可以添加任意数量的按钮组件到消息中：

```
@component('mail::button', ['url' => $url, 'color' => 'green'])
View Order
@endcomponent
```

面板组件

面板组件将给定的文字区块渲染到一个面板中，并且有一个淡淡的背景色与周围的消息区分开。适用于需要引起注意的文字区块：

```
@component('mail::panel')
This is the panel content.
@endcomponent
```

表格组件

表格组件允许你将一个 Markdown 表格转化为 HTML 表格。该组件接收 Markdown 表格作为其内容。表格列对齐支持使用默认的 Markdown 表格列对齐语法：

```
@component('mail::table')
| Laravel      | Table          | Example   |
| ----- | :-----: | -----: |
| Col 2 is    | Centered     | $10       |
| Col 3 is    | Right-Aligned | $20       |
@endcomponent
```

自定义组件

你可以导出所有 Markdown 邮件组件到自己的应用中进行自定义，使用 Artisan 命令 `vendor:publish` 来发布 `laravel-mail` 资源标签：

```
php artisan vendor:publish --tag=laravel-mail
```

该命令会发布 Markdown 邮件组件到 `resources/views/vendor/mail` 目录。`mail` 目录包含 `html` 和 `markdown` 目录，每个子目录中又包含各自的所有有效组件。你可以按照自己的需要编辑这些组件。

自定义 CSS

导出组件之后，`resources/views/vendor/mail/html/themes` 目录将会包含一个默认的 `default.css` 文件，你可以在这个文件中自定义 CSS，这样 Markdown 邮件消息的 HTML 样式就会自动调整。

如果你想要为 Markdown 组件构建全新的主题，只需在 `html/themes` 目录中编写一个新的 CSS 文件并修改 `mail` 配置文件的 `theme` 选项即可。

发送邮件

要发送一条信息，可以使用 `Mail` 门面上的 `to` 方法。`to` 方法接收邮箱地址、用户实例或用户集合作为参数。如果传递的是对象或对象集合，在设置邮件收件人的时候邮件会自动使用它们的 `email` 和 `name` 属性，所以事先要确保这些属性在相应类上有效。指定好收件人以后，传递一个可邮寄类的实例到 `send` 方法：

```
<?php

namespace App\Http\Controllers;

use App\Order;
use App\Mail\OrderShipped;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Mail;
use App\Http\Controllers\Controller;

class OrderController extends Controller
{
    /**
     * 处理传入订单.
     *
     * @param Request $request
     * @param int $orderId
     * @return Response
     * @translator laravelacademy.org
     */
    public function ship(Request $request, $orderId)
    {
        $order = Order::findOrFail($orderId);

        // 处理订单...

        Mail::to($request->user())->send(new OrderShipped($order));
    }
}
```

当然，发送邮件消息时并不仅限于指定单个收件人，你可以通过单个方法链调用设置「to」、「cc」以及「bcc」：

```
Mail::to($request->user())
->cc($moreUsers)
->bcc($evenMoreUsers)
->send(new OrderShipped($order));
```

渲染可邮寄类

有时候你可能想要在不发送可邮寄类的情况下捕获其 HTML 内容，要实现这个功能，可以调用可邮寄类的 `render` 方法，该方法会将处理后的邮件内容以字符串方式返回：

```
$invoice = App\Invoice::find(1);

return (new App\Mail\InvoicePaid($invoice)) ->render();
```

在浏览器中预览邮件

设计邮件的模板时，如果可以像普通的 Blade 模板一样快速预览渲染后的邮件是很方便的。因此，Laravel 允许你从路由闭包或控制器中直接返回可邮寄类。当可邮寄类返回时，就会在浏览器渲染并展示，从而方便你快速预览而不必真的发送邮件后查看：

```
Route::get('/mailable', function () {
    $invoice = App\Invoice::find(1);

    return new App\Mail\InvoicePaid($invoice);
});
```

邮件队列

邮件消息队列

由于发送邮件消息可能会大幅度延长应用的响应时间，许多开发者选择将邮件发送放到队列中在后台发送，Laravel 中可以使用内置的[统一队列 API](#) 来实现这一功能。要将邮件消息推送到队列，可以在指定消息的接收者后使用 `Mail` 门面上的 `queue` 方法：

```
Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->queue(new OrderShipped($order));
```

该方法自动将邮件任务推送到队列以便在后台发送。当然，你需要在使用该特性前[配置队列](#)。

延迟消息队列

如果你想要延迟队列中邮件消息的发送，可以使用 `later` 方法。`later` 方法的第一个参数接收一个 `DateTime` 实例来表示邮件发送时间：

```
$when = Carbon\Carbon::now()->addMinutes(10);

Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->later($when, new OrderShipped($order));
```

推送到指定队列

由于通过 `make:mail` 命令生成的所有可邮寄类都使用了 `Illuminate\Bus\Queueable` trait，所以你可以调用可邮寄类实例上的 `onQueue` 和 `onConnection` 方法，以便为消息指定连接和队列名称：

```
$message = (new OrderShipped($order))
    ->onConnection('sq')
    ->onQueue('emails');

Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->queue($message);
```

默认队列

如果你的可邮寄类总是想要推送到队列，可以在该类上实现 `ShouldQueue` 契约。这样，即使你调用 `send` 方法，可邮寄类还是会被推送到队列，因为它实现了这个契约：

```
use Illuminate\Contracts\Queue\ShouldQueue;

class OrderShipped extends Mailable implements ShouldQueue
{
    //
```

邮件 & 本地开发

本地环境开发发送邮件的应用时，你可能不想要真的发送邮件到有效的电子邮件地址，而只是想要做下测试。为此，Laravel 提供了几种方式“取消”邮件的实际发送。

日志驱动

一种解决方案是在本地开发时使用 `log` 邮件驱动。该驱动将所有邮件信息写到日志文件中以备查看，想要了解更多关于每个环境的应用配置信息，查看[配置文档](#)。

通用配置

Laravel 提供的另一种解决方案是为框架发送的所有邮件设置通用收件人，这样的话，所有应用生成的邮件将会被发送到指定地址，而不是实际发送邮件指定的地址。这可以通过在配置文件 `config/mail.php` 中设置 `to` 选项来实现：

```
'to' => [
    'address' => 'example@example.com',
    'name' => 'Example'
],
```

Mailtrap

最后，你可以使用 [Mailtrap](#) 服务和 `smtp` 驱动发送邮件信息到“虚拟”邮箱，这种方法允许你在 Mailtrap 的消息查看器中查看最终的邮件。

事件

Laravel 会在发送邮件消息前触发两个事件，`MessageSending` 事件在消息发送前触发，`MessageSent` 事件在消息发送后触发。需要注意的是这两个事件是在邮件被发送前后触发，而不是推送到队列时，你可以在 `EventServiceProvider` 中注册对应的事件监听器：

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Mail\Events\MessageSending' => [
        'App\Listeners\LogSendingMessage',
    ],
    'Illuminate\Mail\Events\MessageSent' => [
        'App\Listeners\LogSentMessage',
    ],
];
```

通知

简介

除了支持[发送邮件](#)之外，Laravel 还支持通过多种传输通道发送通知，这些通道包括邮件、短信（通过 [Nexmo](#)）以及 [Slack](#) 等。通知可以存储在数据库以便后续在 Web 界面中显示。

通常，通知都是很短的、用于告知用户应用中所发生事件的消息。例如，如果你在开发一个计费应用，则需要通过邮件或短信等渠道给用户发送“账单支付”通知。

创建通知

在 Laravel 中，每个通知都以单独类的形式存在（通常存放在 `app/Notifications` 目录），如果在应用中没看到这个目录，别担心，它将会在你运行 Artisan 命令 `make:notification` 的时候自动创建：

```
php artisan make:notification InvoicePaid
```

该命令会在 `app/Notifications` 目录下生成一个新的通知类，每个通知类都包含一个 `via` 方法以及多个消息构建方法（如 `toMail` 或 `toDatabase`），这些消息构建方法用于将通知转化成为特定渠道优化的消息。默认生成的模板代码如下：

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Notifications\Notification;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Messages\MailMessage;

class InvoicePaid extends Notification
{
    use Queueable;

    /**
     * Create a new notification instance.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }
}
```

```

/**
 * Get the notification's delivery channels.
 *
 * @param mixed $notifiable
 * @return array
 */
public function via($notifiable)
{
    return ['mail'];
}

/**
 * Get the mail representation of the notification.
 *
 * @param mixed $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)
        ->line('The introduction to the notification.')
        ->action('Notification Action', url('/'))
        ->line('Thank you for using our application!');
}

/**
 * Get the array representation of the notification.
 *
 * @param mixed $notifiable
 * @return array
 */
public function toArray($notifiable)
{
    return [
        //
    ];
}
}

```

发送通知

使用 Notifiable Trait

通知可以通过两种方式发送：使用 `Notifiable` trait 提供的 `notify` 方法或者使用 `Notification` 门面。首先，我们使用 `Notifiable` trait 来测试：

```

<?php

namespace App;

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

class User extends Authenticatable
{
    use Notifiable;
}

```

该 trait 被默认的 `App\User` 模型使用并提供一个可用于发送通知的方法：`notify`。`notify` 方法接收一个通知实例：

```

use App\Notifications\InvoicePaid;

$user->notify(new InvoicePaid($invoice));

```

注：你可以在任何模型中使用 `Illuminate\Notifications\Notifiable` trait，不限于只在 `User` 模型中使用。

使用 Notification 门面

作为替代方案，还可以通过 `Notification` 门面发送通知。这在你需要发送通知给多个用户的时候很有用，要使用这个门面发送通知，需要将所有被通知用户和通知实例传递给 `send` 方法：

```
Notification::send($users, new InvoicePaid($invoice));
```

指定传输通道

每个通知类都有一个 `via` 方法用于决定通知通过何种通道传输，Laravel 开箱支持 `mail`、`database`、`broadcast`、`nexmo` 以及 `slack` 通道。
注：如果你想要使用其他传输通道，比如 `Telegram` 或 `Pusher`，参考社区提供的驱动：[Laravel 通知通道网站](#)。

`via` 方法接收一个 `$notifiable` 实例，用于指定通知被发送到的类实例。你可以使用 `$notifiable` 来判断通知通过何种通道传输：

```
/**
 * 获取通知的传递渠道.
 *
 * @param mixed $notifiable
 * @return array
 * @translator laravelacademy.org
 */
public function via($notifiable)
{
    return $notifiable->prefers_sms ? ['nexmo'] : ['mail', 'database'];
}
```

通知队列

注：使用通知队列前需要配置队列并[开启一个队列任务](#)。

发送通知可能是耗时的，尤其是通道需要调用额外的 API 来传输通知。为了加速应用的响应时间，可以将通知推送到队列中异步发送，而要实现推送通知到队列，可以让对应通知类实现 `ShouldQueue` 接口并使用 `Queueable` trait。如果通知类是通过 `make:notification` 命令生成的，那么该接口和 trait 已经默认导入，你可以快速将它们添加到通知类：

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Notifications\Notification;
use Illuminate\Contracts\Queue\ShouldQueue;

class InvoicePaid extends Notification implements ShouldQueue
{
    use Queueable;

    // ...
}
```

`ShouldQueue` 接口被添加到通知类以后，你可以像之前一样正常发送通知，Laravel 会自动检测到 `ShouldQueue` 接口然后将通知推送到队列：

```
$user->notify(new InvoicePaid($invoice));
```

如果你想要延迟通知的发送，可以在通知实例后加上 `delay` 方法：

```
$when = Carbon::now()->addMinutes(10);

$user->notify((new InvoicePaid($invoice))->delay($when));
```

按需通知

有时候你可能需要发送通知给某个用户，但是该用户不存在于应用的用户系统中，要实现这一目的，我们使用 `Notification::route` 方法在发送通知之前指定特别的通知路由：

```
Notification::route('mail', 'taylor@laravel.com')
    ->route('nexmo', '5555555555')
    ->notify(new InvoicePaid($invoice));
```

邮件通知

格式化邮件消息

如果通知支持以邮件方式发送，你需要在通知类上定义一个 `toMail` 方法。该方法会接收一个 `$notifiable` 实体并返回 `Illuminate\Notifications\Messages\MailMessage` 实例。邮件消息可以包含多行文本以及对动作的调用，让我们来看一个 `toMail` 方法的示例：

```
/**
 * 获取通知对应的邮件.
 *
 * @param mixed $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
```

```
public function toMail($notifiable)
{
    $url = url('/invoice/'.$this->invoice->id);

    return (new MailMessage)
        ->greeting('Hello!')
        ->line('One of your invoices has been paid!')
        ->action('View Invoice', $url)
        ->line('Thank you for using our application!');
}
```

注：注意到我们在 `message` 方法中使用了 `$this->invoice->id`，你可以传递任何通知生成消息所需要的数据到通知的构造器。在这个例子中，我们注册了一条问候、一行文本、对动作的调用以及另一行文本。`MailMessage` 对象提供的这些方法让格式化短小的事务邮件变得简单快捷。`mail` 通道会将消息组件转化为美观的、响应式的、带有纯文本副本的 HTML 邮件模板。下面是一个通过 `mail` 通道生成的邮件示例：



Hello!

One of your invoices has been paid!

[View Invoice](#)

Thank you for using our application!

Regards,
Laravel

If you're having trouble clicking the "View Invoice" button, copy and paste the URL below into your web browser:

<https://example.com/invoice/1>

© 2016 [Laravel](#). All rights reserved.

注：发送邮件通知时，确保在配置文件 `config/app.php` 中设置了 `name` 的值，该值将会用在邮件通知消息的头部和尾部。

其他通知格式化选项

除了在通知类中定义多行文本之外，你还可以使用 `view` 方法来指定一个自定义的、用于渲染通知邮件的模板：

```
/**
 * 获取通知邮件.
 *
 * @param mixed $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 * @translator laravelacademy.org
 */
public function toMail($notifiable)
{
    return (new MailMessage)->view(
        'emails.name', ['invoice' => $this->invoice]
    );
}
```

此外，你可以从 `toMail` 方法中返回一个可邮寄对象：

```
use App\Mail\InvoicePaid as Mailable;

/**
 * 获取通知邮件
 *
 * @param mixed $notifiable
 * @return Mailable
 */

```

```
public function toMail($notifiable)
{
    return new Mailable($this->invoice)->to($this->user->email);
}
```

错误消息

一些通知会告知用户错误信息，例如失败的订单支付。你可以在构建消息的时候调用 `error` 方法来指示该邮件消息表示错误信息。在邮件消息中使用 `error` 方法时，动作按钮将会变成红色：

```
/**
 * 获取通知邮件.
 *
 * @param mixed $notifiable
 * @return \Illuminate\Notifications\Message
 */
public function toMail($notifiable)
{
    return (new MailMessage)
        ->error()
        ->subject('Notification Subject')
        ->line('...');

}
```

自定义接收人

通过 `mail` 通道发送通知时，通知系统会自动在被通知实体上查找 `email` 属性，你可以通过在该实体上定义一个 `routeNotificationForMail` 来自定义使用哪个邮箱地址发送通知：

```
<?php

namespace App;

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

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Route notifications for the mail channel.
     *
     * @return string
     */
    public function routeNotificationForMail()
    {
        return $this->email_address;
    }
}
```

自定义主题

默认情况下，邮件的主题就是格式为“标题风格”的通知类名，因此，如果通知类被命名为 `InvoicePaid`，邮件的主题就是 `Invoice Paid`，如果你想要为消息指定明确的主题，可以在构建消息的时候调用 `subject` 方法：

```
/**
 * Get the mail representation of the notification.
 *
 * @param mixed $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)
        ->subject('Notification Subject')
        ->line('...');

}
```

自定义模板

你可以通过发布通知扩展包的资源来修改邮件通知所使用的 HTML 和纯文本模板。运行完下面这个命令之后，邮件通知模板将会存放到了 `resources/views/vendor/notifications` 目录：

```
php artisan vendor:publish --tag=laravel-notifications
```

Markdown 邮件通知

Markdown 邮件通知允许你利用邮件通知的预置模板，从而让你可以自由编写更长、更具个性化的消息。因为这些消息以 Markdown 格式编写，Laravel 还可以为它们渲染出高颜值、响应式的 HTML 模板，同时自动生成纯文本的副本。

生成消息

要生成带有相应 Markdown 模板的通知，可以在使用 Artisan 命令 `make:notification` 时带上 `--markdown` 选项：

```
php artisan make:notification InvoicePaid --markdown=mail.invoice.paid
```

和其他邮件通知一样，使用 Markdown 模板的通知类也要定义一个 `toMail` 方法。不过，你可以使用 `markdown` 方法取代构造通知的 `line` 和 `action` 方法来指定要使用的 Markdown 模板名称：

```
/**
 * Get the mail representation of the notification.
 *
 * @param mixed $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 * @translator laravelacademy.org
 */
public function toMail($notifiable)
{
    $url = url('/invoice/'.$this->invoice->id);

    return (new MailMessage)
        ->subject('Invoice Paid')
        ->markdown('mail.invoice.paid', ['url' => $url]);
}
```

编写消息

Markdown 邮件通知联合使用了 Blade 组件和 Markdown 语法，从而让你在不脱离 Laravel 预置组件的情况下轻松构建通知：

```
@component('mail::message')
# Invoice Paid

Your invoice has been paid!

@component('mail::button', ['url' => $url])
View Invoice
@endcomponent

Thanks,<br>
{{ config('app.name') }}
@endcomponent
```

按钮组件

按钮组件渲染一个居中的按钮链接。该组件接收两个参数，`url` 和可选的 `color`，支持的颜色有 `blue`，`green` 和 `red`。你可以添加任意数量的按钮组件到消息中：

```
@component('mail::button', ['url' => $url, 'color' => 'green'])
View Invoice
@endcomponent
```

面板组件

面板组件将给定的文字区块渲染到一个面板中，并且有一个淡淡的背景色与周围的消息区分开。适用于需要引起注意的文字区块：

```
@component('mail::panel')
This is the panel content.
@endcomponent
```

表格组件

表格组件允许你将一个 Markdown 表格转化为 HTML 表格。该组件接收 Markdown 表格作为其内容。表格列对齐支持使用默认的 Markdown 表格列对齐语法：

```
@component('mail::table')
| Laravel      | Table          | Example   |
| ----- | :-----: | -----: |
| Col 2 is    | Centered     | $10       |
| Col 3 is    | Right-Aligned | $20       |
@endcomponent
```

自定义组件

你可以导出所有 Markdown 通知组件到应用中进行自定义，要导出组件，使用 Artisan 命令 `vendor:publish` 来发布 `laravel-mail` 资源标签：

```
php artisan vendor:publish --tag=laravel-mail
```

该命令会发布 Markdown 邮件通知组件到 `resources/views/vendor/mail` 目录。`mail` 目录包含 `html` 和 `markdown` 目录，每个子目录中又包含各自的所有有效组件。你可以按照自己的喜好自由编辑这些组件。

自定义 CSS

导出组件之后，`resources/views/vendor/mail/html/themes` 目录将会包含一个默认的 `default.css` 文件，你可以在这个文件中自定义 CSS，这样 Markdown 通知的 HTML 样式就会自动调整。

如果你想要为 Markdown 组件构建全新的主题，只需在 `html/themes` 目录中编写一个新的 CSS 文件并修改 `mail` 配置文件的 `theme` 选项即可。

数据库通知

预备知识

`database` 通知通道会在数据表中存储通知信息，该表包含诸如通知类型以及用于描述通知的自定义 JSON 数据之类的信息。

你可以在用户界面中查询这个数据表来展示通知，不过，在此之前，需要创建数据表来保存信息，你可以使用 `notifications:table` 命令来生成迁移文件然后以此生成相应数据表：

```
php artisan notifications:table
```

```
php artisan migrate
```

格式化数据库通知

如果一个通知支持存放在数据表，则需要在通知类中定义 `toDatabase` 或 `toArray` 方法，该方法接收一个 `$notifiable` 实体并返回原生的 PHP 数组。返回的数组会被编码为 JSON 格式然后存放到 `notifications` 表的 `data` 字段。让我们来看一个 `toArray` 方法的例子：

```
/**
 * Get the array representation of the notification.
 *
 * @param mixed $notifiable
 * @return array
 */
public function toArray($notifiable)
{
    return [
        'invoice_id' => $this->invoice->id,
        'amount' => $this->invoice->amount,
    ];
}
```

`toDatabase` Vs. `toArray`

`toArray` 方法还被 `broadcast` 通道用来判断广播什么数据到 JavaScript 客户端，如果你想要为 `database` 和 `broadcast` 通道提供两种不同的数组表示，则需要定义一个 `toDatabase` 方法来取代 `toArray` 方法。

访问通知

通知被存放到数据表之后，需要在被通知实体中有一个便捷的方式来访问它们。Laravel 默认提供的 `App\User` 模型引入的 `Illuminate\Notifications\Notifiable` trait 包含了返回实体对应通知的 Eloquent 关联关系方法 `notifications`，要获取这些通知，可以像访问其它 Eloquent 关联关系一样访问该关联方法，默认情况下，通知按照 `created_at` 时间戳排序：

```
$user = App\User::find(1);

foreach ($user->notifications as $notification) {
    echo $notification->type;
}
```

如果你只想获取“未读”通知，可使用关联关系 `unreadNotifications`，同样，这些通知也按照 `created_at` 时间戳排序：

```
$user = App\User::find(1);

foreach ($user->unreadNotifications as $notification) {
    echo $notification->type;
}
```

注：要想从 JavaScript 客户端访问通知，需要在应用中定义一个通知控制器为指定被通知实体（比如当前用户）返回通知，然后从 JavaScript 客户端发送一个 HTTP 请求到控制器对应 URI。

标记通知为已读

一般情况下，我们会将用户浏览过的通知标记为已读，`Illuminate\Notifications\Notifiable` trait 提供了一个 `markAsRead` 方法，用于更新对应通知数据库纪录上的 `read_at` 字段：

```
$user = App\User::find(1);
```

```
foreach ($user->unreadNotifications as $notification) {
    $notification->markAsRead();
}
```

如果觉得循环遍历每个通知太麻烦，可以直接在通知集合上调用 `markAsRead` 方法：

```
$user->unreadNotifications->markAsRead();
```

还可以使用批量更新方式标记通知为已读，无需先从数据库获取通知：

```
$user = App\User::find(1);

$user->unreadNotifications()->update(['read_at' => now()]);
```

当然，你也可以通过 `delete` 方法从数据库中移除这些通知：

```
$user->notifications()->delete();
```

广播通知

预备知识

在进行广播通知之前，需要配置并了解 [事件广播](#) 服务，事件广播为 JavaScript 客户端响应服务端触发的事件铺平了道路。

格式化广播通知

`broadcast` 通道广播通知使用了 Laravel 的事件广播服务，从而允许 JavaScript 客户端实时捕获通知。如果通知支持广播，则需要在通知类上定义 `toBroadcast` 方法，该方法接收一个 `$notifiable` 实体并返回一个 `BroadcastMessage` 实例，返回的数组会编码成 JSON 格式然后广播到 JavaScript 客户端。让我们来看一个 `toBroadcast` 方法的示例：

```
use Illuminate\Notifications\Messages\BroadcastMessage;

/**
 * Get the broadcastable representation of the notification.
 *
 * @param mixed $notifiable
 * @return BroadcastMessage
 */
public function toBroadcast($notifiable)
{
    return new BroadcastMessage([
        'invoice_id' => $this->invoice->id,
        'amount' => $this->invoice->amount,
    ]);
}
```

广播队列配置

所有广播通知都会被推送到广播队列，如果你想要配置广播操作队列的连接或名称，可以使用 `BroadcastMessage` 的 `onConnection` 和 `onQueue` 方法：

```
return new BroadcastMessage($data)
    ->onConnection('sqns')
    ->onQueue('broadcasts');
```

除了指定的数据之外，广播通知还包含一个 `type` 字段，用于包含通知的类名。

监听通知

通知将会以格式化为 `{notifiable}.(id)` 的形式在私有频道上广播，因此，如果你要发送通知到 ID 为 1 的 `App\User` 实例，那么该通知将会在私有频道 `App.User.1` 上进行广播，如果使用了 [Laravel Echo](#)，可以使用辅助函数 `notification` 轻松在某个频道上监听通知：

```
Echo.private('App.User.' + userId)
    .notification(notification) => {
        console.log(notification.type);
});
```

自定义通知通道

如果你想要自定义被通知实体在那个通道上接收广播通知，可以在被通知实体上定义一个 `receivesBroadcastNotificationsOn` 方法：

```
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Foundation\Auth\User as Authenticatable;
```

```
class User extends Authenticatable
{
    use Notifiable;

    /**
     * 用户接收广播通知的通道.
     *
     * @return array
     */
    public function receivesBroadcastNotificationsOn()
    {
        return 'users.'.$this->id;
    }
}
```

短信 (SMS) 通知

预备知识

Laravel 基于 [Nexmo](#) 发送短信通知，在使用 Nexmo 发送通知前，需要安装对应 Composer 依赖包 `nexmo/client` 并在配置文件 `config/services.php` 中进行相应配置。你可以从拷贝以下示例配置开始：

```
'nexmo' => [
    'key' => env('NEXMO_KEY'),
    'secret' => env('NEXMO_SECRET'),
    'sms_from' => '15556666666',
],
```

`sms_from` 配置项就是你用于发送短信消息的手机号码，你需要在 Nexmo 控制面板中为应用生成一个手机号码。

格式化短信通知

如果通知支持以短信方式发送，需要在通知类上定义一个 `toNexmo` 方法。该方法接收一个 `$notifiable` 实体并返回 `Illuminate\Notifications\Messages\NexmoMessage` 实例：

```
/**
 * Get the Nexmo / SMS representation of the notification.
 *
 * @param mixed $notifiable
 * @return NexmoMessage
 */
public function toNexmo($notifiable)
{
    return (new NexmoMessage)
        ->content('Your SMS message content');
}
```

Unicode 内容

如果你的短信消息包含 Unicode 字符，需要在构造 `NexmoMessage` 实例时调用 `unicode` 方法：

```
/**
 * Get the Nexmo / SMS representation of the notification.
 *
 * @param mixed $notifiable
 * @return NexmoMessage
 */
public function toNexmo($notifiable)
{
    return (new NexmoMessage)
        ->content('Your unicode message')
        ->unicode();
}
```

自定义来源号码

如果你要通过与配置文件 `config/services.php` 中指定的手机号不同的其他号码发送通知，可以使用 `NexmoMessage` 实例上的 `from` 方法：

```
/**
 * Get the Nexmo / SMS representation of the notification.
 *
 * @param mixed \notifiable
 * @return NexmoMessage
 */
public function toNexmo(\notifiable)
```

```
{
return (new NexmoMessage)
->content('Your SMS message content')
->from('15554443333');
}
```

短信通知路由

使用 `nexmo` 通道发送通知的时候，通知系统会自动在被通知实体上查找 `phone_number` 属性。如果你想要自定义通知被发送到的手机号码，可以在该实体上定义一个 `routeNotificationForNexmo` 方法：

```
<?php

namespace App;

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

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Route notifications for the Nexmo channel.
     *
     * @return string
     */
    public function routeNotificationForNexmo()
    {
        return $this->phone;
    }
}
```

Slack 通知

预备知识

在通过 Slack 发送通知前，必须通过 Composer 安装 Guzzle HTTP 库（已安装忽略）：

```
composer require guzzlehttp/guzzle
```

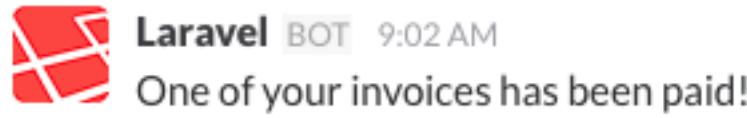
此外，你还要为 Slack 组配置一个「Incoming Webhook」集成。该集成会在你进行 [Slack 通知路由](#) 的时候提供一个 URL。

格式化 Slack 通知

如果通知支持通过 Slack 消息发送，则需要在通知类上定义一个 `toSlack` 方法，该方法接收一个 `$notifiable` 实体并返回 `Illuminate\Notifications\Messages\SlackMessage` 实例，该实例包含文本内容以及格式化额外文本或数组字段的“附件”。让我们来看一个基本的 `toSlack` 使用示例：

```
/**
 * Get the Slack representation of the notification.
 *
 * @param mixed $notifiable
 * @return SlackMessage
 */
public function toSlack($notifiable)
{
    return (new SlackMessage)
        ->content('One of your invoices has been paid!');
}
```

在这个例子中，我们只发送一行简单的文本到 Slack，最终创建的消息如下：



自定义发送者 & 接收者

你可以使用 `from` 和 `to` 方法自定义发送者和接收者，`from` 方法接收一个用户名和 emoji 标识，而 `to` 方法接收通道或用户名：

```
/**
 * Get the Slack representation of the notification.
 *
 * @param mixed $notifiable
 * @return SlackMessage

```

```
/*
public function toSlack($notifiable)
{
    return (new SlackMessage)
        ->from('Ghost', ':ghost:')
        ->to('#other')
        ->content('This will be sent to #other');
}
```

还可以使用图片作为 logo 用以取代 emoji:

```
/***
 * Get the Slack representation of the notification.
 *
 * @param mixed $notifiable
 * @return SlackMessage
 */
public function toSlack($notifiable)
{
    return (new SlackMessage)
        ->from('Laravel')
        ->image('https://laravel.com/favicon.png')
        ->content('This will display the Laravel logo next to the message');
}
```

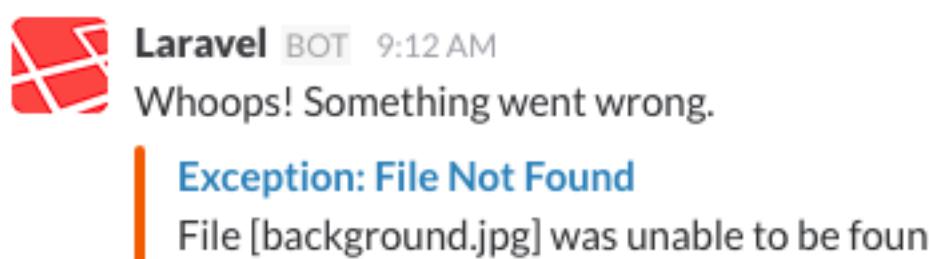
Slack 附件

你还可以添加“附件”到 Slack 消息。相对简单文本消息，附件可以提供更加丰富的格式选择。在这个例子中，我们会发送一个在应用程序中出现的异常错误通知，包含查看更多异常细节的链接:

```
/***
 * Get the Slack representation of the notification.
 *
 * @param mixed $notifiable
 * @return SlackMessage
 */
public function toSlack($notifiable)
{
    $url = url('/exceptions/'.$this->exception->id);

    return (new SlackMessage)
        ->error()
        ->content('Whoops! Something went wrong.')
        ->attachment(function ($attachment) use ($url) {
            $attachment->title('Exception: File Not Found', $url)
            ->content('File [background.jpg] was not found.');
        });
}
```

上述代码会生成如下 Slack 消息:



附件还允许你指定要呈献给用户的数组数据。为了提高可读性，给定的数组会以表格形式展示:

```
/***
 * Get the Slack representation of the notification.
 *
 * @param mixed $notifiable
 * @return SlackMessage
 */
public function toSlack($notifiable)
{
    $url = url('/invoices/'.$this->invoice->id);

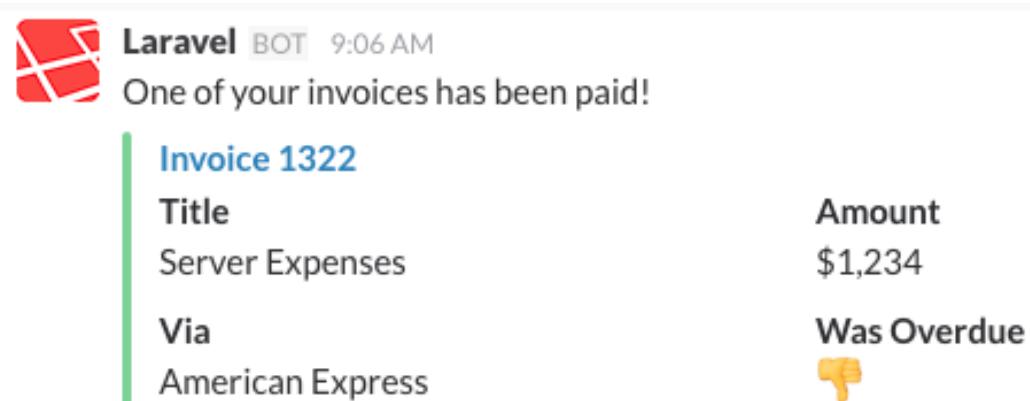
    return (new SlackMessage)
        ->success()
        ->content('One of your invoices has been paid!')
```

```

->attachment(function ($attachment) use ($url) {
    $attachment->title('Invoice 1322', $url)
        ->fields([
            'Title' => 'Server Expenses',
            'Amount' => '$1,234',
            'Via' => 'American Express',
            'Was Overdue' => '::-1:',
        ]);
});
}

```

上述代码会生成如下 Slack 消息：



Markdown 附件内容

如果一些附件字段包含 Markdown，可以使用 `markdown` 方法来构建 Slack 用以解析并显示以 Markdown 格式编写的附件字段，该方法支持的值包括 `pretext`、`text`、/ 或 `fields`。想要了解更多关于 Slack 格式化的信息，查看 [Slack API 文档](#)：

```

/**
 * Get the Slack representation of the notification.
 *
 * @param mixed $notifiable
 * @return SlackMessage
 */
public function toSlack($notifiable)
{
    $url = url('/exceptions/'.$this->exception->id);

    return (new SlackMessage)
        ->error()
        ->content('Whoops! Something went wrong.')
        ->attachment(function ($attachment) use ($url) {
            $attachment->title('Exception: File Not Found', $url)
                ->content('File [background.jpg] was *not found*.')
                ->markdown(['text']);
        });
}

```

Slack 通知路由

要路由 Slack 通知到适当的位置，需要在被通知的实体上定义一个 `routeNotificationForSlack` 方法，这将会返回通知被发送到的 Webhook URL。Webhook URL 可通过在 Slack 组上添加一个「Incoming Webhook」服务来生成：

```

<?php

namespace App;

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

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Route notifications for the Slack channel.
     *
     * @return string
     */
    public function routeNotificationForSlack()
    {
        return $this->slack_webhook_url;
    }
}

```

通知事件

当通知被发送后，通知系统会触发 `Illuminate\Notifications\Events\NotificationSent` 事件，该事件实例包含被通知的实体（如用户）和通知实例本身。你可以在 `EventServiceProvider` 中为该事件注册监听器：

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Notifications\Events\NotificationSent' => [
        'App\Listeners\LogNotification',
    ],
];
```

注：在 `EventServiceProvider` 中注册监听器之后，使用 Artisan 命令 `event:generate` 快速生成监听器类。

在事件监听器中，可以访问事件的 `notifiable`、`notification` 和 `channel` 属性了解通知接收者和通知本身的更多信息：

```
/**
 * Handle the event.
 *
 * @param NotificationSent $event
 * @return void
 */
public function handle(NotificationSent $event)
{
    // $event->channel
    // $event->notifiable
    // $event->notification
}
```

自定义通道

通过上面的介绍，可见 Laravel 为我们提供了多种通知通道，但是如果你想要编写自己的驱动以便通过其他通道发送通知，也很简单。首先定义一个包含 `send` 方法的类，该方法接收两个参数：`$notifiable` 和 `$notification`：

```
<?php

namespace App\Channels;

use Illuminate\Notifications\Notification;

class VoiceChannel
{
    /**
     * 发送给定通知.
     *
     * @param mixed $notifiable
     * @param \Illuminate\Notifications\Notification $notification
     * @return void
     */
    public function send($notifiable, Notification $notification)
    {
        $message = $notification->toVoice($notifiable);

        // 发送通知到$notifiable 实例...
    }
}
```

通知通道类被定义后，就可以在应用中通过 `via` 方法返回类名：

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use App\Channels\VoiceChannel;
use App\Channels\Messages\VoiceMessage;
use Illuminate\Notifications\Notification;
use Illuminate\Contracts\Queue\ShouldQueue;

class InvoicePaid extends Notification
```

```
{
    use Queueable;

    /**
     * 获取通知通道.
     *
     * @param mixed $notifiable
     * @return array|string
     */
    public function via($notifiable)
    {
        return [VoiceChannel::class];
    }

    /**
     * Get the voice representation of the notification.
     *
     * @param mixed $notifiable
     * @return VoiceMessage
     */
    public function toVoice($notifiable)
    {
        // ...
    }
}
```

扩展包开发

简介

扩展包是添加额外功能到 Laravel 的主要方式。扩展包可以提供任何功能，小到处理日期如 [Carbon](#)，大到整个 BDD 测试框架如 [Behat](#)。当然，有很多不同类型的扩展包。有些扩展包是独立于 Laravel 的，意味着可以在任何框架中使用，而不仅是 Laravel。比如 Carbon 和 Behat 都是独立的扩展包。所有这些扩展包都可以通过在 `composer.json` 文件中声明以便被 Laravel 使用。另一方面，其它非独立扩展包只能和 Laravel 一起使用，这些包可能有特定的路由、控制器、视图和配置用于增强 Laravel 的功能，本文档主要讨论只能在 Laravel 中使用的扩展包。

关于门面的注意点

编写 Laravel 应用时，不管你使用 [契约](#) 还是 [门面](#)，通常并没有什么关系，因为两者都提供了基本同等级别的可测试性。不过，编写扩展包时，在扩展包里不能访问所有的 Laravel 测试辅助函数。如果你想要像在扩展包中自如编写扩展包测试，就像在 Laravel 应用中一样，可以使用 [Orchestral Testbench](#) 扩展包。

包自动发现

在 Laravel 应用的配置文件 `config/app.php` 中，`providers` 配置项定义了一个会被 Laravel 加载的服务提供者列表。当安装完新的扩展包后，在老版本中需要将扩展包的服务提供者添加到这个列表以便被 Laravel 使用。从 Laravel 5.5 开始，我们不必再手动添加服务提供者到该列表，而是将提供者定义到扩展包下 `composer.json` 文件的 `extra` 选项中，除了服务提供者之外，我们还可以以这种方式注册 [门面](#)：

```
"extra": {
    "laravel": {
        "providers": [
            "Barryvdh\\Debugbar\\ServiceProvider"
        ],
        "aliases": {
            "Debugbar": "Barryvdh\\Debugbar\\Facade"
        }
    }
},
```

定义好之后，在安装扩展包之后 Laravel 就会自动注册相应的服务提供者和门面，从而为扩展包使用者提供一个更加便捷的安装体验。

选择包发现

如果你是包的使用者并且不想使用包自动发现功能，那么可以像这样在应用 `composer.json` 文件的 `section` 选项中列出包名：

```
"extra": {
    "laravel": {
        "dont-discover": [
            "barryvdh/laravel-debugbar"
        ]
    }
},
```

```
        ]
    }
},
```

你还可以在 `dont-discover` 配置项中通过通配符 `*` 禁止所有扩展包的自动发现功能：

```
"extra": {
    "laravel": {
        "dont-discover": [
            "*"
        ]
    }
},
```

服务提供者

服务提供者是扩展包和 Laravel 之间的连接纽带。服务提供者负责绑定对象到 Laravel 的**服务容器**并告知 Laravel 从哪里加载包资源如视图、配置和本地化文件。

服务提供者继承自 `Illuminate\Support\ServiceProvider` 类并包含两个方法：`register` 和 `boot`。`ServiceProvider` 基类位于 Composer 包 `illuminate/support`。要了解更多关于服务提供者的内容，查看其[文档](#)。

资源

配置

通常，需要发布扩展包配置文件到应用根目录下的 `config` 目录，这将允许扩展包使用者轻松覆盖默认配置选项，要发布一个配置文件，只需在服务提供者的 `boot` 方法中使用 `publishes` 方法即可：

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot() {
    $this->publishes([
        __DIR__.'/path/to/config/courier.php' => config_path('courier.php'),
    ]);
}
```

现在，当扩展包用户执行 Laravel 的 Artisan 命令 `vendor:publish` 时，你的文件将会被拷贝到指定位置，当然，配置被发布后，可以通过和其他配置选项一样的方式进行访问：

```
$value = config('courier.option');
```

注：不要在配置文件中定义闭包，因为当用户执行 Artisan 命令 `config:cache` 时，闭包将不能被正确序列化。

默认扩展包配置

你还可以选择将自己的扩展包配置文件合并到应用的发布副本，这允许使用者只引入他们在应用配置文件中实际想要覆盖的配置选项。要合并两个配置，在服务提供者的 `register` 方法中使用 `mergeConfigFrom` 方法即可：

```
/**
 * 在容器中注册绑定
 *
 * @return void
 */
public function register() {
    $this->mergeConfigFrom(
        __DIR__.'/path/to/config/courier.php', 'courier'
    );
}
```

注：这个方法只合并到配置数据第一维度。如果用户定义了多维配置数组，缺失的部分将不能被合并。

路由

如果扩展包包含路由，可以使用 `loadRoutesFrom` 方法加载它们，这个方法会自动判定应用的路由是否被缓存，如果路由已经被缓存将不会加载路由文件：

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->loadRoutesFrom(__DIR__.'/routes.php');
```

```
}
```

迁移

如果你的扩展包包含数据库迁移，可以使用 `loadMigrationsFrom` 方法告知 Laravel 如何加载它们。`loadMigrationsFrom` 方法接收扩展包迁移的路径作为其唯一参数：

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->loadMigrationsFrom(__DIR__.'/path/to/migrations');
}
```

扩展包迁移注册好之后，会在执行 `php artisan migrate` 时自动运行。不需要将它们导出到应用的 `database/migrations` 目录。

翻译

如果你的扩展包包含翻译文件，你可以使用 `loadTranslationsFrom` 方法告诉 Laravel 如何加载它们，例如，如果你的扩展包命名为 `courier`，应该添加如下代码到服务提供者的 `boot` 方法：

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot() {
    $this->loadTranslationsFrom(__DIR__.'/path/to/translations', 'courier');
}
```

扩展包翻译使用形如 `package::file.line` 的语法进行引用。所以，你可以使用如下方式从 `messages` 文件中加载 `courier` 包的 `welcome` 行：

```
echo trans('courier::messages.welcome');
```

发布翻译文件

如果你想要发布扩展包翻译到应用的 `resources/lang/vendor` 目录，可以使用服务提供者的 `publishes` 方法，该方法接收一个包路径和相应发布路径数组参数，例如，要发布 `courier` 扩展包的翻译文件，可以这么做：

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot() {
    $this->loadTranslationsFrom(__DIR__.'/path/to/translations', 'courier');

    $this->publishes([
        __DIR__.'/path/to/translations' => resource_path('lang/vendor/courier'),
    ]);
}
```

这样，使用者就可以执行 Artisan 命令 `vendor:publish` 将扩展包翻译文件发布到应用的指定目录。

视图

要在 Laravel 中注册扩展包视图，需要告诉 Laravel 视图在哪，可以使用服务提供者的 `loadViewsFrom` 方法来实现。`loadViewsFrom` 方法接收两个参数：视图模板的路径和扩展包名称。例如，如果你的扩展包名称是 `courier`，添加如下代码到服务提供者的 `boot` 方法即可：

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot() {
    $this->loadViewsFrom(__DIR__.'/path/to/views', 'courier');
}
```

扩展包视图通过使用形如 `package::view` 的语法来引用。所以，你可以通过如下方式加载 `courier` 扩展包上的 `admin` 视图：

```
Route::get('admin', function () {
    return view('courier::admin');
});
```

覆盖扩展包视图

当你使用 `loadViewsFrom` 方法的时候，Laravel 实际上为视图注册了两个存放位置：一个是 `resources/views/vendor` 目录，另一个是你指定的目录。所以，以 `courier` 为例：当请求一个扩展包视图时，Laravel 首先检查开发者是否在 `resources/views/vendor/courier` 提供了自定义版本的

视图，如果该视图不存在，Laravel 才会搜索你调用 `loadViewsFrom` 方法时指定的目录。这种机制使得终端用户可以轻松地自定义/覆盖扩展包视图。

发布视图

如果你想要视图能够发布到应用的 `resources/views/vendor` 目录，可以使用服务提供者的 `publishes` 方法。该方法接收包视图路径及其相应的发布路径数组作为参数：

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 * @translator laravelacademy.org
 */
public function boot(){
    $this->loadViewsFrom(__DIR__.'/path/to/views', 'courier');

    $this->publishes([
        __DIR__.'/path/to/views' => base_path('resources/views/vendor/courier'),
    ]);
}
```

现在，当使用者执行 Laravel Artisan 命令 `vendor:publish` 时，扩展包视图将会被拷贝到指定路径。

命令

要通过 Laravel 注册扩展包的 Artisan 命令，可以使用 `commands` 方法。该方法需要传入命令名称数组，注册号命令后，可以使用 `Artisan CLI` 执行它们：

```
/**
 * Bootstrap the application services.
 *
 * @return void
 */
public function boot()
{
    if ($this->app->runningInConsole()) {
        $this->commands([
            FooCommand::class,
            BarCommand::class,
        ]);
    }
}
```

前端资源

你的扩展包可能包含 JavaScript、CSS 和图片，要发布这些前端资源到应用根目录下的 `public` 目录，可以使用服务提供者的 `publishes` 方法。在本例中，我们添加一个前端资源组标签 `public`，用于发布相关的前端资源组：

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot(){
    $this->publishes([
        __DIR__.'/path/to/assets' => public_path('vendor/courier'),
    ], 'public');
}
```

现在，当使用者执行 `vendor:publish` 命令时，前端资源将会被拷贝到指定位置，由于需要在每次包更新时覆盖前端资源，可以使用 `--force` 标识：

```
php artisan vendor:publish --tag=public --force
```

发布文件组

有时候你可能想要分开发布扩展包前端资源和业务资源（配置、视图等），例如，你可能想要使用者发布扩展包配置的同时不发布前端资源，这可以通过在扩展包的服务提供者中调用 `publishes` 方法时给它们打上“标签”来实现。下面我们在扩展包服务提供者的 `boot` 方法中定义两个发布组：

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot(){
    $this->publishes([
```

```
__DIR__.'/../config/package.php' => config_path('package.php')
], 'config');

$this->publishes([
    __DIR__.'/../database/migrations/' => database_path('migrations')
], 'migrations');
}
```

现在，用户可以在使用 Artisan 命令 `vendor:publish` 时通过引用标签名来分开发布这两个组：

```
php artisan vendor:publish --tag=config
```

队列

简介

注： Laravel 现在提供了基于 Redis 的，拥有美观的后台和配置系统的 Horizon 队列扩展包，完整信息参考 [Horizon 文档](#)。

Laravel 队列为不同的后台队列服务提供了统一的 API，例如 Beanstalk，Amazon SQS，Redis，甚至其他基于关系型数据库的队列。队列的目的是将耗时的任务延时处理，比如发送邮件，从而大幅度缩短 Web 请求和响应的时间。

队列配置文件存放在 `config/queue.php`。每一种队列驱动的配置都可以在该文件中找到，包括数据库、Beanstalkd、Amazon SQS、Redis 以及同步（本地使用）驱动。其中还包含了一个 `null` 队列驱动用于那些放弃队列的任务。

连接 Vs. 队列

在开始使用 Laravel 队列以前，了解“连接”和“队列”的关系非常重要。在配置文件 `config/queue.php` 有一个 `connections` 配置项。该配置项定义了后台队列服务的特定连接，如 Amazon SQS, Beanstalk，或 Redis。每种队列连接都可以有很多队列，可以想象在银行办理现金业务的各个窗口队列。

请注意 `queue` 配置文件中的每个连接配置示例都有一个 `queue` 属性。当新的队列任务被添加到指定的连接时，该配置项的值就是默认监听的队列（名称）。换种说法，如果你没有指派特别的队列名称，那么 `queue` 的值，也是该任务默认添加到的队列（名称）：

```
// 以下的任务将被委派到默认队列...
dispatch(new Job);

// 以下任务将被委派到 "emails" 队列...
dispatch((new Job)->onQueue('emails'));
```

有些应用并不需要将任务分配到多个队列，单个队列已经非常适用。但是，应用的任务有优先级差异或者类别差异的时候，推送任务到多个队列将是更好地选择，因为 Laravel 的队列进程支持通过优先级指定处理的队列。举个例子，你可以将高优先级的任务委派到 `high` (高优先级)队列，从而让它优先执行。

```
php artisan queue:work --queue=high,default
```

驱动预备知识

数据库

要使用 `database` 队列驱动，你需要数据表保存任务信息。要生成创建这些表的迁移，可以运行 Artisan 命令 `queue:table`，迁移被创建之后，可以使用 `migrate` 命令生成这些表：

```
php artisan queue:table
php artisan migrate
```

Redis

要使用 `redis` 队列驱动，需要在配置文件 `config/database.php` 中配置 Redis 数据库连接。

Redis 集群

如果 Redis 队列连接使用 Redis Cluster (集群)，队列名称必须包含 `key hash tag`，以确保给定队列对应的所有 Redis keys 都存放到同一个 hash slot：

```
'redis' => [
    'driver' => 'redis',
    'connection' => 'default',
    'queue' => '{default}',
    'retry_after' => 90,
],
```

学院君注：对一般中小型应用推荐使用 Redis 作为队列驱动。

阻塞

使用 Redis 队列时，可以使用 `block_for` 配置项来指定驱动在迭代队列进程循环并重新轮询 Redis 数据库之前等待可用队列任务的时间。

根据队列负载来调整此配置值会比轮询 Redis 数据库来查找新任务更加高效。例如，你可以设置该值为 5 来告诉驱动在等待可用队列任务时需要阻塞五秒：

```
'redis' => [
    'driver' => 'redis',
    'connection' => 'default',
    'queue' => 'default',
    'retry_after' => 90,
    'block_for' => 5,
],
```

注：阻塞是一个实验性功能，如果 Redis 服务器或队列进程与检索队列任务同时崩溃，那么队列任务有可能会丢失。

其他驱动预备知识

如果使用以下几种队列驱动，需要安装相应的依赖：

- Amazon SQS: [aws/aws-sdk-php ~3.0](#)
- Beanstalkd: [pda/pheanstalk ~3.0](#)
- Redis: [predis/predis ~1.0](#)

创建任务

生成任务类

通常，所有的任务类都保存在 `app/Jobs` 目录。如果 `app/Jobs` 不存在，在运行 Artisan 命令 `make:job` 的时候，它将会自动创建。你可以通过 Artisan CLI 来生成队列任务类：

```
php artisan make:job ProcessPodcast
```

生成的类都实现了 `Illuminate\Contracts\Queue\ShouldQueue` 接口，告诉 Laravel 将该任务推送到队列，而不是立即运行：

```
<?php
namespace App\Jobs;

use ...;
use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;
use Illuminate\Queue\ShouldQueue;

class ProcessPodcast implements ShouldQueue
{
    /**
     * Create a new job instance.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Execute the job.
     *
     * @return void
     */
    public function handle()
    {
        //
    }
}
```

任务类结构

任务类非常简单，通常只包含处理该任务的 `handle` 方法，让我们看一个任务类的例子。在这个例子中，我们模拟管理播客发布服务，并在发布以前上传相应的播客文件：

```
<?php

namespace App\Jobs;

use App\Podcast;
use App\AudioProcessor;
use Illuminate\Bus\Queueable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;

class ProcessPodcast implements ShouldQueue
```

```

{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    protected $podcast;

    /**
     * 创建任务实例
     *
     * @param Podcast $podcast
     * @return void
     */
    public function __construct(Podcast $podcast)
    {
        $this->podcast = $podcast;
    }

    /**
     * 执行任务
     *
     * @param AudioProcessor $processor
     * @return void
     */
    public function handle(AudioProcessor $processor)
    {
        // 处理上传的播客...
    }
}

```

在本示例中，我们将 **Eloquent 模型**作为参数直接传递到构造函数。因为该任务使用了 **SerializesModels trait**，Eloquent 模型将会在任务被执行时优雅地序列化和反序列化。如果你的队列任务在构造函数中接收 Eloquent 模型，只有模型的主键会被序列化到队列，当任务真正被执行的时候，队列系统会自动从数据库中获取整个模型实例。这对应用而言是完全透明的，从而避免序列化整个 Eloquent 模型实例引起的问题。

handle 方法在任务被处理的时候调用，注意我们可以在任务的 **handle** 方法中进行依赖注入。Laravel 服务容器会自动注入这些依赖。

注：二进制数据，如原生图片内容，在传递给队列任务之前先经过 **base64_encode** 方法处理，此外，该任务被推送到队列时将不会被序列化为 JSON 格式。

分发任务

创建好任务类后，就可以通过任务自身的 **dispatch** 方法将其分发到队列。**dispatch** 方法需要的唯一参数就是该任务的实例：

```

<?php

namespace App\Http\Controllers;

use App\Jobs\ProcessPodcast;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Create podcast...

        ProcessPodcast::dispatch($podcast);
    }
}

```

延时分发

有时候你可能想要延迟队列任务的执行，这可以通过在分发任务时使用 **delay** 方法实现。例如你希望将某个任务在创建 10 分钟以后才执行：

```

<?php

namespace App\Http\Controllers;

```

```

use App\Jobs\ProcessPodcast;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Create podcast...

        ProcessPodcast::dispatch($podcast)
            ->delay(now() ->addMinutes(10));
    }
}

```

注：Amazon SQS 的队列服务最长延时 15 分钟。

任务链

任务链允许你指定一个需要在一个序列中执行的队列任务列表，如果序列中的某个任务失败，其它任务将不再运行。要执行一个队列任务链，可以使用任意可分发任务上的 `withChain` 方法：

```

ProcessPodcast::withChain([
    new OptimizePodcast,
    new ReleasePodcast
])->dispatch();

```

链接连接 & 队列

如果你想要指定任务链使用的默认连接和队列，可以使用 `allOnConnection` 和 `allOnQueue` 方法。这些方法指定队列连接和队列名称

自定义队列 & 连接

分发到指定的队列

通过推送任务到不同队列，你可以将队列任务进行“分类”，甚至根据优先级来分配每个队列的进程数。请注意，这并不意味着使用了配置项中那些不同的连接来管理队列，实际上只有单一连接会被用到。要指定队列，请在任务实例使用 `onQueue` 方法：

```

<?php

namespace App\Http\Controllers;

use App\Jobs\ProcessPodcast;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Create podcast...

        ProcessPodcast::dispatch($podcast)->onQueue('processing');
    }
}

```

分发到指定的连接

如果你使用了多个连接来管理队列，那么可以分发任务到指定的连接。请在任务实例中使用 `onConnection` 方法来指定连接：

```

<?php

namespace App\Http\Controllers;

use App\Jobs\ProcessPodcast;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

```

```
class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Create podcast...

        ProcessPodcast::dispatch($podcast)->onConnection('sqS');
    }
}
```

当然，你可以同时使用 `onConnection` 和 `onQueue` 方法来指定任务的连接和队列：

```
$job = (new ProcessPodcast($podcast))
    ->onConnection('sqS')
    ->onQueue('processing');
```

指定最大失败次数/超时时间

最大失败次数

指定队列任务最大失败次数的一种实现方式是通过 Artisan 命令 `--tries` 切换：

```
php artisan queue:work --tries=3
```

不过，你还可以在任务类自身定义最大失败次数来实现更加细粒度的控制，如果最大失败次数在任务中指定，则其优先级高于命令行指定的数值：

```
<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
{
    /**
     * The number of times the job may be attempted.
     *
     * @var int
     */
    public $tries = 5;
}
```

基于时间的尝试次数

除了定义在任务失败前的最大尝试次数外，还可以定义在指定时间内允许任务的最大尝试次数，这可以通过在任务类中添加 `retryUntil` 方法来实现：

```
/**
 * Determine the time at which the job should timeout.
 *
 * @return \DateTime
 */
public function retryUntil()
{
    return now()->addSeconds(5);
}
```

注：还可以在队列时间监听器中定义 `retryUntil` 方法。

超时

注：`timeout` 方法为 PHP 7.1+ 和 `pcntl` 扩展做了优化。

类似的，队列任务最大运行时长（秒）可以通过 Artisan 命令上的 `--timeout` 开关来指定：

```
php artisan queue:work --timeout=30
```

同样，你也可以在任务类中定义该任务允许运行的最大时长（单位：秒），任务中指定的超时时间优先级也高于命令行定义的数值：

```
<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
{
    /**
     * The number of seconds the job can run before timing out.
     */
}
```

```

*
* @var int
*/
public $timeout = 120;
}

```

频率限制

注：该功能要求应用可以与 [Redis 服务器](#) 进行交互。

如果应用使用了 Redis，那么可以使用时间或并发来控制队列任务。该功能特性在队列任务与有频率限制的 API 交互时很有帮助，例如，通过 `throttle` 方法，你可以限定给定类型任务每 60 秒只运行 10 次。如果不能获取锁，需要将任务释放回队列以便可以再次执行：

```

Redis::throttle('key')->allow(10)->every(60)->then(function () {
    // Job logic...
}, function () {
    // Could not obtain lock...

    return $this->release(10);
});

```

注：在上面的例子中，`key` 可以是任意可以唯一标识你想要限定访问频率的任务类型的字符串。举个例子，这个键可以基于任务类名和操作 Eloquent 模型的 ID 进行构建。

除此之外，还可以指定可以同时处理给定任务的最大进程数量。这个功能在队列任务正在编辑一次只能由一个任务进行处理的资源时很有用。例如，使用 `funnel` 方法你可以给定类型任务一次只能由一个工作进程进行处理：

```

Redis::funnel('key')->limit(1)->then(function () {
    // Job logic...
}, function () {
    // Could not obtain lock...

    return $this->release(10);
});

```

注：使用频率限制时，任务在运行成功之前需要的最大尝试次数很难权衡，因此，将频率限制和 [基于时间的尝试](#) 次数结合起来使用是个不错的选择。

处理错误

如果任务在处理的时候有异常抛出，则该任务将会被自动释放回队列以便再次尝试执行。任务会持续被释放直到尝试次数达到应用允许的最大次数。最大尝试次数通过 Artisan 命令 `queue:work` 上的 `--tries` 开关来定义。此外，该次数也可以在任务类自身上定义。关于运行队列监听器的更多信息可以在 [下面](#) 看到。

运行队列进程

Laravel 自带了一个队列进程用来处理被推送到队列的新任务。你可以使用 `queue:work` 命令运行这个队列进程。请注意，队列进程开始运行后，会持续监听队列，直至你手动停止或关闭终端：

```
php artisan queue:work
```

注：为了保持队列进程 `queue:work` 持续在后台运行，需要使用进程守护程序，比如 [Supervisor](#) 来确保队列进程持续运行。

请记住，队列进程是长生命周期的进程，会在启动后驻留内存。若应用有任何改动将不会影响到已经启动的进程。所以请在发布程序后，[重启队列进程](#)。

处理单个任务

`--once` 选项可用于告知进程只处理队列中的单个任务：

```
php artisan queue:work --once
```

指定连接和队列

队列进程同样可以自定义连接和队列。传递给 `work` 命令的连接名需要与配置文件 `config/queue.php` 中定义的某个连接配置相匹配：

```
php artisan queue:work redis
```

你可以自定义将某个队列进程指定某个连接来管理。举例来说，如果所有的邮件任务都是通过 `redis` 连接上的 `emails` 队列处理，那么可以用以下命令来启动单一进程只处理单一队列：

```
php artisan queue:work redis --queue=emails
```

资源注意事项

后台队列进程不会再处理每个任务前重启框架，因此你需要在每次任务完成后释放所有重量级的资源。例如，如果你在使用 GD 库处理图片，需要在完成的时候使用 `imagedestroy` 来释放内存。

队列优先级

有时候你需要区分任务的优先级。比如，在配置文件 `config/queue.php` 中，你可以定义 `redis` 连接的默认 `queue` 为 `low`。不过，如果需要将任务分发到高优先级 `high`，可以这么做：

```
dispatch((new Job)->onQueue('high'));
```

如果期望所有 `high` 高优先级的队列都将先于 `low` 低优先级的任务执行，可以像这样启动队列进程：

```
php artisan queue:work --queue=high,low
```

队列进程 & 部署

前文已经提到队列进程是长生命周期的进程，在重启以前，所有源码的修改并不会对其产生影响。所以，最简单的方法是在每次发布新版本后重新启动队列进程。你可以通过 `Aritsan` 命令 `queue:restart` 来优雅地重启队列进程：

```
php artisan queue:restart
```

该命令将在队列进程完成正在进行的任务后，结束该进程，避免队列任务的丢失或错误。由于队列进程会在执行 `queue:restart` 命令后死掉，你仍然需要通过进程守护程序如 `Supervisor` 来自动重启队列进程。

注：队列使用 `缓存` 来存储重启信号，所以在使用此功能前你需要验证缓存驱动配置正确。

任务过期 & 超时

任务过期

在配置文件 `config/queue.php` 中，每个连接都定义了 `retry_after` 项。该配置项的目的是定义任务在执行以后多少秒后释放回队列。如果 `retry_after` 设定的值为 `90`，任务在运行 `90` 秒后还未完成，那么将被释放回队列而不是删除掉。毫无疑问，你需要把 `retry_after` 的值设定为任务执行时间的最大可能值。

注：只有 Amazon SQS 配置信息不包含 `retry_after` 项。Amazon SQS 的任务执行时间基于 `Default Visibility Timeout`，该项在 Amazon AWS 控制台配置。

队列进程超时

队列进程 `queue:work` 可以设定超时 `--timeout` 项。该 `--timeout` 控制队列进程执行每个任务的最长时间，如果超时，该进程将被关闭。各种错误都可能导致某个任务处于“冻结”状态，比如 HTTP 无响应等。队列进程超时就是为了将这些“冻结”的进程关闭：

```
php artisan queue:work --timeout=60
```

配置项 `retry_after` 和 `Aritsan` 参数项 `--timeout` 不同，但目的都是为了确保任务的安全，并且只被成功的执行一次。

注：参数项 `--timeout` 的值应该始终小于配置项 `retry_after` 的值，这是为了确保队列进程总在任务重试以前关闭。如果 `--timeout` 比 `retry_after` 大，那么你的任务可能被执行两次。

进程休眠时间

当任务在队列中有效时，进程会持续处理任务，没有延迟。不过，我们可以使用 `sleep` 配置项来指定没有新的有效任务产生时的休眠时间：

```
php artisan queue:work --sleep=3
```

配置 Supervisor

安装 Supervisor

`Supervisor` 是 Linux 系统中常用的进程守护程序。如果队列进程 `queue:work` 意外关闭，它会自动重启启动队列进程。在 Ubuntu 安装 `Supervisor` 非常简单：

```
sudo apt-get install supervisor
```

注：如果自己配置 `Supervisor` 有困难，可以考虑使用 `Laravel Forge`，它会为 `Laravel` 项目自动安装并配置 `Supervisor`。

配置 Supervisor

`Supervisor` 配置文件通常存放在 `/etc/supervisor/conf.d` 目录，在该目录下，可以创建多个配置文件指示 `Supervisor` 如何监视进程，例如，让我们创建一个开启并监视 `queue:work` 进程的 `laravel-worker.conf` 文件：

```
[program:laravel-worker]
process_name=%(program_name)s_%(process_num)02d
command=php /home/forge/app.com/artisan queue:work sqs --sleep=3 --tries=3
autostart=true
autorestart=true
user=forge
numprocs=8
redirect_stderr=true
stdout_logfile=/home/forge/app.com/worker.log
```

在本例中，`numprocs` 指令让 `Supervisor` 运行 8 个 `queue:work` 进程并监视它们，如果失败的话自动重启。当然，你需要修改 `queue:work` `sqs` 的 `command` 指令来映射你的队列连接。

启动 Supervisor

当成功创建配置文件后，需要刷新 `Supervisor` 的配置信息并使用如下命令启动进程：

```
sudo supervisorctl reread
sudo supervisorctl update
sudo supervisorctl start laravel-worker:*
```

你可以通过 `Supervisor` 官方文档获取更多信息。

处理失败的任务

不可避免会出现运行失败的任务。你不必为此担心，`Laravel` 可以轻松设置任务允许的最大尝试次数，若是执行次数达到该限定，该任务会被插入到 `failed_jobs` 表，要创建一个 `failed_jobs` 表的迁移，可以使用 `queue:failed-table` 命令

```
php artisan queue:failed-table
php artisan migrate
```

然后，运行队列进程时，通过 `--tries` 参数项来设置队列任务允许的最大尝试次数，如果没有指定 `--tries` 选项的值，任务会被无限期重试：

```
php artisan queue:work redis --tries=3
```

清理失败的任务

你可以在任务类中定义 `failed` 方法，从而允许你在失败发生时执行指定的动作，比如发送任务失败的通知，记录日志等。导致任务失败的 `Exception` 会被传递到 `failed` 方法：

```
<?php

namespace App\Jobs;

use Exception;
use App\Podcast;
use App\AudioProcessor;
use Illuminate\Bus\Queueable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class ProcessPodcast implements ShouldQueue
{
    use InteractsWithQueue, Queueable, SerializesModels;

    protected $podcast;

    /**
     * Create a new job instance.
     *
     * @param Podcast $podcast
     * @return void
     */
    public function __construct(Podcast $podcast)
    {
        $this->podcast = $podcast;
    }

    /**
     * Execute the job.
     *
     * @param AudioProcessor $processor
     * @return void
     */
    public function handle(AudioProcessor $processor)
    {
        // Process uploaded podcast...
    }

    /**
     * The job failed to process.
     *
     * @param Exception $exception
     * @return void
     */
    public function failed(Exception $exception)
    {
        // 发送失败通知, etc...
    }
}
```

任务失败事件

如果你期望在任务失败的时候触发某个事件，可以使用 `Queue::failing` 方法。该事件通过邮件或 HipChat 通知团队。举个例子，我么可以在 Laravel 自带的 `AppServiceProvider` 中添加一个回调到该事件：

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Queue;
use Illuminate\Queue\Events\JobFailed;
use Illuminate\Support\ServiceProvider;
```

```

class AppServiceProvider extends ServiceProvider
{
    /**
     * 启动应用服务.
     *
     * @return void
     */
    public function boot()
    {
        Queue::failing(function (JobFailed $event) {
            // $event->connectionName
            // $event->job
            // $event->exception
        });
    }

    /**
     * 注册服务提供者.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}

```

重试失败的任务

要查看已插入到 `failed_jobs` 数据表中的所有失败任务，可以使用 Artisan 命令 `queue:failed`:

```
php artisan queue:failed
```

该命令将会列出任务 ID、连接、队列和失败时间，任务 ID 可用于重试失败任务，例如，要重试一个 ID 为 5 的失败任务，可以运行下面的命令:

```
php artisan queue:retry 5
```

要重试所有失败任务，运行如下命令即可:

```
php artisan queue:retry all
```

如果你要删除一个失败任务，可以使用 `queue:forget` 命令:

```
php artisan queue:forget 5
```

要删除所有失败任务，可以使用 `queue:flush` 命令:

```
php artisan queue:flush
```

任务事件

通过 `Queue` 门面提供的 `before` 和 `after` 方法可以在任务被处理之前或之后指定要执行的回调。这些回调可用来记录日志或者记录统计数据。通常，你可以在 [服务提供者](#) 中使用这些方法。比如，我们可能在 `AppServiceProvider` 这样用:

```

<?php

namespace App\Providers;

use Illuminate\Support\Facades\Queue;
use Illuminate\Support\ServiceProvider;
use Illuminate\Queue\Events\JobProcessed;
use Illuminate\Queue\Events\JobProcessing;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Queue::before(function (JobProcessing $event) {

```

```

    // $event->connectionName
    // $event->job
    // $event->job->payload()
}) ;

Queue::after(function (JobProcessed $event) {
    // $event->connectionName
    // $event->job
    // $event->job->payload()
}) ;
}

/**
 * Register the service provider.
 *
 * @return void
 */
public function register()
{
    //
}
}

```

使用 `Queue` 门面上的 `looping` 方法，你可以在进程尝试从队列中获取任务之前指定要执行的回调。例如，你可以注册一个闭包来回滚之前失败任务遗留下来的事物：

```

Queue::looping(function () {
    while (DB::transactionLevel() > 0) {
        DB::rollBack();
    }
});

```

任务调度

简介

Cron 是 UNIX、SOLARIS、LINUX 下的一个十分有用的工具，通过 Cron 脚本能使计划任务定期地在系统后台自动运行。这种计划任务在 UNIX、SOLARIS、LINUX 下术语为 Cron Jobs。Crontab 则是用来记录在特定时间运行的 Cron 的一个脚本文件，Crontab 文件的每一行均遵守特定的格式：



我们可以在服务器上通过 `crontab -e` 来新增或编辑 Cron 条目，通过 `crontab -l` 查看已存在的 Cron 条目。更多关于 Cron 的原理和使用细节请自行百度或 Google。

在以前，开发者需要为每一个需要调度的任务编写一个 Cron 条目，这是很让人头疼的事。你的任务调度不在源码控制中，你必须使用 SSH 登录到服务器然后添加这些 Cron 条目。

Laravel 命令调度器允许你流式而又不失优雅地在 Laravel 中定义命令调度，并且服务器上只需要一个 Cron 条目即可。任务调度定义在 `app/Console/Kernel.php` 文件的 `schedule` 方法中，该方法中已经包含了一个示例。

开启调度器

下面是你唯一需要添加到服务器的 Cron 条目，如果你不知道如何添加 Cron 条目到服务器，可以考虑使用诸如 [Laravel Forge](#) 这样的服务来为管理 Cron 条目：

```
* * * * * php /path-to-your-project/artisan schedule:run >> /dev/null 2>&1
```

该 Cron 将会每分钟调用一次 Laravel 命令调度器，当 `schedule:run` 命令执行后，Laravel 评估你的调度任务并运行到期的任务。

定义调度

你可以在 `App\Console\Kernel` 类的 `schedule` 方法中定义所有调度任务。让我们从一个调度任务的例子开始，在这个例子中，我们将会在每天午夜调度一个被调用的闭包。在这个闭包中我们将会执行一个数据库操作来清空表：

```
<?php

namespace App\Console;

use DB;
use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;

class Kernel extends ConsoleKernel{
    /**
     * 应用提供的 Artisan 命令
     *
     * @var array
     */
    protected $commands = [
        //
    ];

    /**
     * 定义应用的命令调度
     *
     * @param \Illuminate\Console\Scheduling\Schedule $schedule
     * @return void
     * @translator laravelacademy.org
     */
    protected function schedule(Schedule $schedule)
    {
        $schedule->call(function () {
            DB::table('recent_users')->delete();
        })->daily();
    }
}
```

调度 Artisan 命令

除了调度闭包调用外，还可以调度 [Artisan 命令](#) 和操作系统命令。例如，可以使用 `command` 方法通过命令名或类来调度一个 Artisan 命令：

```
$schedule->command('emails:send --force')->daily();
$schedule->command(EmailsCommand::class, ['--force'])->daily();
```

调度队列任务

`job` 方法可用于调度一个[队列任务](#)，通过该方法可以很方便地调度任务而不必调用 `call` 方法手动创建闭包来推送任务到队列：

```
$schedule->job(new Heartbeat)->everyFiveMinutes();
```

调度 Shell 命令

`exec` 方法可用于调用操作系统命令：

```
$schedule->exec('node /home/forge/script.js')->daily();
```

调度常用选项

当然，你可以分配多种调度到任务：

方法	描述
<code>->cron('* * * * *');</code>	在自定义 Cron 调度上运行任务
<code>->everyMinute();</code>	每分钟运行一次任务
<code>->everyFiveMinutes();</code>	每五分钟运行一次任务
<code>->everyTenMinutes();</code>	每十分钟运行一次任务
<code>->everyFifteenMinutes();</code>	每十五分钟运行一次任务
<code>->everyThirtyMinutes();</code>	每三十分钟运行一次任务
<code>->hourly();</code>	每小时运行一次任务
<code>->hourlyAt(17);</code>	每小时第十七分钟运行一次任务
<code>->daily();</code>	每天凌晨零点运行任务
<code>->dailyAt('13:00');</code>	每天 13:00 运行任务
<code>->twiceDaily(1, 13);</code>	每天 1:00 & 13:00 运行任务
<code>->weekly();</code>	每周运行一次任务
<code>->monthly();</code>	每月运行一次任务
<code>->monthlyOn(4, '15:00');</code>	每月 4 号 15:00 运行一次任务
<code>->quarterly();</code>	每个季度运行一次
<code>->yearly();</code>	每年运行一次
<code>->timezone('America/New_York');</code>	设置时区

这些方法可以和额外的约束一起联合起来创建一周特定时间运行的、更加细粒度的调度，例如，要在每周一调度一个命令：

```
$schedule->call(function () {
    // 每周星期一 13:00 运行一次...
})->weekly()->mondays()->at('13:00');

// 工作日的上午 8 点到下午 5 点每小时运行...
$schedule->command('foo')
    ->weekdays()
    ->hourly()
    ->timezone('America/Chicago')
    ->between('8:00', '17:00');
```

下面是额外的调度约束列表：

方法	描述
<code>->weekdays();</code>	只在工作日运行任务
<code>->sundays();</code>	每个星期天运行任务
<code>->mondays();</code>	每个星期一运行任务
<code>->tuesdays();</code>	每个星期二运行任务
<code>->wednesdays();</code>	每个星期三运行任务
<code>->thursdays();</code>	每个星期四运行任务
<code>->fridays();</code>	每个星期五运行任务
<code>->saturdays();</code>	每个星期六运行任务
<code>->between(\$start, \$end);</code>	基于特定时间段运行任务

方法	描述
<code>->when(Closure);</code>	基于特定测试运行任务

介于时间的约束条件

`between` 方法用于限定一天中特定时间段的任务执行:

```
$schedule->command('reminders:send')
    ->hourly()
    ->between('7:00', '22:00');
```

类似地, `unlessBetween` 方法用于排除指定时间段任务的执行:

```
$schedule->command('reminders:send')
    ->hourly()
    ->unlessBetween('23:00', '4:00');
```

真理测试的约束条件

`when` 方法用于限制任务基于给定真理测试的结果执行。换句话说, 如果给定闭包返回 `true`, 只要没有其它约束条件阻止任务运行, 该任务就会执行:

```
$schedule->command('emails:send')->daily()->when(function () {
    return true;
});
```

`skip` 方法和 `when` 相反, 如果 `skip` 方法返回 `true`, 调度任务将不会执行:

```
$schedule->command('emails:send')->daily()->skip(function () {
    return true;
});
```

使用 `when` 方法链的时候, 调度命令将只会执行返回 `true` 的 `when` 方法。

时区

使用 `timezone` 方法你可以指定调度任务的执行时间在给定时区内切换:

```
$schedule->command('report:generate')
    ->timezone('America/New_York')
    ->at('02:00')
```

注: 请记住有些时区会使用夏令时, 当夏令时改变时, 你的调度任务有可能会运行两次或者根本不会运行, 因此, 建议你最好不要使用这种时区调度。

避免任务重叠

默认情况下, 即使前一个任务仍然在运行调度任务也会运行, 要避免这样的情况, 可使用 `withoutOverlapping` 方法:

```
$schedule->command('emails:send')->withoutOverlapping();
```

在本例中, Artisan 命令 `emails:send` 每分钟都会运行 —— 如果该命令没有在运行的话。如果你的任务在执行时经常大幅度的变化, 那么 `withoutOverlapping` 方法就非常有用, 你不必再去预测给定任务到底要消耗多长时间。

如果需要的话, 你可以指定“without overlapping”锁失效前的分钟数, 默认情况下, 这个锁会在 24 小时后失效:

```
$schedule->command('emails:send')->withoutOverlapping(10);
```

在单台服务器上运行任务

注: 要使用这个功能, 必须使用 `memcached` 或 `redis` 缓存驱动作为应用默认的缓存驱动。此外, 所有服务器必须和同一个中央缓存服务器通信。如果你的应用运行在多台服务器上, 可能需要限制调度任务只在某台服务器上运行。例如, 假设你有一个每个星期五晚上生成新报告的调度任务, 如果任务调度器运行在三台服务器上, 调度任务会在三台服务器上运行并且生成三次报告, 不够优雅!

要告知任务只在单台服务器上运行, 在定义调度任务时使用 `onOneServer` 方法即可。第一台获取到该任务的服务器会给任务上一把原子锁以阻止其他服务器同时运行该任务:

```
$schedule->command('report:generate')
    ->fridays()
    ->at('17:00')
    ->onOneServer();
```

维护模式

当 Laravel 处于 `维护模式` 时, 调度任务不会运行, 不过, 如果你想要在维护模式期间强制运行任务, 可以使用 `evenInMaintenanceMode` 方法:

```
$schedule->command('emails:send')->evenInMaintenanceMode();
```

任务输出

Laravel 调度器为处理调度任务输出提供了多个方便的方法。首先, 使用 `sendOutputTo` 方法, 你可以发送输出到文件以便稍后检查:

```
$schedule->command('emails:send')
    ->daily()
    ->sendOutputTo($filePath);
```

如果你想要追加输出到给定文件，可以使用 `appendOutputTo` 方法：

```
$schedule->command('emails:send')
    ->daily()
    ->appendOutputTo($filePath);
```

使用 `emailOutputTo` 方法，你可以将输出通过邮件发送给接收人。使用邮件发送任务输出之前，需要配置 Laravel 的邮件服务：

```
$schedule->command('foo')
    ->daily()
    ->sendOutputTo($filePath)
    ->emailOutputTo('foo@example.com');
```

注：`emailOutputTo`、`sendOutputTo` 和 `appendOutputTo` 方法只对 `command` 和 `exec` 方法有效。

任务钩子

使用 `before` 和 `after` 方法，你可以指定在调度任务完成之前和之后要执行的代码：

```
$schedule->command('emails:send')
    ->daily()
    ->before(function () {
        // 任务即将开始...
    })
    ->after(function () {
        // 任务已经完成...
    });
});
```

Ping URL

使用 `pingBefore` 和 `thenPing` 方法，调度器可以在任务完成之前和之后自动 ping 给定的 URL。该方法在通知外部服务时很有用，例如 [Laravel Envoyer](#)，在调度任务开始或完成的时候：

```
$schedule->command('emails:send')
    ->daily()
    ->pingBefore($url)
    ->thenPing($url);
```

使用 `pingBefore($url)` 或 `thenPing($url)` 特性需要安装 HTTP 库 Guzzle，可以使用 Composer 包管理器来安装 Guzzle 依赖到项目：

```
composer require guzzlehttp/guzzle
```

十、测试系列

快速入门

简介

Laravel 植根于测试，实际上，内置使 PHPUnit 对测试提供支持是开箱即用的，并且 `phpunit.xml` 文件已经为应用设置好了。框架还提供了方便的辅助方法允许你对应用进行优雅的测试。

默认情况下，`tests` 目录包含了两个子目录：`Feature` 和 `Unit`，分别用于功能测试和单元测试，单元测试专注于小的、相互隔离的代码，实际上，大部分单元测试可能都是聚焦于单个方法。功能测试可用于测试较大区块的代码，包括若干组件之前的交互，甚至一个完整的 HTTP 请求。

`Feature` 和 `Unit` 测试目录下都提供了 `ExampleTest.php` 文件，安装完新的 Laravel 应用后，只需在项目根目录下简单运行 `phpunit` 即可运行测试（如果提示找不到命令，可以通过 `cp vendor/bin/phpunit .` 将命令拷贝过来）：

```
localhost:laravel55 sunqiang$ ./phpunit
PHPUnit 6.3.0 by Sebastian Bergmann and contributors.

..
2 / 2 (100%)

Time: 622 ms, Memory: 14.00MB

OK (2 tests, 2 assertions)
```

注：PHPUnit 是一个面向程序员的、功能强大的 PHP 单元测试框架，如果你之前没接触过 PHPUnit，可以通过[官网](#)及[中文文档](#)快速入门。

环境

运行测试的时候，Laravel 会自动设置环境为 `testing`，这是因为 `phpunit.xml` 中定义了环境变量。Laravel 在测试时还会自动配置 Session 和缓存驱动为 `array`，这意味着测试时不会持久化存储会话和缓存。

如果需要的话，你也可以定义其它测试环境配置值。`testing` 环境变量可以在 `phpunit.xml` 文件中配置，但是要确保在运行命令之前使用 Artisan 命令 `config:clear` 清除配置缓存。

此外，你还可以在项目根目录下创建一个 `.env.testing` 文件，该文件会在运行 PHPUnit 测试或执行带 `--env=testing` 开关的 Artisan 命令时覆盖 `.env` 文件中的环境变量。

创建 & 运行测试

要创建一个新的测试用例，可以使用 Artisan 命令 `make:test`：

```
// 在 Feature 目录下创建测试类...
php artisan make:test UserTest

// 在 Unit 目录下创建测试类...
php artisan make:test UserTest --unit
```

创建完测试类后，就可以像使用 PHPUnit 一样定义测试方法，要运行测试，只需在终端执行 `phpunit` 命令即可：

```
<?php

namespace Tests\Unit;

use Tests\TestCase;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    /**
     * 简单测试示例
     *
     * @return void
     */
    public function testBasicTest()
    {
        $this->assertTrue(true);
    }
}
```

注：如果在测试类中定义了自己的 `setUp` 方法，确保在该方法中调用了 `parent::setUp()`。

HTTP 测试

简介

Laravel 为生成 HTTP 请求、测试输出提供了流式 API。举个例子，我们 Laravel 提供的测试示例：

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function testBasicTest()
    {
        $response = $this->get('/');

        $response->assertStatus(200);
    }
}
```

}

`get` 方法生成了一个 `GET` 请求，而 `assertStatus` 方法断言返回的响应应该包含给定的 HTTP 状态码。除了这个简单的断言之外，Laravel 还包含检查响应头、响应内容、响应 JSON 结构等多种断言。

自定义请求头

你可以通过 `withHeaders` 方法在请求发送给应用之前自定义请求头。你可以添加任意自定义请求头到请求实例：

```
<?php

class ExampleTest extends TestCase
{
    /**
     * 简单的功能测试示例
     *
     * @return void
     */
    public function testBasicExample()
    {
        $response = $this->withHeaders([
            'X-Header' => 'LaravelAcademy',
        ])->json('POST', '/user', ['name' => '学院君']);

        $response
            ->assertStatus(200)
            ->assertJson([
                'created' => true,
            ]);
    }
}
```

注：运行测试时，CSRF 中间件会自动被禁止。

会话 / 认证

Laravel 提供了多个辅助函数用于在 HTTP 测试期间处理会话（Session），首先，你可以使用 `withSession` 方法来设置会话数据。这对于在发起请求之前加载会话数据很有用：

```
<?php

class ExampleTest extends TestCase
{
    public function testApplication()
    {
        $response = $this->withSession(['foo' => 'bar'])
            ->get('/');
    }
}
```

当然，会话最常见的用途还是维护认证用户的状态。对此，辅助函数 `actionAs` 方法提供了一个简单的方式来认证当前用户，例如，我们可以使用 [模型工厂](#) 来生成并认证用户：

```
<?php

use App\User;

class ExampleTest extends TestCase
{
    public function testApplication()
    {
        $user = factory(User::class)->create();

        $response = $this->actingAs($user)
            ->withSession(['foo' => 'bar'])
            ->get('/');
    }
}
```

你还可以通过传递 `guard` 名作为 `actionAs` 方法的第二个参数来指定使用哪一个 `guard` 来认证给定用户：

```
$this->actingAs($user, 'api');
```

测试 JSON API

Laravel 还提供了多个辅助函数用于测试 JSON API 及其响应。例如，`json`、`get`、`post`、`put`、`patch` 和 `delete` 方法用于通过多种 HTTP 请求方式发出请求。你还可以轻松传递数据和请求头到这些方法。作为开始，我们编写测试来生成 `POST` 请求到 `/user` 并断言返回的数据是否是我们所期望的：

```
<?php

class ExampleTest extends TestCase
{
    /**
     * 基本功能测试示例.
     *
     * @return void
     */
    public function testBasicExample()
    {

        $response = $this->json('POST', '/user', ['name' => '学院君']);

        $response
            ->assertStatus(200)
            ->assertJson([
                'created' => true,
            ]);
    }
}
```

注：`assertJson` 方法将响应转化为数组并使用 `PHPUnit::assertArraySubset` 验证给定数组在应用返回的 JSON 响应中是否存在。所以，如果在 JSON 响应中存在其它属性，这个测试仍然会通过，只要给定的片段存在即可。

验证完全匹配

如果你想要验证给定数组和应用返回的 JSON 能够完全匹配，可以使用 `assertExactJson` 方法：

```
<?php

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {

        $response = $this->json('POST', '/user', ['name' => '学院君']);

        $response
            ->assertStatus(200)
            ->assertExactJson([
                'created' => true,
            ]);
    }
}
```

测试文件上传

`Illuminate\Http\UploadedFile` 类提供了一个 `fake` 方法用于生成假文件或图片进行测试。这一机制和 `Storage` 门面的 `fake` 方法联合在一起，极大地简化了文件上传的测试。例如，你可以联合这两个特性来轻松测试头像上传表单：

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Http\UploadedFile;
use Illuminate\Support\Facades\Storage;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;
```

```
class ExampleTest extends TestCase
{
    public function testAvatarUpload()
    {
        Storage::fake('avatars');

        $response = $this->json('POST', '/avatar', [
            'avatar' => UploadedFile::fake()->image('avatar.jpg')
        ]);

        // Assert the file was stored...
        Storage::disk('avatars')->assertExists('avatar.jpg');

        // Assert a file does not exist...
        Storage::disk('avatars')->assertMissing('missing.jpg');
    }
}
```

伪造文件自定义

使用 `fake` 方法创建文件的时候，你可以指定宽度、高度、以及图片的尺寸以便更好的测试验证规则：

```
UploadedFile::fake()->image('avatar.jpg', $width, $height)->size(100);
```

除了创建图片之外，你还可以使用 `create` 方法创建其它类型的文件：

```
UploadedFile::fake()->create('document.pdf', $sizeInKilobytes);
```

有效的断言方法

响应断言

Laravel 为 PHPUnit 测试提供了多个自定义的断言方法。这些断言可以通过测试方法 `json`、`get`、`post`、`put` 和 `delete` 返回的响应进行访问：

assertCookie

断言响应包含给定 Cookie：

```
$response->assertCookie($cookieName, $value = null);
```

assertCookieExpired

断言响应包含给定 Cookie 并且已过期：

```
$response->assertCookieExpired($cookieName);
```

assertCookieMissing

断言响应不包含给定 Cookie：

```
$response->assertCookieMissing($cookieName);
```

assertDontSee

断言给定字符串不在响应中：

```
$response->assertDontSee($value);
```

assertDontSeeText

断言给定字符串不在响应文本中：

```
$response->assertDontSeeText($value);
```

assertExactJson

断言响应与给定 JSON 数据完全匹配：

```
$response->assertExactJson(array $data);
```

assertHeader

断言给定头在响应中是否存在：

```
$response->assertHeader($headerName, $value = null);
```

assertHeaderMissing

断言给定头在响应中不存在：

```
$response->assertHeaderMissing($headerName);
```

assertJson

断言响应包含给定 JSON 数据：

```
$response->assertJson(array $data);
```

assertJsonFragment

断言响应中包含给定 JSON 片段:

```
$response->assertJsonFragment(array $data);
```

assertJsonMissing

断言响应中不包含给定 JSON 片段:

```
$response->assertJsonMissing(array $data);
```

assertJsonMissingExact

断言响应中不包含给定完整的 JSON 片段:

```
$response->assertJsonMissingExact(array $data);
```

assertJsonStructure

断言响应包含给定 JSON 结构:

```
$response->assertJsonStructure(array $structure);
```

assertJsonValidationErrors

断言响应包含给定键的 JSON 验证错误信息:

```
$response->assertJsonValidationErrors($keys);
```

assertPlainCookie

断言响应包含给定 Cookie (未加密) :

```
$response->assertPlainCookie($cookieName, $value = null);
```

assertRedirect

断言响应重定向到给定 URI:

```
$response->assertRedirect($uri);
```

assertSee

断言给定字符串包含在响应中:

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

assertSeeInOrder

断言给定字符串有序包含在响应中:

```
$response->assertSeeInOrder(array $values);
```

assertSeeText

断言给定字符串包含在响应文本中:

```
$response->assertSeeText($value);
```

assertSeeTextInOrder

断言给定字符串有序包含在响应文本中:

```
$response->assertSeeTextInOrder(array $values);
```

assertSessionHas

断言会话中包含给定数据片段:

```
$response->assertSessionHas($key, $value = null);
```

assertSessionHasAll

断言会话中存在给定值列表:

```
$response->assertSessionHasAll($key, $value = null);
```

assertSessionHasErrors

断言会话中包含给定字段的错误信息:

```
$response->assertSessionHasErrors(array $keys, $format = null, $errorBag = 'default');
```

assertSessionHasErrorsIn

断言会话中包含给定错误:

```
$response->assertSessionHasErrorsIn($errorBag, $keys = [], $format = null);
```

assertSessionMissing

断言会话中不包含给定键:

```
$response->assertSessionMissing($key);
```

assertStatus

断言响应中包含给定状态码:

```
$response->assertStatus($code);
```

assertSuccessful

断言响应中包含成功状态码:

```
$response->assertSuccessful();
```

assertViewHas

断言响应视图包含给定数据片段:

```
$response->assertViewHas($key, $value = null);
```

assertViewHasAll

断言响应视图包含给定数据列表:

```
$response->assertViewHasAll(array $data);
```

assertViewIs

断言给定视图由该路由返回:

```
$response->assertViewIs($value);
```

assertViewMissing

断言响应视图不包含绑定数据片段:

```
$response->assertViewMissing($key);
```

认证断言

Laravel 还为 PHPUnit 测试提供了一些认证相关的断言:

方法	描述
<code>\$this->assertAuthenticated(\$guard = null);</code>	断言当前用户已认证
<code>\$this->assertGuest(\$guard = null);</code>	断言当前用户未认证
<code>\$this->assertAuthenticatedAs(\$user, \$guard = null);</code>	断言给定用户已认证
<code>\$this->assertCredentials(array \$credentials, \$guard = null);</code>	断言给定认证信息有效
<code>\$this->assertInvalidCredentials(array \$credentials, \$guard = null);</code>	断言给定认证信息无效

浏览器测试

简介

Laravel Dusk 提供了优雅的、易于使用的浏览器自动测试 API。默认情况下，Dusk 不强制你在机器上安装 JDK 或 Selenium，取而代之地，Dusk 基于独立安装的 [ChromeDriver](#)。不过，你可以使用任意其他兼容 Selenium 的驱动。

注: [Selenium](#) 是一套 Web 应用自动化测试系统，运行 Selenium 测试就像你在浏览器中操作一样；WebDriver 提供更精简的编程接口，以解决 Selenium-RC API 中的一些限制，WebDriver 的目标是提供一套精心设计的、面向对象的 API 来更好地支持现代高级 Web 应用的测试工作。Selenium 2.0 开始就已经集成了 WebDriver API，不过你仍然可以独立使用 WebDriver API，ChromeDriver 由 Chromium 团队维护，是基于 Chromium（含 Chrome）浏览器提供的、实现了 [WebDriver wire 协议](#) 的独立服务器，安装完 Dusk 扩展包会自带 ChromeDriver，无需手动安装。

安装

开始之前，需要添加 Composer 依赖 [laravel/dusk](#) 到项目:

```
composer require --dev laravel/dusk:^2.0
```

安装完 Dusk 后，需要注册服务提供者 [Laravel\Browser\BrowserServiceProvider](#)，通常，这会通过 Laravel 的自动服务提供者注册服务自动完成。

注: 如果你是手动注册 Dusk 服务提供者，永远不要将其注册到生产环境，这会导致所有用户都可以登录到应用。

接下来，运行 Artisan 命令 `dusk:install`:

```
php artisan dusk:install
```

运行完该命令后，会在 `tests` 目录下新建一个 `Browser` 目录并包含一个测试示例。接下来，在 `.env` 文件中设置环境变量 `APP_URL`，该变量值需要和你在浏览器中用于访问应用的 URL 相匹配。

要运行测试，使用 Artisan 命令 `dusk`。`dusk` 命令接收任意 `phpunit` 支持的参数:

```
php artisan dusk
```

```
localhost:laravel55 sunqiang$ php artisan dusk
PHPUnit 6.3.0 by Sebastian Bergmann and contributors.

.

1 / 1 (100%)

Time: 3.94 seconds, Memory: 14.00MB

OK (1 test, 1 assertion)
```

使用其它浏览器

默认情况下，Dusk 使用 Google Chrome 和独立安装的 `ChromeDriver` 来运行浏览器测试（Dusk 扩展包自带），不过，你也可以启动自己的 Selenium 服务器并在浏览器中进行测试。

开始之前，打开 `tests/DuskTestCase.php` 文件，该文件是应用的 Dusk 测试用例基类。在该文件中，你可以移除对 `startChromeDriver` 方法的调用，这样，就可以阻止 Dusk 自动启动 `ChromeDriver`：

```
/**
 * Prepare for Dusk test execution.
 *
 * @beforeClass
 * @return void
 */
public static function prepare()
{
    // static::startChromeDriver();
}
```

接下来，可以编辑 `driver` 方法连接到你所选择的 URL 和接口，此外，你可以编辑需要传递给 WebDriver 的“期望功能”：

```
/**
 * 创建 RemoteWebDriver 实例。
 *
 * @return \Facebook\WebDriver\Remote\RemoteWebDriver
 */
protected function driver()
{
    return RemoteWebDriver::create(
        'http://localhost:4444/wd/hub', DesiredCapabilities::phantomjs()
    );
}
```

快速入门

生成测试

要生成一个 Dusk 测试，使用 Artisan 命令 `dusk:make`，生成的测试位于 `tests/Browser` 目录：

```
php artisan dusk:make LoginTest
```

运行测试

要运行浏览器测试，使用 Artisan 命令 `dusk`：

```
php artisan dusk
```

`dusk` 命令接收任意 PHPUnit 支持的参数，所以你可以为给定 `group` 运行测试：

```
php artisan dusk --group=foo
```

手动启动 `ChromeDriver`

默认情况下，Dusk 会自动尝试启动 `ChromeDriver`，如果在你的系统上不生效，你可以在运行 `dusk` 命令前手动启动 `ChromeDriver`。如果你选择了手动启动 `ChromeDriver`，则需要注释掉 `tests/DuskTestCase.php` 文件中的如下这行代码：

```
/**
 * Prepare for Dusk test execution.
 *
 * @beforeClass
 * @return void
 */
public static function prepare()
{
    // static::startChromeDriver();
}
```

此外，如果你不是在 9515 端口上启动 ChromeDriver，需要在同一个类中编辑 `driver` 方法：

```
/**
 * Create the RemoteWebDriver instance.
 *
 * @return \Facebook\WebDriver\Remote\RemoteWebDriver
 */
protected function driver()
{
    return RemoteWebDriver::create(
        'http://localhost:9515', DesiredCapabilities::chrome()
    );
}
```

环境处理

要想在运行测试时强制 Dusk 使用自己的环境文件，可以在项目根目录下创建一个 `.env.dusk.{environment}` 文件，例如，如果你是在 `local` 环境中启动 `dusk` 命令，需要创建 `.env.dusk.local` 文件。

运行测试的时候，Dusk 会备份 `.env` 文件，并将 Dusk 环境文件重命名为 `.env`。一旦测试完成，`.env` 文件会被恢复。

创建浏览器

作为入门，我们编写一个测试来验证可以登录到应用，生成测试之后，编辑测试类将其导航到登录页面，输入认证信息，点击登录按钮。要创建浏览器实例，调用 `browse` 方法：

```
<?php

namespace Tests\Browser;

use App\User;
use Tests\DuskTestCase;
use Laravel\Dusk\Chrome;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class LoginTest extends DuskTestCase
{
    use DatabaseMigrations;

    /**
     * A basic browser test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $user = factory(User::class)->create([
            'email' => 'taylor@laravel.com',
        ]);

        $this->browse(function ($browser) use ($user) {
            $browser->visit('/login')
                ->type('email', $user->email)
                ->type('password', 'secret')
                ->press('Login')
                ->assertPathIs('/home');
        });
    }
}
```

正如你在上面示例中所看到的，`browse` 方法接收一个回调，浏览器实例会通过 Dusk 自动传递给这个回调并且作为主要对象和应用进行交互并生成断言。

编写好测试用例后我们运行 `php artisan dusk` 执行测试：

```
localhost:laravel55 sunqiang$ php artisan dusk
PHPUnit 6.3.0 by Sebastian Bergmann and contributors.

..
2 / 2 (100%)

Time: 13.24 seconds, Memory: 20.00MB

OK (2 tests, 2 assertions)
```

注：该测试可用于测试通过 Artisan 命令 `make:auth` 生成的登录页面。

创建多个浏览器

有时候你需要多个浏览器以便完成测试，例如，在测试通过 Web 套接字进行交互的聊天室页面时就需要多个浏览器。要创建多个浏览器，只需在调用 `browse` 方法时在回调标识中指明需要的浏览器即可：

```
$this->browse(function ($first, $second) {
    $first->loginAs(User::find(1))
        ->visit('/home')
        ->waitForText('Message');

    $second->loginAs(User::find(2))
        ->visit('/home')
        ->waitForText('Message')

        ->type('message', '学院君你好')
        ->press('Send');

    $first->waitForText('学院君你好')
        ->assertSee('学院君');
});
```

调整浏览器窗口尺寸

你可以使用 `resize` 方法来调整浏览器窗口尺寸：

```
$browser->resize(1920, 1080);
```

`maximize` 方法可用于最大化浏览器窗口：

```
$browser->maximize();
```

认证

我们经常会测试需要认证的页面，可以使用 Dusk 的 `loginAs` 方法用于避免在每个测试中与登录页面进行交互，`loginAs` 方法接收用户 ID 或用户模型实例：

```
$this->browse(function ($first, $second) {
    $first->loginAs(User::find(1))
        ->visit('/home');
});
```

注：使用 `loginAs` 方法后，用户会话在某个文件中被所有测试维护。

数据库迁移

测试需要迁移时，例如上面的认证示例，不要使用 `RefreshDatabase trait`，`RefreshDatabase trait` 会影响不能跨请求应用的数据库事务。取而代之的，我们使用 `DatabaseMigrations trait`：

```
<?php

namespace Tests\Browser;

use App\User;
use Tests\DuskTestCase;
use Laravel\Dusk\Chrome;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class ExampleTest extends DuskTestCase
{
    use DatabaseMigrations;
}
```

与元素交互

Dusk 选择器

编写 Dusk 测试最困难的部分之一就是为元素交互选择好的 CSS 选择器，随着时间的推移，前端的改变可能导致 CSS 选择器像下面这样中断测试：

```
// HTML...
<button>Login</button>
// Test...
$browser->click('.login-page .container div > button');
```

Dusk 选择器允许你专注于编写高效的测试而不是记住 CSS 选择器。要定义一个选择器，添加 `dusk` 属性到 HTML 元素，然后，通过在选择器前添加 `@` 前缀在 Dusk 测试中操作元素：

```
// HTML...
<button dusk="login-button">Login</button>
// Test...
$browser->click('@login-button');
```

点击链接

要点击链接，可以使用浏览器实例上的 `clickLink` 方法，`clickLink` 方法将会点击包含给定文本的链接：

```
$browser->clickLink($linkText);
```

注：该方法通过 jQuery 进行交互，如果 jQuery 在页面上无效，Dusk 会自动将其注入到页面，以便在测试期间生效。

文本、值 & 属性

获取 & 设置值

Dusk 提供了多个方法用于和页面元素的当前显示文本、值和属性进行交互，例如，要获取匹配给定选择器的元素值，使用 `value` 方法：

```
// 获取值...
$value = $browser->value('selector');

// 设置值...
$browser->value('selector', 'value');
```

获取文本

`text` 方法可用于获取匹配给定选择器的元素显示文本：

```
$text = $browser->text('selector');
```

获取属性

最后，`attribute` 方法可用于获取匹配给定选择器的元素属性：

```
$attribute = $browser->attribute('selector', 'value');
```

使用表单

输入值

Dusk 提供了多个方法与表单和输入元素进行交互，首先，让我们看一个输入文本到一个输入字段的例子：

```
$browser->type('email', 'xueyuanjun@laravelacademy.org');
```

注意，虽然 `type` 方法在需要的时候可以接收，但是我们也不是必须要传递 CSS 选择器到该方法。如果没有提供 CSS 选择器，Dusk 会搜索包含给定 `name` 属性的输入字段，最后，Dusk 会尝试查找包含给定 `name` 属性的 `textarea`。

要追加文本到某个字段而不清除其值，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”输入的值：

```
$browser->clear('email');
```

下拉框

要在下拉选择框中选择值，可以使用 `select` 方法。和 `type` 方法一样，`select` 方法不需要完整的 CSS 选择器，当传递值到 `select` 方法时，需要传递底层 `option` 值而不是显示文本：

```
$browser->select('size', 'Large');
```

你可以通过省略第二个参数来选择一个随机的 `option`：

```
$browser->select('size');
```

复选框

要“选择”复选字段，可以使用 `check` 方法，和其它输入相关方法一样，不需要传入完整的 CSS 选择器，如果找不到精准匹配的选择器，Dusk 会通过一个相匹配 `name` 属性来搜索复选框：

```
$browser->check('terms');

$browser->uncheck('terms');
```

单选按钮

要“选择”一个单选按钮选项，可以使用 `radio` 方法，和其它输入相关方法一样，不需要完整的 CSS 选择器，如果找不到精准匹配的选择器，Dusk 会搜索匹配 `name` 和 `value` 属性的单选框：

```
$browser->radio('version', 'php7');
```

上传文件

`attach` 方法可用于上传文件到 `file` 输入元素，和其它输入相关方法一样，也不需要完整的 CSS 选择器，如果精准匹配的选择器找不到，Dusk 会通过 `name` 属性搜索匹配的文件输入：

```
$browser->attach('photo', __DIR__.'/photos/me.png');
```

注：使用 `attach` 函数要求系统安装并启用了 PHP `ZIP` 扩展。

使用键盘

`keys` 方法允许你提供比 `type` 方法更多的复杂输入序列到给定元素。例如，你可以存放编辑键及对应输入值，在本例中，`shift` 键会被存放而 `taylor` 被输入到与给定选择器匹配的元素中。输入 `taylor` 后，`otwell` 会以不带任何编辑键的形式被输入：

```
$browser->keys('selector', ['{shift}', 'taylor'], 'otwell');
```

你甚至还可以发送“热键”到包含应用的主 CSS 选择器上：

```
$browser->keys('.app', ['{command}', 'j']);
```

注：所有编辑键都被包裹在 `{}` 中，并匹配定义在 `Facebook\WebDriver\WebDriverKeys` 类中的常量，这些常量可以在 [GitHub](#) 上找到。

使用鼠标

点击元素

`click` 方法可用于点击与给定选择器匹配的元素：

```
$browser->click('.selector');
```

鼠标悬停

`mouseover` 方法可以在你将鼠标悬停在与给定选择器匹配的元素上时使用：

```
$browser->mouseover('.selector');
```

拖拽

`drag` 方法可用于拖放与给定选择器匹配的元素到另一个元素：

```
$browser->drag('.from-selector', '.to-selector');
```

或者，你可以在单个方向上拖放元素：

```
$browser->dragLeft('.selector', 10);
$browser->dragRight('.selector', 10);
$browser->dragUp('.selector', 10);
$browser->dragDown('.selector', 10);
```

选择器作用域

有时候你可能只想要在某个给定选择器上执行一系列操作，例如，你可能想要在某个表格中断言文本是否存在然后在同一个表格中点击按钮。你可以使用 `with` 方法来完成这一功能，传递给 `with` 方法的回调中所有被执行的操作都被限制在给定的选择器中：

```
$browser->with('.table', function ($table) {
    $table->assertSee('Hello World')
        ->clickLink('Delete');
});
```

等待元素

当测试到广泛使用 JavaScript 的应用时，经常需要等待特定元素或数据加载完成之后才能进行测试。Dusk 让这种测试变得简单，通过使用多种方法，你可以在页面上等待元素变得可访问，甚至直到给定 JavaScript 表达式值为 `true`。

等待

如果你需要暂停测试指定毫秒数，可以使用 `pause` 方法：

```
$browser->pause(1000);
```

等待选择器

`waitFor` 方法可用于暂停测试执行直到匹配给定 CSS 选择器的元素在页面上显示。默认情况下，这将会在抛出异常前暂停测试最多 5 秒，如果需要的话，你可以传递自定义超时时间作为该方法的第二个参数：

```
// 最多等待选择器 5s...
$browser->waitFor('.selector');

// 最多等待选择器 1s...
$browser->waitFor('.selector', 1);
```

还可以等到给定选择器在页面消失：

```
$browser->waitUntilMissing('.selector');

$browser->waitUntilMissing('.selector', 1);
```

选择器作用域（有效的话）

某些情况下，你可能想要等待给定选择器然后和匹配给定选择器的元素进行交互。例如，你可能想要等待直到模态窗口有效然后在该窗口中按下“OK”按钮。`whenAvailable` 方法可以用于此种案例。给定回调中的所有元素操作执行都会限制在特定选择器中：

```
$browser->whenAvailable('.modal', function ($modal) {
    $modal->assertSee('Hello World')
        ->press('OK');
});
```

等待文本

`waitForText` 方法可用于等到给定文本在页面上显示：

```
// 最多等待文本 5s...
$browser->waitForText('Hello World');

// 最多等待文本 1s...
$browser->waitForText('Hello World', 1);
```

等待链接

`waitForLink` 方法可用于等到给定链接文本在页面上显示：

```
// 最多等待链接 5s...
$browser->waitForLink('Create');

// 最多等待链接 1s...
$browser->waitForLink('Create', 1);
```

等待页面加载

当进行页面断言如 `$browser->assertPathIs('/home')` 时，如果 `window.location.pathname` 被异步更新则断言会失败。你可以使用 `waitForLocation` 方法等待 `location` 成为给定值：

```
$browser->waitForLocation('/secret');
```

等待页面重载

如果你需要在页面重载后断言，可以使用 `waitForReload` 方法：

```
$browser->click('.some-action')
    ->waitForReload()
    ->assertSee('something');
```

等待 JavaScript 表达式

有时候你可能想要暂停测试直到 JavaScript 表达式值为 `true`。你可以简单通过 `waitFor` 方法完成这一功能。当暂停表达式时，不需要包含 `return` 关键词或者结束分号：

```
// Wait a maximum of five seconds for the expression to be true...
$browser->waitFor('App.dataLoaded');

$browser->waitFor('App.data.servers.length > 0');

// Wait a maximum of one second for the expression to be true...
$browser->waitFor('App.data.servers.length > 0', 1);
```

带回调的等待

Dusk 里面的很多「wait」方法依赖于底层的 `waitUsing` 方法。你可以直接使用这个方法来等待给定回调返回 `true`，`waitUsing` 方法接收等待的最大秒数、闭包执行的时间间隔、闭包以及可选的失败信息：

```
$browser->waitUsing(10, 1, function () use ($something) {
    return $something->isReady();
}, "Something wasn't ready in time.");
```

Vue 断言

Dusk 甚至允许你对 Vue 组件数据的状态进行断言，例如，假设你的应用包含了如下 Vue 组件：

```
// HTML...
<profile dusk="profile-component"></profile>

// Component Definition...

Vue.component('profile', {
    template: '<div>{{ user.name }}</div>',
```

```

data: function () {
    return {
        user: {
            name: '学院君'
        }
    }
}
);

```

你可以像这样断言 Vue 组件状态：

```

/**
 * A basic Vue test example.
 */
public function testVue()
{
    $this->browse(function (Browser $browser) {
        $browser->visit('/')

        ->assertVue('user.name', '学院君', '@profile-component');

    });
}

```

有效的断言

Dusk 提供了多种断言用于应用测试，所有有效的断言罗列如下：

断言	描述
\$browser->assertTitle(\$title)	断言页面标题匹配给定文本
\$browser->assertTitleContains(\$title)	断言页面标题包含给定文本
\$browser->assertUrlIs(\$url)	断言当前 URL(不含查询字符串)匹配给定字符串
\$browser->assertPathBeginsWith(\$path)	断言当前路径以给定路径开头
\$browser->assertPathIs('/home')	断言当前路径匹配给定路径
\$browser->assertPathIsNot('/home')	断言当前路径不匹配给定路径
\$browser->assertRouteIs(\$name, \$parameters)	断言当前 URL 匹配给定命名路由 URL
\$browser->assertQueryStringHas(\$name, \$value)	断言给定查询字符串参数存在并包含给定值
\$browser->assertQueryStringMissing(\$name)	断言给定查询字符串参数缺失
\$browser->assertHasQueryStringParameter(\$name)	断言给定查询字符串存在
\$browser->assertHasCookie(\$name)	断言给定 Cookie 存在
\$browser->assertCookieMissing(\$name)	断言给定 Cookie 不存在
\$browser->assertCookieValue(\$name, \$value)	断言 Cookie 包含给定值
\$browser->assertPlainCookieValue(\$name, \$value)	断言未加密 Cookie 包含给定值
\$browser->assertSee(\$text)	断言给定文本在页面存在
\$browser->assertDontSee(\$text)	断言给定文本在页面不存在
\$browser->assertSeeIn(\$selector, \$text)	断言给定文本在指定选择器中存在
\$browser->assertDontSeeIn(\$selector, \$text)	断言给定文本在指定选择器中不存在
\$browser->assertSourceHas(\$code)	断言给定源码在页面中存在

断言	描述
<code>\$browser->assertSourceMissing(\$code)</code>	断言给定源码在页面中不存在
<code>\$browser->assertSeeLink(\$linkText)</code>	断言给定链接在页面中存在
<code>\$browser->assertDontSeeLink(\$linkText)</code>	断言给定链接在页面中不存在
<code>\$browser->assertInputValue(\$field, \$value)</code>	断言给定输入字段包含给定值
<code>\$browser->assertInputValueIsNot(\$field, \$value)</code>	断言给定输入字段不包含给定值
<code>\$browser->assertChecked(\$field)</code>	断言给定复选框被选中
<code>\$browser->assertNotChecked(\$field)</code>	断言给定复选框未被选中
<code>\$browser->assertRadioSelected(\$field, \$value)</code>	断言给定单选框被选中
<code>\$browser->assertRadioNotSelected(\$field, \$value)</code>	断言给定单选框未被选中
<code>\$browser->assertSelected(\$field, \$value)</code>	断言给定下拉框包含给定选择值
<code>\$browser->assertNotSelected(\$field, \$value)</code>	断言给定下拉框不包含给定选择值
<code>\$browser->assertSelectHasOptions(\$field, \$values)</code>	断言给定数值数组可以被选择
<code>\$browser->assertSelectMissingOptions(\$field, \$values)</code>	断言给定数值数组不能被选择
<code>\$browser->assertSelectHasOption(\$field, \$value)</code>	断言给定数值可以被选择
<code>\$browser->assertValue(\$selector, \$value)</code>	断言匹配给定选择器的元素包含给定值
<code>\$browser->assertVisible(\$selector)</code>	断言匹配给定选择器的元素可见
<code>\$browser->assertMissing(\$selector)</code>	断言匹配给定选择器的元素不可见
<code>\$browser->assertDialogOpened(\$message)</code>	断言带给定字符串的 JavaScript 对话框已打开
<code>\$browser->assertVue(\$property, \$value, \$component)</code>	断言给定 Vue 组件数据与给定值匹配
<code>\$browser->assertVueIsNot(\$property, \$value, \$component)</code>	断言给定 Vue 组件数据与给定值不匹配

页面

有时候，测试需要多个复杂动作在一个序列中执行，这会使得测试代码难以阅读和理解。页面允许你使用单个方法定义优雅的可以在给定页面执行的动作，页面还允许你定义指向应用或单个页面的通用选择器的快捷方式。

生成页面

要生成页面对象，使用 Artisan 命令 `dusk:page` 即可。所有页面对象都位于 `tests/Browser/Pages` 目录：

```
php artisan dusk:page Login
```

配置页面

默认情况下，页面有三个方法：`url`、`assert` 和 `elements`，下面我们来讨论 `url` 和 `assert`，至于 `elements`，我们将会在后续选择器速记中详细讨论。

url 方法

`url` 方法会返回代表页面的 URL 路径，Dusk 会在浏览器中导航到页面时使用这个 URL：

```
/**
 * Get the URL for the page.
 *
 * @return string
 */
public function url()
{
    return '/login';
}
```

assert 方法

`assert` 方法会生成必要的断言来验证浏览器确实在给定页面，完成这个方法不是必须的；不过，你可以在需要的时候生成这些断言。这些断言将会在导航到页面时自动运行：

```
/**
 * Assert that the browser is on the page.
 *
 * @return void
 */
public function assert(Browser $browser)
{
    $browser->assertPathIs($this->url());
}
```

导航到页面

页面被配置后，可以使用 `visit` 方法导航到该页面：

```
use Tests\Browser\Pages\Login;

$browser->visit(new Login);
```

有时候，你可能已经在给定页面上了，并且需要“加载”页面选择器和方法到当前测试上下文，这在点击按钮以及重定向到给定页面时很常见。在本例中，我们能可以在加载页面时使用 `on` 方法：

```
use Tests\Browser\Pages\CreatePlaylist;

$browser->visit('/dashboard')
    ->clickLink('Create Playlist')
    ->on(new CreatePlaylist)
    ->assertSee('@create');
```

速记选择器

页面的 `elements` 方法允许你为页面的 CSS 选择器定义快速的、易记的快捷方式，例如，让我们为应用登录页面的“email”输入定义一个快捷方式：

```
/**
 * Get the element shortcuts for the page.
 *
 * @return array
 */
public function elements()
{
    return [
        '@email' => 'input[name=email]',
    ];
}
```

现在，你可以在任意你想要试用完整 CSS 选择器的地方使用速记选择器：

```
$browser->type('@email', 'xueyuanjun@laravelacademy.org');
```

全局速记选择器

安装完 Dusk 后，一个基本的 `Page` 类会生成到 `tests/Browser/Pages` 目录下。该类包含一个可用于定义全局速记选择器的 `siteElements` 方法，所谓全局速记选择器，指的是在整个应用中生效的速记选择器：

```
/**
 * Get the global element shortcuts for the site.
 *
 * @return array
 */
public static function siteElements()
{
    return [
        '@element' => '#selector',
    ];
}
```

页面方法

除了页面定义的默认方法外，你还可以定义额外的方法用于测试。例如，假设我们正在构建一个音乐管理应用，应用页面常用的功能就是创建一个播放列表，我们不必在每次测试的时候重复编写创建播放列表的逻辑，只需在页面类中定义一个 `createPlaylist` 方法即可：

```
<?php

namespace Tests\Browser\Pages;

use Laravel\Dusk\Browser;

class Dashboard extends Page
{
    // Other page methods...

    /**
     * Create a new playlist.
     *
     * @param \Laravel\Dusk\Browser $browser
     * @param string $name
     * @return void
     */
    public function createPlaylist(Browser $browser, $name)
    {
        $browser->type('name', $name)
            ->check('share')
            ->press('Create Playlist');
    }
}
```

定义好方法后，可以在任意使用该页面的测试中使用，浏览器实例会自动传递给页面方法：

```
use Tests\Browser\Pages\Dashboard;

$browser->visit(new Dashboard)
    ->createPlaylist('My Playlist')
    ->assertSee('My Playlist');
```

组件

组件和 Dusk 的「页面对象」类似，只不过用于在整个应用中作为可复用的 UI 和功能片段，例如导航条或通知窗口。因此，组件并不与特定 URL 绑定。

生成组件

要生成组件，使用 Artisan 命令 `dusk:component`，新生成的组件位于 `test/Browser/Components` 目录下：

```
php artisan dusk:component DatePicker
```

如上所示，「日期选择器」是一个可以在整个应用的不同页面有效的示例组件。手动编写浏览器自动测试逻辑在测试套件的各个测试中选择日期会显得很笨拙，所以，我们可以定义一个 Dusk 组件来表示这个日期选择器，从而方面我们在单个可复用组件中实现相应业务逻辑：

```
<?php

namespace Tests\Browser\Components;

use Laravel\Dusk\Browser;
use Laravel\Dusk\Component as BaseComponent;

class DatePicker extends BaseComponent
{
    /**
     * Get the root selector for the component.
     *
     * @return string
     */
    public function selector()
    {
        return '.date-picker';
    }

    /**
     * Assert that the browser page contains the component.
     *
     * @param Browser $browser
     * @return void
     */
    public function assert(Browser $browser)
    {
```

```

    $browser->assertVisible($this->selector());
}

/**
 * Get the element shortcuts for the component.
 *
 * @return array
 */
public function elements()
{
    return [
        '@date-field' => 'input.datepicker-input',
        '@month-list' => 'div > div.datepicker-months',
        '@day-list' => 'div > div.datepicker-days',
    ];
}

/**
 * Select the given date.
 *
 * @param \Laravel\Dusk\Browser $browser
 * @param int $month
 * @param int $year
 * @return void
 */
public function selectDate($browser, $month, $year)
{
    $browser->click('@date-field')
        ->within('@month-list', function ($browser) use ($month) {
            $browser->click($month);
        })
        ->within('@day-list', function ($browser) use ($day) {
            $browser->click($day);
        });
}
}

```

使用组件

定义好组件后，就可以在任何测试中通过日期选择器轻松选择日期，使用组件还有一个好处就是，如果选择日期的必要逻辑有改动，只需更新这个组件即可：

```

<?php

namespace Tests\Browser;

use Tests\DuskTestCase;
use Laravel\Dusk\Browser;
use Tests\Browser\Components\DatePicker;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class ExampleTest extends DuskTestCase
{
    /**
     * A basic component test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->browse(function (Browser $browser) {
            $browser->visit('/')
                ->within(new DatePicker, function ($browser) {
                    $browser->selectDate(1, 2018);
                })
                ->assertSee('January');
        });
    }
}

```

持续集成

Circle CI

CircleCI 1.0

如果你想通过 Circle CI 运行 Dusk 测试，可以使用这个配置文件作为起点，和 Travis CI 一样，我们使用 `php artisan serve` 命令启动 PHP 自带的 Web 服务器：

```
dependencies:
  pre:
    - curl -L -o google-chrome.deb https://dl.google.com/linux/direct/google-chrome-stable_current_amd64.deb
      - sudo dpkg -i google-chrome.deb
      - sudo sed -i 's|HERE/chrome\"|HERE/chrome\" --disable-setuid-sandbox|g' /opt/google/chrome/google-chrome
      - rm google-chrome.deb

  test:
    pre:
      - "./vendor/laravel/dusk/bin/chromedriver-linux":
        background: true
      - cp .env.testing .env
      - "php artisan serve":
        background: true

  override:
    - php artisan dusk
```

CircleCI 2.0

如果你使用的是 CircleCI 2.0，可以在 `build` 部分添加以下步骤：

```
version: 2
jobs:
  build:
    steps:
      - run: sudo apt-get install -y libsqlite3-dev
      - run: cp .env.testing .env
      - run: composer install -n --ignore-platform-reqs
      - run: npm install
      - run: npm run production
      - run: vendor/bin/phpunit

      - run:
        name: Start Chrome Driver
        command: ./vendor/laravel/dusk/bin/chromedriver-linux
        background: true

      - run:
        name: Run Laravel Server
        command: php artisan serve
        background: true

      - run:
        name: Run Laravel Dusk Tests
        command: php artisan dusk
```

Codeship

想要在 Codeship 中运行 Dusk 测试，添加下面的命令到 Codeship 项目，当然，这些命令这是一个入门示例，如果需要的话你可以添加更多额外命令：

```
phpenv local 7.1
cp .env.testing .env
composer install --no-interaction
nohup bash -c "./vendor/laravel/dusk/bin/chromedriver-linux 2>&1 &" 
nohup bash -c "php artisan serve 2>&1 &" && sleep 5
php artisan dusk
```

Heroku CI

要想在 Heroku CI 上运行 Dusk 测试，添加以下 Google Chrome buildpack 和 脚本到 Heroku 的 `app.json` 文件中：

```
{
  "environments": {
    "test": {
      "buildpacks": [
        { "url": "heroku/php" },
        { "url": "heroku/nodejs" }
      ]
    }
  }
}
```

```
{
  "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 > /dev/null 2>&1 &' && php artisan dusk"
}
}
```

Travis CI

要想通过 **Travis CI** 运行 Dusk 测试，需要拥有 `sudo` 权限，由于 Travis CI 并不是一个图形化的环境，我们还需要一些额外的操作步骤以便启动 Chrome 浏览器。此外，我们将会使用 `php artisan serve` 来启动 PHP 自带的 Web 服务器：

```
sudo: required
dist: trusty

addons:
  chrome: stable

install:
  - cp .env.testing .env
  - travis_retry composer install --no-interaction --prefer-dist --no-suggest

before_script:
  - google-chrome-stable --headless --disable-gpu --remote-debugging-port=9222 http://localhost
  &
  - php artisan serve &

script:
  - php artisan dusk
```

数据库测试

简介

Laravel 提供了多个有用的工具让测试数据库驱动的应用变得更加简单。首先，你可以使用辅助函数 `assertDatabaseHas` 来断言数据库中的数据是否和给定数据集合匹配。例如，如果你想要通过 `email` 值为 `xueyuanjun@laravelacademy.org` 的条件去数据表 `users` 查询是否存在该记录，我们可以这样做：

```
public function testDatabase()
{
    // Make call to application...

    $this->assertDatabaseHas('users', [
        'email' => 'xueyuanjun@laravelacademy.org'
    ]);
}
```

你还可以使用 `assertDatabaseMissing` 辅助函数断言数据在数据库中不存在。

当然，`assertDatabaseHas` 方法和其它类似辅助方法都是为了方便起见进行的封装，你也可以使用其它 PHPUnit 内置的断言方法来进行测试。

生成模型工厂

模型工厂可用于快速填充数据表。要创建一个模型工厂，可以使用 Artisan 命令 `make:factory`：

```
php artisan make:factory PostFactory
```

新创建的工厂类位于 `database/factories` 目录下。

`--model` 选项可用于指示模型工厂对应的模型类。该选项通过给定的模型名称预填充生成的工厂类：

```
php artisan make:factory PostFactory --model=Post
```

生成的 `PostFactory` 内容如下：

```
<?php

use Faker\Generator as Faker;

$factory->define(App\Post::class, function (Faker $faker) {
```

```
return [
    //
];
});
```

每次测试后重置数据库

每次测试后重置数据库通常很有用，这样的话上次测试的数据不会影响下一次测试。`RefreshDatabase` trait 基于你是用的是内存数据库还是关系数据库使用最优方式来迁移测试数据库。在测试类上使用这个 trait，一切都不需要操心，系统会自动帮你在每次测试后重置数据库：

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase
{
    use RefreshDatabase;

    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $response = $this->get('/');

        // ...
    }
}
```

编写工厂

测试时，通常需要在执行测试前插入新数据到数据库。在创建测试数据时，Laravel 允许你使用模型工厂为每个 Eloquent 模型定义默认的属性值集合，而不用手动为每一列指定值。作为开始，我们看一下 `database/factories/ModelFactory.php` 文件，该文件包含了一个工厂定义：

```
use Faker\Generator as Faker;

$factory->define(App\User::class, function (Faker $faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->unique()->safeEmail,
        'password' => '$2y$10$TKh8H1.PfQx37YgCzwiKb.KjNyWgaHb9cbc0QgdIVf1Yg7B77UdFm', // secret
        'remember_token' => str_random(10),
    ];
});
```

在闭包中，作为工厂定义，我们返回该模型上所有属性默认测试值。该闭包接收 PHP 库 `Faker` 实例，从而允许你方便地为测试生成多种类型的随机数据。

你还可以为每个模型创建额外的工厂文件以便更好地组织管理，例如，你可以在 `database/factories` 目录下创建 `UserFactory.php` 和 `CommentFactory.php` 文件。`factories` 目录下的所有文件都会被 Laravel 自动加载。

注：你可以通过添加 `faker_locale` 配置项到 `config/app.php` 配置文件来设置 `Faker` 的本地化。

工厂状态

状态允许你在任意组合中定义可用于模型工厂的离散修改，例如，`User` 模型可能有一个 `delinquent` 状态用于修改某个默认属性值，你可以使用 `state` 方法来定义状态转化。对于简单的状态，可以传递一个属性修改数组：

```
$factory->state(App\User::class, 'delinquent', [
    'account_status' => 'delinquent',
]);
```

如果你的状态需要计算或一个 `$faker` 实例，可以使用一个闭包来计算状态的属性修改：

```
$factory->state(App\User::class, 'address', function ($faker) {
    return [
        'address' => $faker->address,
    ];
});
```

使用工厂

创建模型

定义好工厂后，可以在测试或数据库填充文件中通过全局的 `factory` 方法使用它们来生成模型实例，所以，让我们看一些创建模型的例子，首先，我们使用 `make` 方法，该方法创建模型但不将其保存到数据库：

```
public function testDatabase() {
    $user = factory(App\User::class)->make();
    // 用户模型测试...
}
```

还可以创建多个模型集合或者创建给定类型的模型：

```
// 创建 3 个 App\User 实例...
$users = factory(App\User::class, 3)->make();
```

应用状态

还可以应用任意状态到模型，如果你想要应用多个状态转化到模型，需要指定每个你想要应用的状态名：

```
$users = factory(App\User::class, 5)->states('deliquent')->make();
$users = factory(App\User::class, 5)->states('premium', 'deliquent')->make();
```

覆盖属性

如果你想要覆盖模型中的某些默认值，可以传递数组值到 `make` 方法，只有指定值才会被替换，剩下值保持工厂指定的默认值不变：

```
$user = factory(App\User::class)->make([
    'name' => 'Abigail',
]);
```

持久化模型

`create` 方法不仅能创建模型实例，还可以使用 Eloquent 的 `save` 方法将它们保存到数据库：

```
public function testDatabase()
{
    // 创建单个 App\User 实例...
    $user = factory(App\User::class)->create();

    // 创建 3 个 App\User 实例...
    $users = factory(App\User::class, 3)->create();

    // 在测试中使用模型...
}
```

你可以通过传递数组到 `create` 方法覆盖模型上的属性：

```
$user = factory(App\User::class)->create([
    'name' => 'Abigail',
]);
```

关联关系

在本例中，我们添加一个关联到创建的模型，使用 `create` 方法创建多个模型的时候，会返回一个 Eloquent 集合实例，从而允许你使用集合提供的所有方法，例如 `each`：

```
$users = factory(App\User::class, 3)
    ->create()
    ->each(function($u) {
        $u->posts()->save(factory(App\Post::class)->make());
    });
});
```

关联关系 & 属性闭包

还可以使用工厂中的闭包属性添加关联关系到模型，例如，如果你想要在创建 `Post` 的时候创建一个新的 `User` 实例，可以这么做：

```
$factory->define(App\Post::class, function ($faker) {
    return [
        'title' => $faker->title,
        'content' => $faker->paragraph,
        'user_id' => function () {
            return factory(App\User::class)->create()->id;
        }
});
```

```
];
});
```

这些闭包还接收包含它们的工厂属性数组:

```
$factory->define(App\Post::class, function ($faker) {
    return [
        'title' => $faker->title,
        'content' => $faker->paragraph,
        'user_id' => function () {
            return factory(App\User::class)->create()->id;
        },
        'user_type' => function (array $post) {
            return App\User::find($post['user_id'])->type;
        }
    ];
});
```

有效的断言方法

Laravel 为 [PHPUnit](#) 测试提供了多个数据库断言方法:

方法	描述
<code>\$this->assertDatabaseHas(\$table, array \$data);</code>	断言数据表包含给定数据
<code>\$this->assertDatabaseMissing(\$table, array \$data);</code>	断言数据表不包含给定数据
<code>\$this->assertSoftDeleted(\$table, array \$data);</code>	断言给定记录已经被软删除

模拟

简介

测试 Laravel 应用的时候, 你可能还想要“模拟”应用的特定状态, 以便在测试中不让它们真的执行。例如, 测试触发事件的控制器时, 你可能想要模拟事件监听器以便它们不在测试期间真的执行。这样的话你就可以只测试控制器的 HTTP 响应, 而不必担心事件监听器的执行, 因为事件监听器可以在它们自己的测试用例中被测试。

Laravel 开箱为模拟事件、任务以及门面提供了辅助函数, 这些辅助函数主要是在 [Mockery](#) 之上提供了一个方便的层这样你就不必手动调用复杂的 [Mockery](#) 方法。当然, 你也可以使用 [Mockery](#) 或 [PHPUnit](#) 来创建自己的模拟。

伪造 Bus

作为模拟的替代方案, 你可以使用 [Bus](#) 门面的 [fake](#) 方法来阻止任务被分发, 使用 [fake](#) 的时候, 测试代码执行后会进行断言:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use App\Jobs\ShipOrder;
use Illuminate\Support\Facades\Bus;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
    {
        Bus::fake();

        // Perform order shipping...

        Bus::assertDispatched(ShipOrder::class, function ($job) use ($order) {
            return $job->order->id === $order->id;
        });

        // Assert a job was not dispatched...
        Bus::assertNotDispatched(AnotherJob::class);
    }
}
```

```
}
```

伪造事件

作为模拟的替代方案，你可以使用 `Event` 门面的 `fake` 方法来阻止事件监听器被执行，然后断言事件被分发，甚至检查接收的数据。使用 `fake` 方法时，测试代码执行后会进行断言：

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use App\Events\OrderShipped;
use App\Events\OrderFailedToShip;
use Illuminate\Support\Facades\Event;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase
{
    /**
     * Test order shipping.
     */
    public function testOrderShipping()
    {
        Event::fake();

        // Perform order shipping...

        Event::assertDispatched(OrderShipped::class, function ($e) use ($order) {
            return $e->order->id === $order->id;
        });

        Event::assertNotDispatched(OrderFailedToShip::class);
    }
}
```

伪造邮件

你可以使用 `Mail` 门面的 `fake` 方法阻止邮件发送，然后断言发送给用户的可邮寄类，甚至检查接收的数据。使用 `fake` 的时候，断言会在测试代码执行后进行：

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use App\Mail\OrderShipped;
use Illuminate\Support\Facades\Mail;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
    {
        Mail::fake();

        // Perform order shipping...

        Mail::assertSent(OrderShipped::class, function ($mail) use ($order) {
            return $mail->order->id === $order->id;
        });

        // Assert a message was sent to the given users...
        Mail::assertSent(OrderShipped::class, function ($mail) use ($user) {
            return $mail->hasTo($user->email) &&
                $mail->hasCc('...') &&
                $mail->hasBcc('...');
        });

        // Assert a mailable was sent twice...
    }
}
```

```

        Mail::assertSent(OrderShipped::class, 2);

        // Assert a mailable was not sent...
        Mail::assertNotSent(AnotherMailable::class);
    }
}

```

如果你将邮件发送推到了后台异步队列，需要使用 `assertQueued` 来替代 `assertSent`：

```

Mail::assertQueued(...);
Mail::assertNotQueued(...);

```

伪造通知

你可以使用 `Notification` 门面的 `fake` 方法来阻止通知被发送，之后断言通知是否被发送给用户，甚至可以检查接收的数据。使用 `fake` 的时候，断言会在测试代码执行后进行：

```

<?php

namespace Tests\Feature;

use Tests\TestCase;
use App\Notifications\OrderShipped;
use Illuminate\Support\Facades\Notification;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
    {
        Notification::fake();

        // Perform order shipping...

        Notification::assertSentTo(
            $user,
            OrderShipped::class,
            function ($notification, $channels) use ($order) {
                return $notification->order->id === $order->id;
            }
        );

        // Assert a notification was sent to the given users...
        Notification::assertSentTo(
            [$user], OrderShipped::class
        );

        // Assert a notification was not sent...
        Notification::assertNotSentTo(
            [$user], AnotherNotification::class
        );
    }
}

```

伪造队列

作为模拟的替代方案，你可以使用 `Queue` 门面的 `fake` 方法来阻止任务被推动到队列，然后断言任务是否被推送到队列，甚至检查接收的数据。使用 `fake` 的时候，断言会在测试代码执行后进行：

```

<?php

namespace Tests\Feature;

use Tests\TestCase;
use App\Jobs\ShipOrder;
use Illuminate\Support\Facades\Queue;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase
{
    public function testOrderShipping()

```

```
{
    Queue::fake();

    // Perform order shipping...

    Queue::assertPushed(ShipOrder::class, function ($job) use ($order) {
        return $job->order->id === $order->id;
    });

    // Assert a job was pushed to a given queue...
    Queue::assertPushedOn('queue-name', ShipOrder::class);

    // Assert a job was pushed twice...
    Queue::assertPushed(ShipOrder::class, 2);

    // Assert a job was not pushed...
    Queue::assertNotPushed(AnotherJob::class);
}

}
```

伪造存储

`Storage` 门面的 `fake` 方法允许你轻松构造伪造硬盘，以及使用 `UploadedFile` 类生成的文件，从而极大简化了文件上传测试，例如：

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Http\UploadedFile;
use Illuminate\Support\Facades\Storage;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase
{
    public function testAvatarUpload()
    {
        Storage::fake('avatars');

        $response = $this->json('POST', '/avatar', [
            'avatar' => UploadedFile::fake()->image('avatar.jpg')
        ]);

        // Assert the file was stored...
        Storage::disk('avatars')->assertExists('avatar.jpg');

        // Assert a file does not exist...
        Storage::disk('avatars')->assertMissing('missing.jpg');
    }
}
```

注：默认情况下，`fake` 方法会删除临时目录下的所有文件，如果你想要保留这些文件，可以使用 `persistentFake` 方法。

门面

不同于传统的静态方法调用，`门面`可以被模拟。这与传统静态方法相比是一个巨大的优势，并且你可以对依赖注入进行测试。测试的时候，你可能经常想要在控制器中模拟 Laravel 门面的调用，例如，看看下面的控制器动作：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

class UserController extends Controller
{
    /**
     * 显示应用用户列表
     *
     * @return Response
     */
}
```

```
public function index()
{
    $value = Cache::get('key');

    //
}

}
```

我们可以通过使用 `shouldReceive` 方法模拟 `Cache` 门面的调用，该方法返回一个 `Mockery` 模拟的实例，由于门面通过 `Laravel 服务容器` 进行解析和管理，所以它们比通常的静态类更具有可测试性。例如，我们可以来模拟 `Cache` 门面 `get` 方法的调用：

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Support\Facades\Cache;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;

class UserControllerTest extends TestCase
{
    public function testGetIndex()
    {
        Cache::shouldReceive('get')
            ->once()
            ->with('key')
            ->andReturn('value');

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

        // ...
    }
}
```

注：不要模拟 `Request` 门面，取而代之地，可以在测试时传递期望输入到 HTTP 辅助函数如 `get` 和 `post`，类似地，也不要模拟 `Config` 门面，在测试中调用 `Config::set` 方法即可。

附：官方扩展包

Laravel Cashier

简介

Laravel Cashier 为通过 `Stripe` 和 `Braintree` 实现订阅支付服务提供了一个优雅的流式接口。它封装了几乎所有你恐惧编写的样板化的订阅支付代码。除了基本的订阅管理外，Cashier 还支持处理优惠券、订阅升级/替换、订阅“数量”、取消宽限期，甚至生成 PDF 发票。

注：如果你只需要一次性支付，并不提供订阅，就不应该使用 Cashier，而是直接使用 Stripe 和 Braintree SDK。

配置

Stripe

Composer

首先，通过 Composer 安装 Cashier 扩展包依赖：

```
composer require "laravel/cashier": "~7.0"
```

数据库迁移

使用 Cashier 之前，我们需要准备好数据库。我们需要添加一个字段到 `users` 表，还要创建新的 `subscriptions` 表来处理所有用户订阅：

```
Schema::table('users', function ($table) {
    $table->string('stripe_id')->nullable();
    $table->string('card_brand')->nullable();
    $table->string('card_last_four')->nullable();
    $table->timestamp('trial_ends_at')->nullable();
});

Schema::create('subscriptions', function ($table) {
    $table->increments('id');
    $table->integer('user_id');
```

```
$table->string('name');
$table->string('stripe_id');
$table->string('stripe_plan');
$table->integer('quantity');
$table->timestamp('trial_ends_at')->nullable();
$table->timestamp('ends_at')->nullable();
$table->timestamptamps();
});
```

创建好迁移后，只需简单运行 `migrate` 命令，相应修改就会更新到数据库。

Billable 模型

接下来，添加 `Billable` trait 到模型定义，这个 trait 提供了多个方法以便执行常用支付任务，例如创建订阅、使用优惠券以及更新信用卡信息：

```
use Laravel\Cashier\Billable;

class User extends Authenticatable
{
    use Billable;
}
```

API Key

最后，在配置文件 `services.php` 中配置 Stripe 的 key，你可以在 Stripe 官网个人中心的控制面板中获取这些 Stripe API key 信息：

```
'stripe' => [
    'model' => App\User::class,
    'key' => env('STRIPE_KEY'),
    'secret' => env('STRIPE_API_SECRET'),
],
```

Braintree

Braintree 注意事项

对于很多操作，Stripe 和 Braintree 实现的 Cashier 功能都是一样的，两者都提供了通过信用卡进行订阅支付的功能，但是 Braintree 还支持通过 PayPal 支付。不过，Braintree 也缺失一些 Stripe 支持的功能，在决定使用 Stripe 还是 Braintree 之前，需要考虑以下几点：

- Braintree 支持 PayPal 而 Stripe 不支持；
- Braintree 不支持 `increment` 和 `decrement` 方法，这个是 Braintree 级别的限制，不是 Cashier 级别的；
- Braintree 不支持基于百分比的折扣，这个也是 Braintree 级别的限制，不是 Cashier 级别的

Composer

首先，通过 Composer 安装 Cashier 扩展包依赖：

```
composer require "laravel/cashier-braintree": "~2.0"
```

服务提供者

接下来，在配置文件 `config/app.php` 中注册服务提供者 `Laravel\Cashier\CashierServiceProvider`。

计划信用优惠券

在使用基于 Braintree 的 Cashier 之前，需要在 Braintree 的控制面板中定义一个 `plan-credit` 折扣，这个折扣将会用于从年到月或者从月到年支付的优惠额度比例分配。

配置在 Braintree 控制面板中的这个折扣值可以是你希望的任何值，Cashier 将会在每次使用优惠券的时候通过自定义的值来重写这个默认定义的值。该优惠券之所以是必须的是因为 Braintree 不支持本地根据订阅时长进行优惠额度的按比例分配。

数据库迁移

使用 Cashier 之前，需要准备好数据库。我们需要添加额外的字段到 `users` 表并创建一个新的 `subscriptions` 表用于存放所有用户订阅：

```
Schema::table('users', function ($table) {
    $table->string('braintree_id')->nullable();
    $table->string('paypal_email')->nullable();
    $table->string('card_brand')->nullable();
    $table->string('card_last_four')->nullable();
    $table->timestamp('trial_ends_at')->nullable();
});

Schema::create('subscriptions', function ($table) {
    $table->increments('id');
    $table->integer('user_id');
    $table->string('name');
    $table->string('braintree_id');
    $table->string('braintree_plan');
    $table->integer('quantity');
    $table->timestamp('trial_ends_at')->nullable();
    $table->timestamp('ends_at')->nullable();
    $table->timestamptamps();
});
```

迁移被创建之后，只需要执行 Artisan 命令 `migrate` 即可。

Billable 模型

接下来，添加 `Billable` trait 到模型定义：

```
use Laravel\Cashier\Billable;

class User extends Authenticatable
{
    use Billable;
}
```

API Key

接下来，需要在配置文件 `services.php` 中配置如下选项：

```
'braintree' => [
    'model' => App\User::class,
    'environment' => env('BRAINTREE_ENV'),
    'merchant_id' => env('BRAINTREE_MERCHANT_ID'),
    'public_key' => env('BRAINTREE_PUBLIC_KEY'),
    'private_key' => env('BRAINTREE_PRIVATE_KEY'),
],
```

然后你需要在服务提供者 `AppServiceProvider` 的 `boot` 方法中添加对 Braintree SDK 的调用：

```
\Braintree\Configuration::environment(config('services.braintree.environment'));
\Braintree\Configuration::merchantId(config('services.braintree.merchant_id'));
\Braintree\Configuration::publicKey(config('services.braintree.public_key'));
\Braintree\Configuration::privateKey(config('services.braintree.private_key'));
```

货币配置

Cashier 默认货币是美元（USD），你可以在某个服务提供者的 `boot` 方法中通过调用 `Cashier::useCurrency` 方法来更改默认的货币，`useCurrency` 方法接收两个字符串参数：货币及货币符号：

```
use Laravel\Cashier\Cashier;

Cashier::useCurrency('rmb', '¥');
```

订阅

创建订阅

要创建一个订阅，首先要获取一个账单模型的实例，通常是 `App\User` 的实例。获取到该模型实例之后，你可以使用 `newSubscription` 方法来创建该模型的订阅：

```
$user = User::find(1);

$user->newSubscription('main', 'premium')->create($stripeToken);
```

第一个传递给 `newSubscription` 方法的参数是该订阅的名字，如果应用只有一个订阅，可以将其称作 `main` 或 `primary`，第二个参数用于指定用户订阅的 Stripe/Braintree 计划，该值对应 Stripe 或 Braintree 中相应计划的 id。

`create` 方法会自动创建这个 Stripe 订阅，同时更新数据库中 Stripe 的客户 ID（即 `users` 表中的 `stripe_id`）和其它相关的账单信息。

额外的用户信息

如果你想要指定额外的客户信息，你可以将其作为第二个参数传递给 `create` 方法：

```
$user->newSubscription('main', 'monthly')->create($stripeToken, [
    'email' => $email,
]);
```

要了解更多 Stripe 支持的字段，可以查看 Stripe 关于[创建消费者的文档](#)或者相应的 [Braintree 文档](#)。

优惠券

如果你想要在创建订阅的时候使用优惠券，可以使用 `withCoupon` 方法：

```
$user->newSubscription('main', 'monthly')
    ->withCoupon('code')
    ->create($stripeToken);
```

检查订阅状态

用户订阅你的应用后，你可以使用各种便利的方法来简单检查订阅状态。首先，如果用户有一个有效的订阅，则 `subscribed` 方法返回 `true`，即使订阅现在处于试用期：

```
if ($user->subscribed('main')) {
    //
}
```

`subscribed` 方法还可以用于[路由中间件](#)，基于用户订阅状态允许你对路由和控制器的访问进行过滤：

```
public function handle($request, Closure $next) {
    if ($request->user() && ! $request->user()->subscribed('main')) {
```

```
// This user is not a paying customer...
return redirect('billing');
}

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

如果你想要判断一个用户是否还在试用期，可以使用 `onTrial` 方法，该方法对于还处于试用期的用户显示警告信息很有用：

```
if ($user->subscription('main')->onTrial()) {
    //
}
```

`subscribedToPlan` 方法可用于判断用户是否基于 Stripe/Braintree ID 订阅了给定的计划，在本例中，我们会判断用户的 `main` 订阅是否订阅了 `monthly` 计划：

```
if ($user->subscribedToPlan('monthly', 'main')) {
    //
}
```

已取消的订阅状态

要判断用户是否曾经是有效的订阅者，但现在取消了订阅，可以使用 `cancelled` 方法：

```
if ($user->subscription('main')->cancelled()) {
    //
}
```

你还可以判断用户是否曾经取消过订阅，但现在仍然在“宽限期”直到完全失效。例如，如果一个用户在 3 月 5 号取消了一个实际有效期到 3 月 10 号的订阅，该用户处于“宽限期”直到 3 月 10 号。注意 `subscribed` 方法在此期间仍然返回 `true`。

```
if ($user->subscription('main')->onGracePeriod()) {
    //
}
```

修改计划

用户订阅应用后，偶尔想要改变到新的订阅计划，要将用户切换到新的订阅，传递计划标识到 `swap` 方法：

```
$user = App\User::find(1);
$user->subscription('main')->swap('provider-plan-id');
```

如果用户在试用，试用期将会被维护。还有，如果订阅存在多个，数量也可以被维护。

如果你想要切换计划并取消用户所在的所有试用期，你可以使用 `skipTrial` 方法：

```
$user->subscription('main')
    ->skipTrial()
    ->swap('provider-plan-id');
```

订阅数量

注：只有 Stripe 版本的 Cashier 支持订阅数量，Braintree 没有提供类似 Stripe “数量”这样的特性。

有时候订阅也会被数量影响，例如，应用中每个账户每月需要付费\$10，要简单增加或减少订阅数量，使用 `incrementQuantity` 和 `decrementQuantity` 方法：

```
$user = User::find(1);

$user->subscription('main')->incrementQuantity();

// Add five to the subscription's current quantity...
$user->subscription('main')->incrementQuantity(5);

$user->subscription('main')->decrementQuantity();

// Subtract five to the subscription's current quantity...
$user->subscription('main')->decrementQuantity(5);
```

你也可以使用 `updateQuantity` 方法指定数量：

```
$user->subscription('main')->updateQuantity(10);
```

想要了解更多订阅数量信息，查阅相关 [Stripe 文档](#)。

订阅税金

要指定用户支付订阅的税率，实现账单模型的 `taxPercentage` 方法，并返回一个在 0 到 100 之间的数值，不要超过两位小数：

```
public function taxPercentage() {
    return 20;
}
```

这将使你可以在模型基础上使用税率，对跨越不同国家不同税率的用户很有用。

注: `taxPercentage` 方法只能用于订阅支付，如果你使用 `Cashier` 生成一次性账单，需要手动指定税率。

取消订阅

要取消订阅，可以调用用户订阅上的 `cancel` 方法:

```
$user->subscription('main')->cancel();
```

当订阅被取消时，`Cashier` 将会自动设置数据库中的 `ends_at` 字段。该字段用于了解 `subscribed` 方法什么时候开始返回 `false`。例如，如果客户 3 月 1 号份取消订阅，但订阅直到 3 月 5 号才会结束，那么 `subscribed` 方法继续返回 `true` 直到 3 月 5 号。

你可以使用 `onGracePeriod` 方法判断用户是否已经取消订阅但仍然在“宽限期”:

```
if ($user->subscription('main')->onGracePeriod()) {
    //
}
```

如果你想要立即取消订阅，调用用户订阅上的方法 `cancelNow` 即可:

```
$user->subscription('main')->cancelNow();
```

恢复订阅

如果用户已经取消订阅但想要恢复该订阅，可以使用 `resume` 方法，前提是该用户必须在宽限期内:

```
$user->subscription('main')->resume();
```

如果该用户取消了一个订阅然后在订阅失效之前恢复了这个订阅，则不会立即支付该账单，取而代之的，他们的订阅只是被重新激活，并回到正常的支付周期。

更新信用卡

`updateCard` 方法可用于更新用户的信用卡信息，该方法接收一个 `Stripe` 令牌并设置新的信用卡作为支付源:

```
$user->updateCard($stripeToken);
```

订阅试用期

带信用卡信息

如果你想要在提供给用户试用期的同时预先收集支付方式信息，可以在创建订阅的时候使用 `trialDays` 方法:

```
$user = User::find(1);

$user->newSubscription('main', 'monthly')
    ->trialDays(10)
    ->create($stripeToken);
```

该方法会在数据库订阅记录上设置试用期结束日期，以便告知 `Stripe/Braintree` 在此之前不要计算用户的账单信息。

注: 如果用户的订阅没有在试用期结束之前取消，则会在试用期结束时立即支付，所以要确保通知用户试用期结束时间。

`trialUntil` 方法允许你提供一个 `DateTime` 实例来指定试用期什么时候结束:

```
use Carbon\Carbon;

$user->newSubscription('main', 'monthly')
    ->trialUntil(Carbon::now()->addDays(10))
    ->create($stripeToken);
```

可以使用用户实例或订阅实例上的 `onTrial` 方法判断用户是否处于试用期，下面两个例子作用是等价的:

```
if ($user->onTrial('main')) {
    //
}

if ($user->subscription('main')->onTrial()) {
    //
}
```

不带信用卡信息

如果你不想在提供试用期的时候收集用户支付方式信息，只需设置用户记录的 `trial_ends_at` 字段为期望的试用期结束日期即可，这通常在用户注册期间完成:

```
$user = User::create([
    // Populate other user properties...
    'trial_ends_at' => Carbon::now()->addDays(10),
]);
```

注：确保已添加 `trial_ends_at` 日期修改器到模型定义。

`Cashier` 将这种类型的试用期看作“一般体验”，因为这种使用并没有附加到任何已经在的订阅，如果当前日期没有超过 `trial_ends_at` 的值，`User` 实例上的 `onTrial` 方法将返回 `true`：

```
if ($user->onTrial()) {
    // User is within their trial period...
}
```

如果你想要知道用户是否在“一般”试用期并且还没有创建实际的订阅还可以使用 `onGenericTrial` 方法：

```
if ($user->onGenericTrial()) {
    // User is within their "generic" trial period...
}
```

一旦你准备好为用户创建实际的订阅，可以使用 `newSubscription` 方法：

```
$user = User::find(1);

$user->newSubscription('main', 'monthly')->create($stripeToken);
```

处理 Stripe Webhooks

Stripe 和 Braintree 都可以通过 webhooks 通知应用各种事件，要处理 Stripe webhooks，需要定义一个指向 `Cashier webhook` 控制器的路由，这个控制器将会处理所有输入 webhook 请求并将它们分发到合适的控制器方法：

```
Route::post(
    'stripe/webhook',
    '\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook'
);
```

注：注册好控制器后，还要在 Stripe 控制面板中配置 webhook URL。

默认情况下，这个控制器将会自动对支付失败次数（这个次数可以在 Stripe 设置中定义）过多的订阅进行取消；此外，我们很快会发现，你可以扩展这个控制器来处理任何你想要处理的 webhook 事件。

Webhooks & CSRF 防护

由于 Stripe webhook 需要绕开 Laravel 的 `CSRF 保护`，所以需要将其罗列到 `VerifyCsrfToken` 中间件的排除列表或者将其置于 `web` 中间件组之外：

```
protected $except = [
    'stripe/*',
];
```

定义 Webhook 事件处理器

`Cashier` 会基于支付失败次数自动取消订阅，但是如果你想要处理额外的 Stripe webhook 事件，扩展 `Webhook` 控制器即可。定义的方法名需要与 `Cashier` 约定的格式保持一致，特别是方法名需要以 `handle` 开头并且是想要处理的 Stripe webhook 的驼峰格式。例如，如果你想要处理 `invoice.payment_succeeded` webhook，则需要添加一个 `handleInvoicePaymentSucceeded` 方法到控制器：

```
<?php

namespace App\Http\Controllers;

use Laravel\Cashier\Http\Controllers\WebhookController as CashierController;

class WebhookController extends CashierController
{
    /**
     * Handle a Stripe webhook.
     *
     * @param array $payload
     * @return Response
     */
    public function handleInvoicePaymentSucceeded($payload)
    {
        // Handle The Event
    }
}
```

接下来，在 `routes/web.php` 中定义一个指向 `Cashier` 控制器的路由：

```
Route::post(
    'stripe/webhook',
    '\App\Http\Controllers\WebhookController@handleWebhook'
);
```

失败的订阅

如果用户的信用卡过期怎么办？不用担心 —— Cashier webhook 控制器可以轻松为你取消该用户的订阅，正如上面所提到的，你所需要做的只是将路由指向该控制器：

```
Route::post(
    'stripe/webhook',
    '\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook'
);
```

就是这样，失败的支付将会被控制器捕获和处理，该控制器将会在 Stripe 判断订阅支付失败次数（通常是 3 次）达到上限时取消该用户的订阅。

处理 Braintree Webhooks

Stripe 和 Braintree 都可以通过 webhooks 通知应用各种事件，要处理 Braintree webhooks，需要定义一个指向 Cashier webhook 控制器的路由，这个控制器将会处理所有输入 webhook 请求并将它们分发到合适的控制器方法：

```
Route::post(
    'braintree/webhook',
    '\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook'
);
```

注：注册好控制器后，还要在 Braintree 控制面板中配置 webhook URL。

默认情况下，这个控制器将会自动对支付失败次数（这个次数可以在 Braintree 设置中定义）过多的订阅进行取消；此外，我们很快会发现，你可以扩展这个控制器来处理任何你想要处理的 webhook 事件。

Webhooks & CSRF 防护

由于 Braintree webhook 需要绕开 Laravel 的 CSRF 保护，所以需要将其罗列到 `VerifyCsrfToken` 中间件的排除列表或者将其置于 `web` 中间件组之外：

```
protected $except = [
    'braintree/*',
];
```

定义 Webhook 事件处理器

Cashier 会基于支付失败次数自动取消订阅，但是如果你想要处理额外的 Braintree webhook 事件，扩展 Webhook 控制器即可。定义的方法名需要与 Cashier 约定的格式保持一致，特别是方法名需要以 `handle` 开头并且是想要处理的 Braintree webhook 的驼峰格式。例如，如果你想要处理 `dispute_opened` webhook，则需要添加一个 `handleDisputeOpened` 方法到控制器：

```
<?php

namespace App\Http\Controllers;

use Braintree\WebhookNotification;
use Laravel\Cashier\Http\Controllers\WebhookController as CashierController;

class WebhookController extends CashierController
{
    /**
     * Handle a Braintree webhook.
     *
     * @param WebhookNotification $webhook
     * @return Response
     */
    public function handleDisputeOpened(WebhookNotification $notification)
    {
        // Handle The Event
    }
}
```

失败的订阅

如果用户的信用卡过期怎么办？不用担心 —— Cashier webhook 控制器可以轻松为你取消该用户的订阅，正如上面所提到的，你所需要做的只是将路由指向该控制器：

```
Route::post(
    'braintree/webhook',
    '\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook'
);
```

就是这样，失败的支付将会被控制器捕获和处理，该控制器将会在 Braintree 判断订阅支付失败次数（通常是 3 次）达到上限时取消该用户的订阅。不要忘记在 Braintree 控制面板中配置 webhook URI。

一次性支付

简单支付

注：使用 Stripe 时，`charge` 方法可以接收应用所使用货币对应的最小单位金额，但是使用 Braintree 时，必须传递完整的美元金额到 `charge` 方法。

如果你想要使用订阅客户的信用卡一次性结清账单，可以使用账单模型实例上的 `charge` 方法：

```
// Stripe Accepts Charges In Cents...
$user->charge(100);

// Braintree Accepts Charges In Dollars...
$user->charge(1);
```

`charge` 方法接收一个数组作为第二个参数，允许你传递任何你想要传递的底层 Stripe/Braintree 账单创建参数，创建账单时我们可以参考 Stripe 或者 Braintree 文档提供的可用选项：

```
$user->charge(100, [
    'custom_option' => $value,
]);
```

如果支付失败 `charge` 方法将抛出异常，如果支付成功，该方法会返回完整的 Stripe / Braintree 响应：

```
try {
    $response = $user->charge(100);
} catch (Exception $e) {
    //
}
```

带发票的支付

有时候你需要创建一个一次性支付并且同时生成对应发票以便为用户提供一个 PDF 单据，`invoiceFor` 方法可以帮助我们实现这个需求。例如，让我们为用户的“一次性费用”生成一张\$5.00 的发票：

```
// Stripe Accepts Charges In Cents...
$user->invoiceFor('One Time Fee', 500);

// Braintree Accepts Charges In Dollars...
$user->invoiceFor('One Time Fee', 5);
```

该单据会通过用户信用卡立即支付，`invoiceFor` 方法还可以接收一个数组作为第三个参数，从而允许你传递任意你希望的选项到底层 Stripe/Braintree 支付创建：

```
$user->invoiceFor('One Time Fee', 500, [
    'custom-option' => $value,
]);
```

注：`invoiceFor` 方法会创建一个对失败支付进行重试的 Stripe 单据，如果你不想要单据重试失败的支付，需要在首次支付失败后使用 Stripe API 关闭它们。

发票

你可以使用 `invoices` 方法轻松获取账单模型的发票数组：

```
$invoices = $user->invoices();

// Include pending invoices in the results...
$invoices = $user->invoicesIncludingPending();
```

当列出客户发票时，你可以使用发票的辅助函数来显示相关的发票信息。例如，你可能想要在表格中列出每张发票，从而方便用户下载它们：

```
<table>
    @foreach ($invoices as $invoice)
        <tr>
            <td>{{ $invoice->date()->toFormattedDateString() }}</td>
            <td>{{ $invoice->total() }}</td>
            <td><a href="/user/invoice/{{ $invoice->id }}">Download</a></td>
        </tr>
    @endforeach
</table>
```

生成 PDF 发票

在生成 PDF 发票之前，需要安装 PHP 库 `dompdf`：

```
composer require dompdf/dompdf
```

然后，在路由或控制器中，使用 `downloadInvoice` 方法生成发票的 PDF 下载，该方法将会自动生成相应的 HTTP 响应发送下载到浏览器：

```
use Illuminate\Http\Request;

Route::get('user/invoice/{invoice}', function (Request $request, $invoiceId) {
    return $request->user()->downloadInvoice($invoiceId, [
        'vendor' => 'Your Company',
        'product' => 'Your Product',
    ]);
});
```

Envoy Task Runner

简介

Laravel Envoy 为定义运行在远程主机上的通用任务提供了一套干净的、最简化的语法。使用 Blade 风格语法，你可以轻松为开发设置任务，Artisan 命令，以及更多。目前，Envoy 只支持 Mac 和 Linux 操作系统。

安装

首先，使用 Composer 的 `global require` 命令全局安装 Envoy：

```
composer global require laravel/envoy
```

由于全局的 Composer 库有时候会导致包版本冲突，所以需要考虑使用 `cgr`，该命令是 `composer global require` 命令的替代实现，`cgr` 库的安装指南可以在 [GitHub 上找到](#)。

注：确保 `~/.composer/vendor/bin` 目录已经在系统路径 PATH 中，否则在终端中由于找不到 `envoy` 而无法执行该命令。

更新 Envoy

还可以使用 Composer 保持安装的 Envoy 是最新版本，因为 `composer global update` 命令将会更新所有全局安装的 Composer 包：

```
composer global update
```

编写任务

所有的 Envoy 任务都定义在项目根目录下的 `Envoy.blade.php` 文件中，下面是一个让你上手的示例：

```
@servers(['web' => 'user@192.168.1.1'])

@task('foo', ['on' => 'web'])
    ls -la
@endtask
```

正如你所看到的，`@servers` 数组定义在文件顶部，从而允许你在任务声明中使用 `on` 选项引用这些服务器，在 `@task` 声明中，应该放置将要在服务器上运行的 Bash 代码。

你可以通过指定服务器 IP 地址为 `127.0.0.1` 强制脚本在本地运行：

```
@servers(['localhost' => '127.0.0.1'])
```

起步

有时候，你需要在执行 Envoy 任务之前执行一些 PHP 代码，可以在 Envoy 文件中使用 `@setup` 指令来声明变量和要执行的 PHP 代码：

```
@setup
    $now = new DateTime();
    $environment = isset($env) ? $env : "testing";
@endsetup
```

如果你需要在任务执行之前引入其它 PHP 文件，可以在 `Envoy.blade.php` 文件顶部使用 `@include` 来引入：

```
@include('vendor/autoload.php')

@task('foo')
    # ...
@endtask
```

变量

如果需要的话，你可以使用命令行传递选项值到 Envoy 任务：

```
envoy run deploy --branch=master
```

你可以在任务中通过 Blade 的“echo”语法访问该选项，当然，你还可以使用 `if` 语句并在任务中循环，例如，下面我们在执行 `git pull` 命令前验证 `$branch` 变量是否存在：

```
@servers(['web' => '192.168.1.1'])
```

```
@task('deploy', ['on' => 'web'])
    cd site
    @if ($branch)
        git pull origin {{ $branch }}
    @endif
    php artisan migrate
@endtask
```

Story

`story` 通过一个便捷的名字对任务集合进行分组，从而允许你将小而专注的任务组合成大的任务，例如，`deploy story` 将会通过在其定义中罗列任务名的方式运行 `git` 和 `composer` 任务：

```
@servers(['web' => '192.168.1.1'])

@story('deploy')
    git
    composer
@endstory

@task('git')
    git pull origin master
@endtask

@task('composer')
    composer install
@endtask
```

`story` 编写好之后，就可以像普通任务一样运行：

```
envoy run deploy
```

多个服务器

你可以轻松地在多主机上运行同一个任务，首先，添加额外服务器到 `@servers` 声明，每个服务器应该被指配一个唯一的名字。定义好服务器后，在任务的 `on` 数组中列出所有服务器即可：

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2']])
    cd site
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

平行运行

默认情况下，该任务将会依次在每个服务器上执行，这意味着，该任务在第一台服务器上运行完成后才会开始在第二台服务器运行。如果你想要在多个服务器上平行运行，添加 `parallel` 选项到任务声明：

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2'], 'parallel' => true])
    cd site
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

运行任务

要运行一个在 `Envoy.blade.php` 文件中定义的任务，需要执行 `Envoy` 的 `run` 命令，然后传递你要执行的任务或 `story` 的名字。Envoy 将会运行命令并从服务打印输出：

```
envoy run task
```

确认任务执行

如果你想要在服务器上运行给定任务之前弹出提示进行确认，可以在任务声明中使用 `confirm` 指令：

```
@task('deploy', ['on' => 'web', 'confirm' => true])
    cd site
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

通知

Slack

Envoy 还支持在任务执行之后发送通知到 Slack。`@slack` 指令接收一个 Slack 钩子 URL 和频道名称，你可以通过在 Slack 控制面板中创建“Incoming WebHooks”集成来获取 webhook URL，需要传递完整的 webhook URL 到 `@slack` 指令：

```
@after
@slack('webhook-url', '#bots')
@endafter
```

你可以提供下面两种其中之一作为频道参数：

- 发送通知到频道: `#channel`
- 发送通知到用户: `@user`

Laravel Horizon

简介

Horizon 为 Laravel 提供了基于 Redis 的、拥有美观后台的、代码驱动配置的队列系统。Horizon 让我们可以轻松监控队列系统的关键指标，例如任务吞吐量、运行时间和失败任务等。

所有的队列进程配置都存放在一个单独的简单配置文件中，这样的话配置文件就可以存放到源码控制以便团队所有成员的合作。

安装

注：由于 Horizon 使用了异步进程信号，所以 PHP 7.1+ 以上版本才可以使用。

我们使用 Composer 安装 Horizon 到 Laravel 项目：

```
composer require laravel/horizon
```

安装完成后，使用 Artisan 命令 `vendor:publish` 发布前端资源：

```
php artisan vendor:publish --provider="Laravel\Horizon\HorizonServiceProvider"
```

配置

发布好前端资源后，主配置文件就会出现在 `config/horizon.php`。在这个配置文件中，你可以配置队列进程选项以及每个包含目的描述的配置项，所以使用 Horizon 前务必浏览下这个配置文件。

balance 配置项

Horizon 提供了三种负载均衡策略以供选择：`simple`、`auto` 和 `false`，`simple` 是默认策略，在进程之间平均分配进入任务：

```
'balance' => 'simple',
```

`auto` 策略基于队列当前负载调整每个队列的工作进程数量。例如，如果 `notifications` 队列有 1000 个等待执行的任务而 `render` 队列是空的，那么 Horizon 将会为 `notifications` 队列分配更多的工作进程直到队列为空。

如果把 `balance` 选项设置为 `false`，就会使用默认的 Laravel 行为，也就是按照配置文件中的排列顺序处理队列。

后台认证

我们可以通过 `/horizon` 访问 Horizon 后台：

The screenshot shows the Laravel Horizon dashboard. On the left is a sidebar with links: Dashboard, Monitoring, Metrics, Recent Jobs, and Failed. The main area is titled 'Overview' and contains a table with the following data:

JOBS PER MINUTE	JOBS PAST HOUR	FAILED JOBS PAST HOUR	STATUS
0	0	0	✖ Inactive

TOTAL PROCESSES	MAX WAIT TIME	MAX RUNTIME	MAX THROUGHPUT
0	-	-	-

Laravel is a trademark of Taylor Otwell. Copyright © Laravel LLC. All rights reserved.

默认情况下，你只能在 `local` 环境下访问这个后台。如果想要为后台定义更多的特定访问策略，需要使用 `Horizon::auth` 方法。`auth` 方法接收一个返回 `true` 或 `false` 的回调，从而决定用户是否可以访问 Horizon 后台。通常，我们会在 `AppServiceProvider` 的 `boot` 方法中调用 `Horizon::auth`：

```
Horizon::auth(function ($request) {
    // return true / false;
});
```

运行 Horizon

如果你已经在配置文件 `config/horizon.php` 中配置过工作进程，就可以使用 Artisan 命令 `horizon` 来启动 Horizon，该命令会启动所有配置的工作进程：

```
php artisan horizon
```

你可以使用 Artisan 命令 `horizon:pause` 和 `horizon:continue` 来暂停或继续处理队列任务：

```
php artisan horizon:pause
```

```
php artisan horizon:continue
```

你还可以使用 Artisan 命令 `horizon:terminate` 来优雅地终止 Horizon 主进程 —— Horizon 会在所有当前正在执行的任务全部完成后退出：

```
php artisan horizon:terminate
```

部署 Horizon

如果要将 Horizon 部署到线上服务器，需要配置一个进程监控来监控 `php artisan horizon` 命令的运行并在异常退出的情况下重启该进程。部署新代码到服务器的时候，需要终止 Horizon 主进程以便通过配置的进程监控以最新代码重启进程。如上所述，我们可以通过 Artisan 命令 `horizon:terminate` 优雅地终止 Horizon 主进程。

Supervisor 配置

如果你在使用 Supervisor 进程监控来管理 `horizon` 进程，可以像这样配置（按照自己的实际情况修改相应的目录信息）：

```
[program:horizon]
process_name=%(program_name)s
command=php /home/forge/app.com/artisan horizon
autostart=true
autorestart=true
user=forge
redirect_stderr=true
stdout_logfile=/home/forge/app.com/horizon.log
```

注：如果你对如何管理自己的服务器感到摸不着头脑，可以考虑使用 [Laravel Forge](#)，Forge 提供了 PHP 7+ 版本的服务器，并且拥有运行最新版 Laravel 应用所需的所有软件工具集。

标签

Horizon 允许分配“标签”到任务，包括邮件、事件广播、通知以及队列事件监听器等。实际上，Horizon 会基于附加到任务的 Eloquent 模型为大部分任务以智能的方式自动打上标签。例如，我们来看看下面这个例子：

```
<?php
namespace App\Jobs;
```

```

use App\Video;
use Illuminate\Bus\Queueable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;

class RenderVideo implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    /**
     * The video instance.
     *
     * @var \App\Video
     */
    public $video;

    /**
     * Create a new job instance.
     *
     * @param \App\Video $video
     * @return void
     */
    public function __construct(Video $video)
    {
        $this->video = $video;
    }

    /**
     * Execute the job.
     *
     * @return void
     */
    public function handle()
    {
        //
    }
}

```

如果任务被推送到队列时附带了一个 `id` 为 1 的 `App\Video` 实例，它将会自动打上 `App\Video:1` 的标签，这是因为 Horizon 会检查队列任务属性中的 Eloquent 模型，如果 Eloquent 模型被找到，Horizon 就会使用模型类名和主键为任务智能地打上标签：

```

$video = App\Video::find(1);

App\Jobs\RenderVideo::dispatch($video);

```

手动打标签

如果你想要手动定义某个队列对象的标签，可以在该类中定义一个 `tags` 方法：

```

class RenderVideo implements ShouldQueue
{
    /**
     * Get the tags that should be assigned to the job.
     *
     * @return array
     */
    public function tags()
    {
        return ['render', 'video:' . $this->video->id];
    }
}

```

通知

注：在使用通知之前，需要通过 Composer 安装 `guzzlehttp/guzzle` 依赖。在配置 Horizon 发送短信通知前，还要回顾下 `Nexmo` 通知驱动的预备知识。

如果你想要在某个队列任务等待很长时间后被通知，可以使用 `Horizon::routeMailNotificationsTo`、`Horizon::routeSlackNotificationsTo` 以及 `Horizon::routeSmsNotificationsTo` 方法。你可以在 `AppServiceProvider` 中调用这些方法：

```

Horizon::routeMailNotificationsTo('example@example.com');
Horizon::routeSlackNotificationsTo('slack-webhook-url', '#channel');
Horizon::routeSmsNotificationsTo('15556667777');

```

配置通知等待时间下限

你可以在配置文件 `config/horizon.php` 中通过修改 `waits` 配置项来配置每个连接/队列上的等待时间下限（秒）：

```
'waits' => [
    'redis:default' => 60,
],
```

监控

Horizon 提供了一个监控后台查看任务和队列的等待时间和吞吐量信息，为了获取实时信息，可以配置 Horizon 的 Artisan 命令 `snapshot` 通过应用的调度器每五分钟运行一次：

```
/***
 * Define the application's command schedule.
 *
 * @param \Illuminate\Console\Scheduling\Schedule $schedule
 * @return void
 */
protected function schedule(Schedule $schedule)
{
    $schedule->command('horizon:snapshot')->everyFiveMinutes();
}
```

Laravel Scout

简介

Laravel Scout 为 [Eloquent 模型](#)全文搜索实现提供了简单的、基于驱动的解决方案。通过使用模型观察者，Scout 会自动同步更新模型记录的索引。目前，Scout 通过 [Algolia](#) 驱动提供搜索功能，不过，编写自定义驱动很简单，你可以很轻松地通过自己的搜索实现来扩展 Scout。

注：Algolia 是一个托管式的全文搜索引擎，我们可以通过其提供的 API 在网站和移动应用中快速实现实时搜索功能。Algolia 提供的服务是收费的，不过我们可以使用其免费版本进行测试，免费版本支持 1 万条记录/10 万次操作。

安装

首先，我们通过 Composer 包管理器来安装 Scout：

```
composer require laravel/scout
```

安装完成后，需要通过 Artisan 命令 `vendor:publish` 发布 Scout 配置，该命令会发布配置文件 `scout.php` 到 `config` 目录：

```
php artisan vendor:publish --provider="Laravel\Scout\ScoutServiceProvider"
```

最后，如果你想要某个模型支持 Scout 搜索，需要添加 [Laravel\Scout\Searchable](#) trait 到对应模型类，该 trait 会注册模型观察者来保持搜索驱动与模型记录数据的一致性：

```
<?php

namespace App;

use Laravel\Scout\Searchable;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    use Searchable;
}
```

队列

虽然不强制，但是在使用 Scout 之前强烈建议配置一个队列驱动。运行一个队列进程将允许 Scout 把所有同步模型信息到搜索索引的操作推送到队列中，从而为应用的 Web 接口提供更快的响应时间。

配置好队列驱动后，在配置文件 `config/scout.php` 中设置 `queue` 选项的值为 `true`：

```
'queue' => env('SCOUT_QUEUE', true),
```

驱动预备知识

Algolia

使用 Algolia 驱动的话，需要在配置文件 `config/scout.php` 中设置 Algolia 的 `id` 和 `secret` 信息（这些信息可以在注册登录 Algolia 之后在用户后台找到，分别对应 API Keys 下的 Application ID 和 Admin API Key）。配置好之后，还需要通过 Composer 包管理器安装 Algolia PHP SDK：

```
composer require algolia/algoliasearch-client-php
```

配置

配置模型索引

每个 Eloquent 模型都是通过给定的搜索“索引”进行同步，该索引包含了所有可搜索的模型记录，换句话说，你可以将索引看作是一个 MySQL 数据表。默认情况下，每个模型都会被持久化到与模型对应表名（通常是模型名称的复数形式）相匹配的索引中，不过，你可以通过重写模型中的 `searchableAs` 方法来覆盖这一默认设置：

```
<?php

namespace App;

use Laravel\Scout\Searchable;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    use Searchable;

    /**
     * 获取模型的索引名称.
     *
     * @return string
     */
    public function searchableAs()
    {
        return 'posts_index';
    }
}
```

配置搜索数据

默认情况下，模型以完整的 `toArray` 格式持久化到搜索索引，如果你想要自定义被持久化到搜索索引的数据，可以重写模型上的 `toSearchableArray` 方法：

```
<?php

namespace App;

use Laravel\Scout\Searchable;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    use Searchable;

    /**
     * 获取模型的索引数据数组
     *
     * @return array
     */
    public function toSearchableArray()
    {
        $array = $this->toArray();

        // 自定义数组...

        return $array;
    }
}
```

索引

批量导入

如果将 Scout 安装到了已存在的项目，可能该项目之前已经有了可以导入搜索驱动的数据库记录，Scout 提供了 Artisan 命令 `import` 用于导入所有已存在的数据到搜索索引：

```
php artisan scout:import "App\Post"
```

```
localhost:laravel55 sunqiang$ php artisan scout:import "App\Post"
Imported [App\Post] models up to ID: 4
All [App\Post] records have been imported.
```

导入成功后，我们在 Algolia 后台就可以看到导入成功的索引数据：

添加记录

添加 `Laravel\Scout\Searchable` trait 到模型之后，剩下需要做的就是保存模型实例，然后该实例会自动被添加到模型索引，如果你配置了 Scout 使用队列，该操作会被推送到队列在后台执行：

```
$post = new App\Post;

$post->title = 'Scout 是什么';
$post->content = 'Scout 是 Laravel 官方提供的全文搜索解决方案';
$post->user_id = 1;

$post->save();
```

模型数据保存之后，也会通过 Scout 同步到 Algolia：

通过查询添加

如果你想要通过 Eloquent 查询添加模型集合到搜索索引，可以在 Eloquent 查询之后追加 `searchable()` 方法调用。`searchable()` 方法会分组块进行查询并将结果添加到搜索索引。再次强调，如果你配置了 Scout 使用队列，所有的组块查询会被推送到队列在后台进行：

```
// 通过 Eloquent 查询添加...
App\Post::where('user_id', '>', 10)->searchable();

// 还可以通过关联关系添加记录...
$user->posts()->searchable();
```

```
// 还可以通过集合添加记录...
```

```
$posts->searchable();
```

`searchable` 方法会进行“`upsert`”操作，换句话说，如果模型记录已经存在于索引，则会被更新，如果不存在，才会被添加。

更新记录

要更新支持搜索的模型，只需更新模型实例的属性并保存模型到数据库。Scout 会自动持久化更新到搜索索引：

```
$post = App\Post::find(2);

$post->title = '学院君是谁';

$post->content = '学院君创建了 Laravel 学院，故而得名';

$post->save();
```

去 Algolia 查看索引数据，已更新：

id:	2
user_id:	5
title:	"学院君是谁"
content:	"学院君创建了 Laravel 学院，故而得名"
created_at:	"2018-03-01 15:55:42"
updated_at:	"2018-03-01 16:13:17"
objectID:	"2"



还可以使用模型查询提供的 `searchable` 方法更新模型集合，如果模型在搜索索引中不存在，则会被创建：

```
// 通过 Eloquent 查询更新...
App\Post::where('user_id', '>', 10)->searchable();

// 还可以通过关联关系更新...
$user->posts()->searchable();

// 还可以通过集合更新...
$posts->searchable();
```

删除记录

要从索引中删除记录，只需从数据库中删除对应记录即可，这种删除方式甚至兼容软删除模型：

```
$post = App\Post::find(1);

$post->delete();
```

如果你在删除记录前不想获取模型，可以使用模型查询实例或集合上的 `unsearchable` 方法：

```
// 通过 Eloquent 查询移除...
App\Post::where('user_id', '>', 10)->unsearchable();

// 还可以通过关联关系移除...
$user->posts()->unsearchable();

// 还可以通过集合移除...
$posts->unsearchable();
```

暂停索引

有时候你需要在不同步模型数据到搜索索引的情况下执行批量的 Eloquent 操作，可以通过 `withoutSyncingToSearch` 方法来实现。该方法接收一个立即被执行的回调，该回调中出现的所有模型操作都不会同步到搜索索引：

```
App\Post::withoutSyncingToSearch(function () {
```

```
// Perform model actions...
});
```

搜索

你可以通过 `search` 方法来搜索一个模型，该方法接收一个用于搜索模型的字符串，然后你还需要在这个搜索查询上调用一个 `get` 方法来获取与给定搜索查询相匹配的 Eloquent 模型：

```
$posts = App\Post::search('学院')->get();
```

由于 Scout 搜索返回的是 Eloquent 模型集合，你甚至可以直接从路由或控制器中返回结果，它们将被自动转换为 JSON 格式：

```
use Illuminate\Http\Request;

Route::get('/search', function (Request $request) {
    return App\Post::search($request->search)->get();
});
```

搜索「学院」的话，返回两条搜索结果：

```
[
  {
    "id": 2,
    "user_id": 5,
    "title": "学院君是谁",
    "content": "学院君创建了Laravel学院，故而得名",
    "created_at": "2018-03-01 15:55:42",
    "updated_at": "2018-03-01 16:13:17"
  },
  {
    "id": 4,
    "user_id": 3,
    "title": "欢迎来到Laravel学院",
    "content": "Laravel学院致力于提供优质Laravel中文学习资源",
    "created_at": "2018-03-01 15:55:42",
    "updated_at": "2018-03-01 15:55:42"
  }
]
```

如果你想要获取原生搜索结果而不是转化后的 Eloquent 模型，可以使用 `raw` 方法：

```
$posts = App\Post::search('学院')->raw();
```

返回数据如下：

```

{
  "hits": [
    {
      "id": 2,
      "user_id": 5,
      "title": "学院君是谁",
      "content": "学院君创建了Laravel学院，故而得名",
      "created_at": "2018-03-01 15:55:42",
      "updated_at": "2018-03-01 16:13:17",
      "objectID": "2",
      "_highlightResult": { ... } // 4 items
    },
    {
      "id": 4,
      "user_id": 3,
      "title": "欢迎来到Laravel学院",
      "content": "Laravel学院致力于提供优质Laravel中文学习资源",
      "created_at": "2018-03-01 15:55:42",
      "updated_at": "2018-03-01 15:55:42",
      "objectID": "4",
      "_highlightResult": { ... } // 4 items
    }
  ],
  "nbHits": 2,
  "page": 0,
  "nbPages": 1,
  "hitsPerPage": 20,
  "processingTimeMS": 1,
  "exhaustiveNbHits": true,
  "query": "学院",
  "params": "query=%E5%AD%A6%E9%99%A2"
}

```

搜索查询使用模型类的 `searchAs` 方法指定的索引进行查询。不过，你也可以使用 `within` 方法指定一个自定义的索引进行搜索：

```
$posts = App\Post::search('学院')
  ->within('users_index')
  ->get();
```

where 子句

Scout 允许你添加简单的 `where` 子句到搜索查询，目前，这些子句仅支持简单的数值相等检查，由于搜索索引不是关系型数据库，更多高级的 `where` 子句暂不支持：

```
$posts = App\Post::search('学院')->where('user_id', 1)->get();
```

分页

除了获取模型集合之外，还可以使用 `paginate` 方法对搜索结果进行分页，该方法返回一个 `Paginator` 实例 —— 就像你对传统 Eloquent 查询进行分页一样：

```
$posts = App\Post::search('学院')->paginate();
```

返回结果如下：

```

{
    "current_page": 1,
    "data": [
        {
            "id": 2,
            "user_id": 5,
            "title": "学院君是谁",
            "content": "学院君创建了Laravel学院，故而得名",
            "created_at": "2018-03-01 15:55:42",
            "updated_at": "2018-03-01 16:13:17"
        },
        {
            "id": 4,
            "user_id": 3,
            "title": "欢迎来到Laravel学院",
            "content": "Laravel学院致力于提供优质Laravel中文学习资源",
            "created_at": "2018-03-01 15:55:42",
            "updated_at": "2018-03-01 15:55:42"
        }
    ],
    "first_page_url": "http://laravel55.dev/test/scout?query=%E5%AD%A6%E9%99%A2&page=1",
    "from": 1,
    "last_page": 1,
    "last_page_url": "http://laravel55.dev/test/scout?query=%E5%AD%A6%E9%99%A2&page=1",
    "next_page_url": null,
    "path": "http://laravel55.dev/test/scout",
    "per_page": 15,
    "prev_page_url": null,
    "to": 2,
    "total": 2
}

```

你可以通过传入数量作为 `paginate` 方法的第一个参数来指定每页显示多少个模型:

```
$posts = App\Post::search('学院')->paginate(15);
```

获取结果之后，可以使用 `Blade` 显示结果并渲染分页链接，就像对传统 `Eloquent` 查询进行分页时一样:

```
<div class="container">
    @foreach ($orders as $order)
        {{ $order->price }}
    @endforeach
</div>

{{ $orders->links() }}
```

软删除

如果你的索引模型被软删除但是需要搜索软删除模型，可以设置配置文件 `config/scout.php` 的 `soft_delete` 选项为 `true`:

```
'soft_delete' => true,
```

该配置项被设置为 `true` 时，Scout 将不会从搜索索引中移除软删除模型。取而代之地，将会在索引记录上设置一个隐藏的 `_soft_deleted` 属性。然后，你可以使用 `withTrashed` 或 `onlyTrashed` 方法在搜索时获取软删除记录:

```
// Include trashed records when retrieving results...
$orders = App\Order::withTrashed()->search('Star Trek')->get();

// Only include trashed records when retrieving results...
$orders = App\Order::onlyTrashed()->search('Star Trek')->get();
```

注：使用 `forceDelete` 永久删除软删除模型后，Scout 会自动将其从搜索索引中移除。

自定义引擎

编写引擎

如果某个内置的 Scout 搜索引擎不满足你的需求，可以编写自定义的引擎并将其注册到 Scout，自定义的引擎需要继承自抽象类 `Laravel\Scout\Engines\Engine`，该抽象类包含了 7 个自定义引擎必须实现的方法:

```
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($results, $model);
abstract public function getTotalCount($results);
```

这些方法的实现可以参考 `Laravel\Scout\Engines\AlgoliaEngine` 类，这个类为我们学习如何在自定义引擎中实现这些方法提供了最佳范本。

注册引擎

编写好自定义引擎之后，可以通过 Scout 引擎管理器提供的 `extend` 方法将其注册到 Scout。你需要在 `AppServiceProvider`（或者其他服务提供者）的 `boot` 方法中调用这个 `extend` 方法。例如，如果你编写了 `MySqlSearchEngine`，可以这样注册：

```
use Laravel\Scout\EngineManager;

/**
 * 启动任意应用服务.
 *
 * @return void
 */
public function boot()
{
    resolve(EngineManager::class)->extend('mysql', function () {
        return new MySqlSearchEngine;
    });
}
```

引擎被注册之后，可以在配置文件 `config/scout.php` 中将其设置为 Scout 默认的驱动：

```
'driver' => env('SCOUT_DRIVER', 'mysql'),
```

Laravel Socialite

简介

除了传统的基于表单的登录认证外，Laravel 还可以通过 `Laravel Socialite` 提供 OAuth 认证，目前支持的认证驱动包括 Facebook、Twitter、Google、LinkedIn、GitHub 和 Bitbucket。

注：其他平台的驱动可以在社区驱动的 `Socialite 提供者` 网站上找到。

安装

要使用 Socialite，首先需要通过 Composer 安装扩展包：

```
composer require laravel/socialite
```

配置

在使用 Socialite 之前，还需要为应用用到的 OAuth 服务添加认证信息，这些认证信息位于配置文件 `config/services.php`，而且对应 key 必须为 `facebook`、`twitter`、`linkedin`、`google`、`github` 或 `bitbucket`，配置哪些 key 取决于应用需要的提供者。例如：

```
'github' => [
    'client_id' => env('GITHUB_CLIENT_ID'),           // Your GitHub Client ID
    'client_secret' => env('GITHUB_CLIENT_SECRET'), // Your GitHub Client Secret
    'redirect' => 'http://your-callback-url',
],
```

上述 `GITHUB_CLIENT_ID` 和 `GITHUB_CLIENT_SECRET` 可以通过在 <https://github.com/settings/developers> 页面新增 OAuth 应用获取到：

Laravel学院测试

nonfu owns this application.

You can list your application in the [GitHub Marketplace](#) so that other users can discover it. [List this application in the Marketplace](#)

0 users

Client ID

Client Secret

[Revoke all user tokens](#) [Reset client secret](#)

此外，应用的表单信息也要按照自己的应用正确填写：

Application logo

Drag & drop

Upload new logo

You can also drag and drop a picture from your computer.

Application name

Something users will recognize and trust

Homepage URL

The full URL to your application homepage

Application description

This is displayed to all users of your application

Authorization callback URL

Your application's callback URL. Read our [OAuth documentation](#) for more information.

[Update application](#)
[Delete application](#)

注：如果 `redirect` 配置项包含的是相对路径，系统会自动将其转化为完整 URL。

路由

接下来，准备开始认证用户！你需要两个路由：一个用于重定向用户到 OAuth 提供者，另一个用于认证后获取来自提供者的回调。我们使用 `Socialite` 门面访问 Socialite：

```
<?php
namespace App\Http\Controllers\Auth;
use Socialite;
```

```

class LoginController extends Controller
{
    /**
     * 将用户重定向到 Github 认证页面
     *
     * @return Response
     */
    public function redirectToProvider()
    {
        return Socialite::driver('github')->redirect();
    }

    /**
     * 从 Github 获取用户信息.
     *
     * @return Response
     */
    public function handleProviderCallback()
    {
        $user = Socialite::driver('github')->user();

        dd($user->token);
    }
}

```

`redirectToProvider` 方法将用户发送到 OAuth 提供者, `handleProviderCallback` 方法读取请求信息并从提供者中获取用户信息。

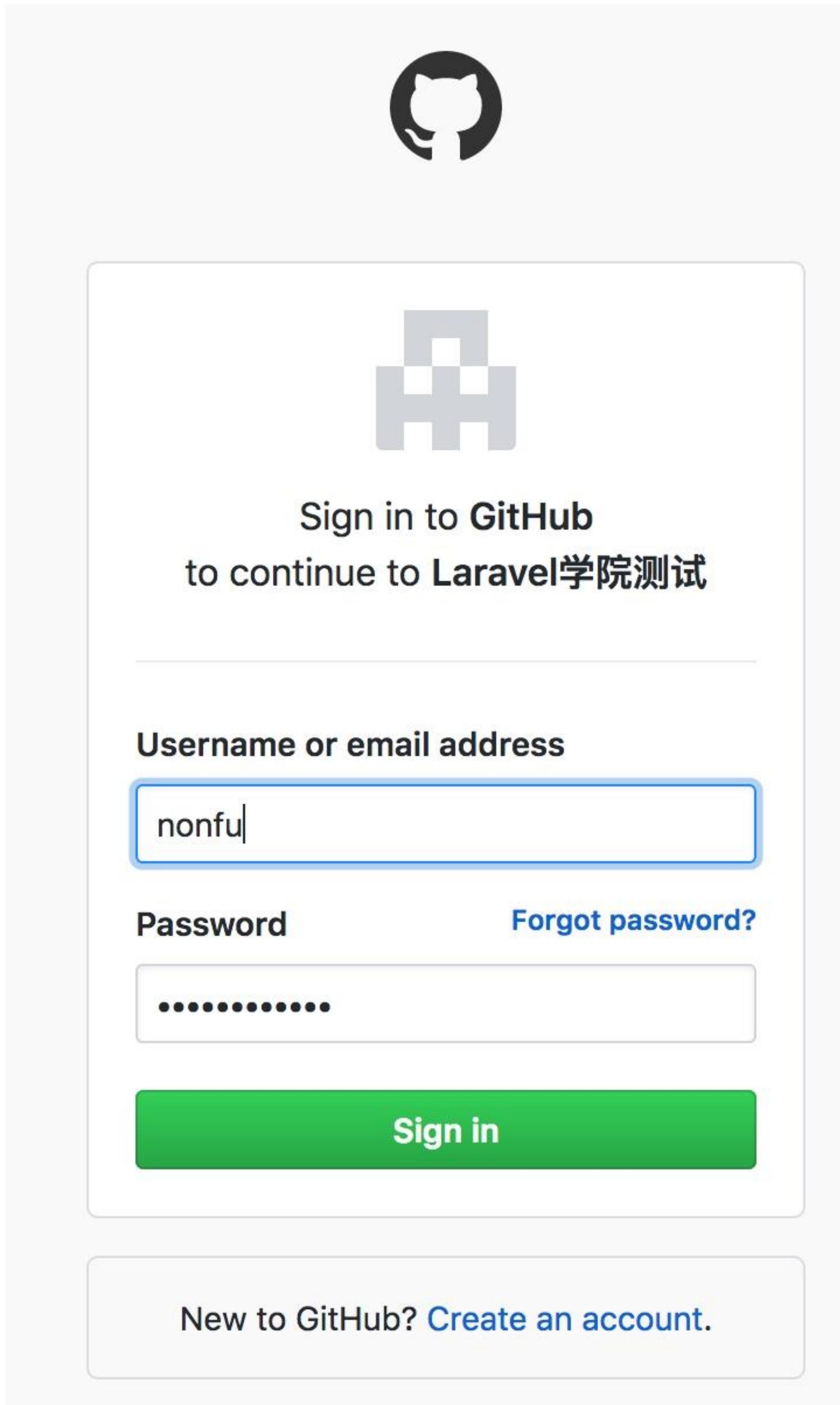
当然, 你需要定义路由到控制器方法:

```

Route::get('login/github', 'Auth\LoginController@redirectToProvider');
Route::get('login/github/callback', 'Auth\LoginController@handleProviderCallback');

```

这样, 我们在访问 `login/github` 路由时, 页面就会跳转到 Github 认证界面:



登录授权之后，页面就会重定向到 `login/github/callback` 页面打印用户信息：

```
User {#337 ▼
  +token: "a...9551c1280043a...ead"
  +refreshToken: null
  +expiresIn: null
  +id: 1...
  +nickname: "nonfu"
  +name: "学院君"
  +email: "yaojinbu@163.com"
  +avatar: "https://avatars1.githubusercontent.com/u/8003210?v=4"
  +user: array:30 [▶]
}
```

可选参数

很多 OAuth 提供者在重定向请求中支持可选参数，要在请求中包含可选参数，可以通过一个关联数组调用 `with` 方法：

```
return Socialite::driver('google')
    ->with(['hd' => 'example.com'])
    ->redirect();
```

注：使用 `with` 方法的时候，注意不要传递保留关键字作为数组的 `key`，例如 `state` 或 `response_type`。

访问作用域

在重定向用户之前，还可以使用 `scopes` 方法在请求上添加额外的“作用域”，该方法会合并所有提供的作用域

```
return Socialite::driver('github')
    ->scopes(['read:user', 'public_repo'])
    ->redirect();
```

你可以使用 `setScopes` 方法覆盖所有已存在的作用域：

```
return Socialite::driver('github')
    ->setScopes(['read:user', 'public_repo'])
    ->redirect();
```

无状态认证

`stateless` 方法可用于禁止会话状态验证。这在添加 Socialite 认证到某个 API 时很有用：

```
return Socialite::driver('google')->stateless()->user();
```

获取用户信息

有了用户实例之后，就可以获取更多用户详情：

```
$user = Socialite:::driver('github')->user();

// OAuth Two Providers
$token = $user->token;
$refreshToken = $user->refreshToken; // not always provided
$expiresIn = $user->expiresIn;

// OAuth One Providers
$token = $user->token;
$tokenSecret = $user->tokenSecret;

// All Providers
$user->getId();
$user->getNickname();
$user->getName();
$user->getEmail();
$user->getAvatar();
```

获取用户信息后，我们可以将其保存到 `users` 数据表，然后使用 `Auth::login` 方法将用户登录到应用。下次同一用户使用 Github 登录，我们从提供者获取用户信息后先通过 `github_id` 从数据库中查找该用户，找到后即登录。

从 Token (OAuth2) 中获取用户信息

如果你已经有了某个用户的有效访问令牌，就可以使用 `userFromToken` 方法获取其信息：

```
$user = Socialite::driver('github')->userFromToken($token);
```

从 **Token (OAuth1)** 和 **Secret** 中获取用户信息
如果你已经有了某个用户的有效 token/secret，就可以使用 `userFromTokenAndSecret` 方法获取其信息：

至此， Laravel 官方文档已全部完结，想了解更多 Laravel 动态及教程，请访问 [Laravel 学院](#)。