Table of Contents

Laravel5 中文文档

概述

这是 Laravel5 文档的中文仓库，由 `laravel-china` 发起并维护这个项目。
您可在 laravel-china.org 查看在线文档，或是在 这里 下载离线版本。

如果您在阅读的过程中发现问题，欢迎提交 issue 或 pull request。

推广

您可以在您的朋友圈中推广 laravel-china.org 和 PHPHub（一个积极向上的 PHP & Laravel 开发者社区）来支持我们。

感谢

感谢 台湾 Laravel 项目维护者 @appleboy 和他的队友, 此次的翻译是在台湾同学们的劳动上再次翻译为简体中文, 感谢 PHPHub 维护者 @NauxLiu 提供的方法（详细讨论见此处）。 感谢所有为此文档做出贡献的同学。

贡献列表：
@lifesign
@NauxLiu
@summerblue
@appleboy
@zhangxuanming
@zhuzhichao
@frankyang4

发行说明

Laravel 5.0

Laravel 5.0 在默认的项目上引进了新的应用程序架构。新的架构提供了更好的功能来构建健壮的 Laravel 应用程序，以及在应用程序中全面采用新的自动加载标准（ PSR-4 ）。首先，来查看一些主要变更：

新的目录结构

旧的 `app/models` 目录已经完全被移除。相对的，你所有的代码都放在 `app` 目录下，以及默认使用 `App` 命名空间。这个默认的命名空间可以使用新的 `app:name` Artisan 命令来快速更改。

控制器（ controller ），中间件（ middleware ），以及请求（ requests，Laravel 5.0 中新型态的类别），现在都存放在 `app/Http` 的对应目录下，因为他们都与应用程序的 HTTP 传输层相关。除了一个路由设置的文件外，所有的中间件现在都拆分开成单独的类文件。

新的 `app/Providers` 目录取代了旧版 Laravel 4.x `app/start` 里的文件。这些服务提供者有很多启动应用程序相关的方法，像是错误处理，日志记录，路由加载，以及更多。当然，你可以自由的建立新的服务提供者到应用程序。

应用程序的语言文件和视图都移到 `resources` 目录下。

Contracts

所有 Laravel 主要组件实现所用的接口都放在 `illuminate/contracts` 项目下。这个项目没有其他的外部依赖。这些方便、集成的接口，可以让你用来让依赖注入变得低耦合，将可以简单作为 Laravel Facades 的替代选项。

更多关于 contracts 的信息，参考完整文档。

路由缓存

如果你的应用程序全部都是使用控制器路由，你可以使用新的 `route:cache` Artisan 命令大幅度地加快路由注册。这对于有 100 个以上路由规则的应用程序很有用，可以大幅度地加快应用程序这部分的处理速度。

路由中间件（ Middleware ）

除了像 Laravel 4 风格的路由「过滤器（ filters ）」，Laravel 5 现在也支持 HTTP 中间件，而原本的认证和 CSRF 「过滤器」已经改写成中间件。中间件提供了单一、一致的接口取代了各种过滤器，让你在请求进到应用程序前，可以方便地检查甚至拒绝请求。

更多关于中间件的信息，参考完整文档。

控制器方法依赖注入

除了之前有的控制器依赖注入，你现在可以在控制器方法使用类型提示（ type-hint ）进行依赖注入。服务容器会自动注入依赖，即使路由包含了其他参数也不成问题：

```
public function createPost(Request $request, PostRepository $posts)
{
    //
}
```

认证基本架构

用户注册，认证，以及重设密码的控制器现在已经默认含括了，包含相对应的视图，放在 `resources/views/auth`。除此之外，「users」数据表迁移也已经默认存在框架中了。这些简单的资源，可以让你快速开发应用程序的业务逻辑，而不用陷在撰写认证模板的泥潭上。认证相关的视图可以经由 `auth/login` 以及 `auth/register` 路由访问。 `App\Services\Auth\Registrar` 服务会负责处理用户认证和添加的相关逻辑。

事件对象

你现在可以将事件定义成对象，而不是仅使用字串。例如，查看以下的事件：

```
class PodcastWasPurchased {

    public $podcast;

    public function __construct(Podcast $podcast)
    {
        $this->podcast = $podcast;
    }

}
```

这个事件可以像一般使用那样被派发：

```
Event::fire(new PodcastWasPurchased($podcast));
```

当然，你的事件处理会收到事件的对象而不是数据的列表：

```
class ReportPodcastPurchase {

    public function handle(PodcastWasPurchased $event)
    {
        //
    }

}
```

更多关于使用事件的信息，参考完整文档。

命令（ **Commands** ）、队列（ **Queueing** ）

除了 Laravel 4 形式的队列任务，Laravel 5 以简单的命令对象作为队列任务。这些命令放在 `app/Commands` 目录下。下面是个简单的命令：

```
class PurchasePodcast extends Command implements SelfHandling, ShouldBeQueued {

    use SerializesModels;

    protected $user, $podcast;

    /**
     * Create a new command instance.
     *
     * @return void
     */
    public function __construct(User $user, Podcast $podcast)
    {
        $this->user = $user;
        $this->podcast = $podcast;
    }

    /**
     * Execute the command.
     *
     * @return void
     */
    public function handle()
    {
        // Handle the logic to purchase the podcast...

        event(new PodcastWasPurchased($this->user, $this->podcast));
    }

}
```

Laravel 的基底控制器使用了新的 `DispatchesCommands` trait，让你可以简单的派发命令执行。

```
$this->dispatch(new PurchasePodcastCommand($user, $podcast));
```

当然，你也可以将命令视为同步执行（而不会被放到队列里）的任务。事实上，使用命令是个好方式，让你可以封装应用程序需要执行的复杂任务。更多相关的信息，参考 command bus 文档。

数据库队列

`database` 队列驱动现在已经包含在 Laravel 中了，提供了简单的本地端队列驱动，让你除了数据库相关软件外不需安装其他套件。

Laravel 调用（ **Scheduler** ）

过去，开发者可以产生 Cron 设置，用以调用所有他们想要执行的命令行指令。然而，这是件很头痛的事情，你的命令行调用不再属于版本控制的一部分，而你必须 SSH 到服务器里加入 Cron 设置。

为了让生活变得简单点, 让你可以流畅而且具有表达性的在 Laravel 里面定义你的命令调用，而且服务器只

需要单一个 Cron 设置。

它会看起来如下：

```
$schedule->command('artisan:command')->dailyAt('15:00');
```

当然，快参考完整文档学习所有调用相关知识。

Tinker、Psysh

`php artisan tinker` 命令现在使用 Justin Hileman 的 Psysh，一个 PHP 更强大的 REPL。如果你喜欢 Laravel 4 的 Boris，你也会喜欢上 Psysh。更好的是，它可以跑在 Windows！要开始使用，只要输入：

```
php artisan tinker
```

DotEnv

比起一堆令人困惑的、嵌套的环境配置文件目录，Laravel 5 现在使用了 Vance Lucas 的 DotEnv。这个套件提供了超级简单的方式管理配置文件，并且让 Laravel 5 环境检测变得轻松。更多的细节，参考完整的配置文件文档。

Laravel Elixir

Jeffrey Way 的 Laravel Elixir 提供了一个流畅、语义化的接口，可以编译以及合并 assets。如果你曾经在学习 Grunt 或 Gulp 被吓到，不必再害怕了。Elixir 让使用 Gulp 编译 Less、Sass 及 CoffeeScript 变得简单。它甚至可以帮你执行测试！

更多关于 Elixir 的信息，参考完整文档。

Laravel Socialite

Laravel Socialite 是个可选的，Laravel 5.0 以上兼容的套件，提供了无痛的 OAuth 认证。目前 Socialite 支持 Facebook、Twitter、Google 以及 GitHub。它写起来可能像这样：

```
public function redirectForAuth()
{
    return Socialize::with('twitter')->redirect();
}

public function getUserFromProvider()
{
    $user = Socialize::with('twitter')->user();
}
```

不用再花上数小时撰写 OAuth 的认证流程。数分钟就可开始！查看完整文档 里有所有的细节。

Flysystem 集成

Laravel 现在包含了强大的 Flysystem（一个文件系统的抽象类库），提供了无痛的集成，把本地端文件系

统、Amazon S3 和 Rackspace 云存储集成在一起， 有统一且优雅的 API！现在要将文件存到 Amazon S3 相当简单：

```
Storage::put('file.txt', 'contents');
```

更多关于 Laravel 文件系统集成，参考完整文档。

Form Requests

Laravel 5.0 使用了 **form requests**，是继承了 `Illuminate\Foundation\Http\FormRequest` 的类。这些 request 对象可以和控制器方法依赖注入结合，提供一个不需样板的方法，可以验证用户输入。让我们深入点，看一个 `FormRequest` 的示例：

```
<?php namespace App\Http\Requests;

class RegisterRequest extends FormRequest {

    public function rules()
    {
        return [
            'email' => 'required|email|unique:users',
            'password' => 'required|confirmed|min:8',
        ];
    }

    public function authorize()
    {
        return true;
    }

}
```

定义好类后，我们可以在控制器动作里使用类型提示：

```
public function register(RegisterRequest $request)
{
    var_dump($request->input());
}
```

当 Laravel 的服务容器辨别出要注入的类别是个 `FormRequest` 实例，请求会被自动验证。意味着，当你的控制器动作被调用了，你可以安全的假设 HTTP 的请求输入已经被验证过，根据你在 form request 类别里自定的规则。甚至，若这个请求验证不通过，一个 HTTP 重定向（可以自定），会自动发出，错误消息可以被闪存到 session 或是转换成 JSON 返回。表单验证就是如此的简单。更多关于 `FormRequest` 验证，参考文档。

简易控制器请求验证

Laravel 5 基类控制器包含一个 `ValidatesRequests` trait。这个 trait 包含了一个简单的 `validate` 方法可以验证请求。如果对应用程序来说 `FormRequests` 太复杂了，参考这个：

```
public function createPost(Request $request)
{
    $this->validate($request, [
        'title' => 'required|max:255',
        'body' => 'required',
    ]);
}
```

如果验证失败，会抛出异常并且返回对应的 HTTP 回应到浏览器。验证错误信息会被闪存到 session！而如果请求是 AJAX 请求，Laravel 会自动返回 JSON 格式的验证错误信息。

更多关于这个新方法的信息，参考这个文档。

新的 Generators

因采用了新的应用程序默认架构，框架也添加了 Artisan generator 命令。使用 `php artisan list` 查看更多细节。

配置文件缓存

你现在可以在单一文件里缓存所有配置文件了，使用 `config:cache` 命令。

Symfony VarDumper

常用的 `dd` 辅助函数，其可以在除错时印出变量信息，已经升级成使用令人惊艳的 Symfony VarDumper。它提供了颜色标记的输出，甚至数组可以自动缩合。在项目中试试下列代码：

```
dd([1, 2, 3]);
```

Laravel 4.2

此发行版本的完整变更列表可以从一个 4.2 的完整安装下，执行 `php artisan changes` 命令，或者 Github 上的变更纪录。此纪录仅含括主要的强化更新和此发行的变更部分。

> 附注: 在 4.2 发布周期间，许多小的 bug 修正与功能强化被整并至各个 4.1 的子发行版本中。所以最好确认 Laravel 4.1 版本的更新列表。

PHP 5.4 需求

Laravel 4.2 需要 PHP 5.4 以上的版本。此 PHP 更新版本让我们可以使用 PHP 的新功能：traits 来为像是 Laravel 收银台 来提供更具表达力的接口。PHP 5.4 也比 PHP 5.3 带来显著的速度及性能提升。

Laravel Forge

Larvel Forge，一个网页应用程序，提供一个简单的接口去建立管理你云端上的 PHP 服务器，像是 Linode、DigitalOcean、Rackspace 和 Amazon EC2。支持自动化 nginx 设置、SSH 密钥管理、Cron job 自动化、透过 NewRelic & Papertrail 服务器监控、「推送部署」、Laravel queue worker 设置等等。Forge 提供最简单且更实惠的方式来部署所有你的 Laravel 应用程序。

默认 Laravel 4.2 的安装里，`app/config/database.php` 配置文件默认已为 Forge 设置完成，让在平台上的全新应用程序更方便部署。

关于 Laravel Forge 的更多信息可以在官方 Forge 网站上找到。

Laravel Homestead

Laravel Homestead 是一个为部署健全的 Laravel 和 PHP 应用程序的官方 Vagrant 环境。绝大多数的封装包的相依与软件在发布前已经部署处理完成，让封装包可以极快的被启用。Homestead 包含 Nginx 1.6、PHP 5.5.12、MySQL、Postres、Redis、Memcached、Beanstalk、Node、Gulp、Grunt 和 Bower。Homestead 包含一个简单的 `Homestead.yaml` 配置文件，让你在单一个封装包中管理多个 Laravel 应用程序。

默认的 Laravel 4.2 安装中包含的 `app/config/local/database.php` 配置文件使用 Homestead 的数据库作为默认。让 Laravel 初始化安装与设置更为方便。

官方文档已经更新并包含在 Homestead 文档 中。

Laravel 收银台

Laravel 收银台是一个简单、具表达性的资源库，用来管理 Stripe 的订阅帐务。虽然在安装中此组件依然是选用，我们依然将收银台文档包含在主要 Laravel 文档中。此收银台发布版本带来了数个错误修正、多货币支持还有支持最新的 Stripe API。

Queue Workers 常驻程序

Artisan `queue:work` 命令现在支持 `--daemon` 参数让 worker 可以以「常驻程序」启用。代表 worker 可以持续的处理队列工作不需要重启框架。这让一个复杂的应用程序部署过程中，使得 CPU 的使用有显著的降低。

更多关于 Queue Workers 常驻程序信息请详阅 queue 文档。

Mail API Drivers

Laravel 4.2 为 `Mail` 函式采用了新的 Mailgun 和 Mandrill API 驱动。对许多应用程序而言，他提供了比 SMTP 更快也更可靠的方法来递送邮件。新的驱动使用了 Guzzle 4 HTTP 资源库。

软删除 Traits

对于软删除和全作用域更简洁的方案 PHP 5.4 的 `traits` 提供了一个更加简洁的软删除架构和全局作用域，这些新架构为框架提供了更有扩展性的功能，并且让框架更加简洁。

更多关于软删除的文档请见: Eloquent documentation.

更为方便的 认证(auth) & Remindable Traits

得益于 PHP 5.4 traits，我们有了一个更简洁的用户认证和密码提醒接口，这也让 `User` 模型文档更加精简。

"简易分页"

一个新的 `simplePaginate` 方法已被加入到查找以及 Eloquent 查找器中。让你在分页视图中，使用简单的「上一页」和「下一页」链接查找更为高效。

迁移确认

在正式环境中，破坏性的迁移动作将会被再次确认。如果希望取消提示字符确认请使用 `--force` 参数。

Laravel 4.1

完整变更列表

此发行版本的完整变更列表，可以在版本 4.1 的安装中命令行执行 `php artisan changes` 取得，或者浏览 Github 变更档 中了解。其中只记录了该版本比较主要的强化功能和变更。

新的 SSH 组件

一个全新的 `SSH` 组件在此发行版本中登场。此功能让你可以轻易的 SSH 至远程服务器并执行命令。更多信息，可以参阅 SSH 组件文档。

新的 `php artisan tail` 指令就是使用这个新的 SSH 组件。更多的信息，请参阅 `tail` 指令集文档。

Boris In Tinker

如果您的系统支持 Boris REPL，`php artisan thinker` 指令将会使用到它。系统中也必须先行安装好 `readline` 和 `pcntl` 两个 PHP 套件。如果你没这些套件，从 4.0 之后将会使用到它。

Eloquent 强化

Eloquent 添加了新的 `hasManyThrough` 关系链。想要了解更多，请参见 Eloquent 文档。

一个新的 `whereHas` 方法也同时登场，他将允许检索基于关系模型的约束。

数据库读写分离

Query Builder 和 Eloquent 目前透过数据库层，已经可以自动做到读写分离。更多的信息，请参考 文档。

队列排序

队列排序已经被支持，只要在 `queue:listen` 命令后将队列以逗号分隔送出。

失败队列作业处理

现在队列将会自动处理失败的作业，只要在 `queue:listen` 后加上 `--tries` 即可。更多的失败作业处理可以参见 队列文档。

缓存标签

缓存「区块」已经被「标签」取代。缓存标签允许你将多个「标签」指向同一个缓存对象，而且可以清空所有被指定某个标签的所有对象。更多使用缓存标签信息请见 缓存文档。

更具弹性的密码提醒

密码提醒引擎已经可以提供更强大的开发弹性，如：认证密码、显示状态消息等等。使用强化的密码提醒引擎，更多的信息 请参阅文档。

强化路由引擎

Laravel 4.1 拥有一个完全重新编写的路由层。API 一样不变。然而与 4.0 相比，速度快上 100%。整个引擎大幅的简化，且对于路由表达式的编译大大减少对 Symfony Routing 的依赖。

强化 **Session** 引擎

此发行版本中，我们亦发布了全新的 Session 引擎。如同路由增进的部分，新的 Session 曾更加简化且更快速。我们不再使用 Symfony 的 Session 处理工具，并且使用更简单、更容易维护的客制化解法。

Doctrine DBAL

如果你有在你的迁移中使用到 `renameColumn`，之后你必须在 `composer.json` 里加 `doctrine/dbal` 进相依套件中。此套件不再默认包含在 Laravel 之中。

升级向导

从 4.2 升级到 5.0

全新安装，然后迁移

推荐的升级方式是建立一个全新的 Laravel `5.0` 项目，然后复制您在 `4.2` 的文件到此新的应用程序，这将包含控制器、路由、Eloquent 模型、Artisan 命令（Asset）、资源和关于此应用程序的其他特定文件。

最开始，安装新的 Laravel 5 应用程序到新的本地目录下，我们将详细探讨迁移各部分的过程。

Composer 依赖与组件

别忘了将任何附加于 Composer 的依赖组件加入 5.0 应用程序内，包含第三方代码(例如 SDKs)。

部分组件也许不兼容刚发布的 Laravel 5 版本，请向组件管理者确认该组件支持 Laravel 5 的版本，当您在 Composer 内加入任何组件，请执行 `composer update`。

命名空间

默认情况下，Laravel 4 没有在应用程序的源码中使用命名空间，所以，举例来说，所有的 Eloquent 模型和控制器仅存在「全局」的命名空间下，为了更快速的迁移，Laravel 5 也允许您可以将这些类别一样保留在「全局」的命名空间。

设置文件

迁移环境变量

复制新的 `.env.example` 文件到 `.env`，在 `5.0` 这相当于原本的 `.env.php`。像是您的 `APP_ENV` 和 `APP_KEY` (您的加密钥匙)、数据库认证和您的缓存驱动与 session 驱动。

此外，复制原先自定义的 `.env.php` 文件，并修改为 `.env` (本机环境的真实设置值) 和 `.env.example` (给其他团队成员的示例)。

更多关于环境设置值，请见完整文档。

> 注意: 在部署 Laravel 5 应用程序之前，您需要在正式主机上放置 `.env` 文件并设置适当的值。

设置文件

Laravel 5.0 不再使用 `app/config/{environmentName}/` 目录结构来提供对应该环境的设置文件，取而代之的是，将环境对应的设置值复制到 `.env`，然后在设置文件使用 `env('key', 'default value')` 来访

问，您可以在 `config/database.php` 文件内看到相关范例。

将设置文件放在 `config/` 目录下，来表示所有环境共用的设置文件，或是在文件内使用 `env()` 来取得对应该环境的设置值。

请记住，若您在 `.env` 文件内增加 key 值，同时也要对应增加到 `.env.example` 文件中，这将可以帮助团队成员修改他们的 `.env` 文件。

路由

复制原本的 `routes.php` 文件到 `app/Http/routes.php`。

控制器

下一步，请将所有的控制器复制到 `app/Http/Controllers` 目录，既然在本指南中我们不打算迁移到完整的命名空间，请将 `app/Http/Controllers` 添加到 `composer.json` 的 `classmap`，接下来，您可以从 `app/Http/Controllers/Controller.php` 基础抽象类中移除命名空间，并确认迁移过来的控制器要继承这个基础类。

在 `app/Providers/RouteServiceProvider.php` 文件中，将 `namespace` 属性设置为 `null`。

路由过滤器

将过滤器绑定从原来的 `app/filters.php` 复制到 `app/Providers/RouteServiceProvider.php` 的 `boot()` 方法中，并在 `app/Providers/RouteServiceProvider.php` 加入 `use Illuminate\Support\Facades\Route;` 来继续使用 `Route` Facade。

您不需要移动任何 Laravel 4.0 默认的过滤器，像是 `auth` 和 `csrf`。他们已经内置，只是换作以中间件形式出现。那些在路由或控制器内有参照到旧有的过滤器 (例如 `['before' => 'auth']`) 请修改参照到新的中间件 (例如 `['middleware' => 'auth'].`)

Laravel 5 并没有将过滤器移除，您一样可以使用 `before` 和 `after` 绑定和使用您自定义的过滤器。

全局 CSRF

默认情况下，所有路由都会使用CSRF 保护。若想关闭他们，或是在指定在特定路由开启，请移除 `App\Http\Kernel` 中 `middleware` 数组内的这一行：

```
'App\Http\Middleware\VerifyCsrfToken',
```

如果您想在其他地方使用它，加入这一行到 `$routeMiddleware`：

```
'csrf' => 'App\Http\Middleware\VerifyCsrfToken',
```

现在，您可于路由内使用 `['middleware' => 'csrf']` 即可个别添加中间件到路由/控制器。了解更多关于中间件，请见完整文档。

Eloquent 模型

你可以建立新的 `app/Models` 目录来放置所有 Eloquent 模型。并且同样的，在 `composer.json` 将此目录添加到 `classmap` 内。

在模型内加入 `SoftDeletingTrait` 来使用 `Illuminate\Database\Eloquent\SoftDeletes`.

Eloquent 缓存

Eloquent 不再提供 `remember` 方法来缓存查询。现在你需要手动使用 `Cache::remember` 方法快速缓存。了解更多关于缓存，请见完整文档。

会员认证模型

要使用 Laravel 5 的会员认证系统，请遵循以下指引来升级您的 `User` 模型：

从 **`use`** 区块删除以下内容：

```
use Illuminate\Auth\UserInterface;
use Illuminate\Auth\Reminders\RemindableInterface;
```

添加以下内容到 **`use`** 区块：

```
use Illuminate\Auth\Authenticatable;
use Illuminate\Auth\Passwords\CanResetPassword;
use Illuminate\Contracts\Auth\Authenticatable as AuthenticatableContract;
use Illuminate\Contracts\Auth\CanResetPassword as CanResetPasswordContract;
```

移除 **UserInterface** 和 **RemindableInterface** 接口。

实现以下接口：

```
implements AuthenticatableContract, CanResetPasswordContract
```

在类中声明引入以下 **traits**：

```
use Authenticatable, CanResetPassword;
```

如果你引入了上面的 **traits**，从 **use** 区块和类声明中移除 **`Illuminate\Auth\Reminders\RemindableTrait`** 和 **`Illuminate\Auth\UserTrait`**

Cashier 的用户需要的修改

Laravel Cashier 的 trait 和接口名称已作修改。trait 请改用 `Laravel\Cashier\Billable` 取代 `BillableTrait`。接口请改用 `Laravel\Cashier\Contracts\Billable` 取代 `Larave\Cashier\BillableInterface`。不需要修改任何方法。

Artisan 命令

将所有的命令从旧的 `app/commands` 目录移到新的 `app/Console/Commands` 目录。接下来，把 `app/Console/Commands` 目录添加到 `composer.json` 的 `classmap` 中。

然后，复制 Artisan 命令列表从 `start/artisan.php` 到 `app/Console/Kernel.php` 文件的 `command` 数组内。

数据库迁移和数据填充

如果在您的数据库内已经有 users 表，请移除 Laravel 5 内置的两个迁移文件。

将所有的迁移文件从旧的 `app/database/migrations` 目录复制到新的 `database/migrations` 。所有的数据填充文件也要从 `app/database/seeds` 复制到 `database/seeds` 。

全局 IoC 绑定

若您在 `start/global.php` 有绑定任何 IoC，请将它们复制到 `app/Providers/AppServiceProvider.php` 内的 `register` 方法，您可能需要引入 `App` facade。

你可以选择将这些绑定，依照类型拆分到不同的服务提供者中。

视图

将所有的视图从旧的 `app/views` 复制到新的 `resources/views` 目录内。

Blade 标签修改

从安全的角度考虑，Laravel 5.0 会过滤所有输出，不论您使用 `{{ }}` 或 `{{{ }}}` 标签。您可以使用 `{!! !!}` 新的标签来取消输出过滤。请务必 确定 输出内容是安全地才使用 `{!! !!}` 标签。

不过，如果您 仍然必须 使用旧的 Blade 语法，请在 `AppServiceProvider@register` 开头加入以下内容：

```
\Blade::setRawTags('{{', '}}');
\Blade::setContentTags('{{{', '}}}');
\Blade::setEscapedContentTags('{{{', '}}}');
```

但是轻易不要这样做，这可能使您的应用进程更加容易受到 XSS 攻击，并且用 `{{--` 来注释代码将不再起作用。

多语言配置文件

将所有的多语言配置文件从旧的 `app/lang` 目录复制到新的 `resources/lang` 目录。

公开目录

将 `4.2` 版公共目录内的资源复制到新应用程序内的 `public` 目录。并确认保留 `5.0` 版的 `index.php` 文件。

测试

将所有的测试文件从旧的 `app/tests` 复制到 `tests` 目录。

各式各样的文件

复制项目内其他各式各样的文件，例如：`.scrutinizer.yml`，`bower.json` 以及其他类似工具的设置文件。

您可以将 Sass，Less 或 CoffeeScript 移动到任何您想放置的地方。 `resources/assets` 目录是一个不错的默认位置。

表单和 HTML 辅助函数

如果您使用表单或 HTML 辅助函数，您将会看到以下错误 `class 'Form' not found` 或 `class 'Html' not found` 。Form 类以及 HTML 辅助函数在 Laravel 5.0 中已经废弃了；不过，这里有一些替代方法，比如基于社区驱动的，由 Laravel Collective 维护。

比如，你可以在 `composer.json` 文件中的 `require` 区块增加 `"laravelcollective/html": "~5.0"` 。

您也需要添加表单和 HTML 的 facades 以及服务提供者。 编辑 `config/app.php` 文件，添加此行到 'providers' 数组内：

```
'Collective\Html\HtmlServiceProvider',
```

接着，添加以下到 'aliases' 数组内：

```
'Form' => 'Collective\Html\FormFacade',
'Html' => 'Collective\Html\HtmlFacade',
```

缓存管理员

如果您的程序注入 `Illuminate\Cache\CacheManager` 来取得非 Facade 版本的 Laravel 缓存，请改用 `Illuminate\Contracts\Cache\Repository` 注入。

分页

请将所有的 `$paginator->links()` 用 `$paginator->render()` 取代。

Beanstalk 队列

Laravel 5.0 使用 `"pda/pheanstalk": "~3.0"` 取代原本的 `"pda/pheanstalk": "~2.1"` 。

Remote

Remote 组件已不再使用。

工作区

工作区组件已不再使用。

从 4.1 升级到 4.2

PHP 5.4+

Laravel 4.2 需要 PHP 5.4.0 以上。

默认加密

增加一个新的 `cipher` 选项在你的 `app/config/app.php` 设置文件中。其选项值应为 `MCRYPT_RIJNDAEL_256`。

```
'cipher' => MCRYPT_RIJNDAEL_256
```

该设置可用于设置所使用的 Laravel 加密工具的默认加密方法。

> 附注: 在 Laravel 4.2，默认加密方法为 `MCRYPT_RIJNDAEL_128` (AES)，被认为是最安全的加密。必须将加密改回 `MCRYPT_RIJNDAEL_256` 来解密在 Laravel <= 4.1 下加密的 cookies/values

软删除模型现在改使用特性

如果你在模型下有使用软删除，现在 `softDeletes` 的属性已经被移除。你现在要使用 `SoftDeletingTrait` 如下：

```
use Illuminate\Database\Eloquent\SoftDeletingTrait;

class User extends Eloquent {
    use SoftDeletingTrait;
}
```

你一样必须手动增加 `deleted_at` 字段到你的 `dates` 属性中：

```
class User extends Eloquent {
    use SoftDeletingTrait;

    protected $dates = ['deleted_at'];
}
```

而所有软删除的 API 使用方式维持相同。

> 附注: `SoftDeletingTrait` 无法在基本模型下被使用。他只能在一个实际模型类别中使用。

视图 / 分页 / 环境 类别改名

如果你直接使用 `Illuminate\View\Environment` 或 `Illuminate\Pagination\Environment` 类别，请更新你的代码将其改为参照 `Illuminate\View\Factory` 和 `Illuminate\Pagination\Factory`。改名后的这两个类别更可以代表他们的功能。

Additional Parameter On Pagination Presenter

如果你扩展了 `Illuminate\Pagination\Presenter` 类别，抽象方法 `getPageLinkWrapper` 参数表变成要加上 `rel` 参数：

```
abstract public function getPageLinkWrapper($url, $page, $rel = null);
```

Iron.Io Queue 加密

如果你使用 Iron.io queue 驱动，你将需要增加一个新的 `encrypt` 选项到你的 queue 设置文件中：

```
'encrypt' => true
```

从 4.1.x 升级到 4.1.29

Laravel 4.1.29 对于所有的数据库驱动加强了 column quoting 的部分。当你的模型中没有使用 `fillable` 属性，他保护你的应用程序不会受到 mass assignment 漏洞影响。如果你在模型中使用 `fillable` 属性来防范 mass assignment，你的应用程序将不会有漏洞。如果你使用 `guarded` 且在「更新」或「保存」类型的函式中，传递了末端用户控制的数组，那你应该立即升级到 `4.1.29` 以避免 mass assignment 的风险。

升级到 Laravel 4.1.29，只要 `composer update` 即可。在这个发行版本中没有重大的更新。

从 4.1.25 升级到 4.1.26

Laravel 4.1.26 采用了针对「记得我」cookies 的安全性更新。在此更新之前，如果一个记得我的 cookies 被恶意用户劫持，该 cookie 将还可以生存很长一段时间，即使真实用户重设密码或者注销亦同。

此更动需要在你的 `users` (或者类似的) 的数据表中增加一个额外的 `remember_token` 字段。在更新之后，当用户每次登录你的应用程序将会有一个全新的 token 将会被指派。这个 token 也会在用户注销应用程序后被更新。这个更新的影响为：如果一个「记得我」的 cookie 被劫持，只要用户注销应用程序将会废除该 cookie。

升级路径

首先，增加一个新的字段，可空值、属性为 VARCHAR(100)、TEXT 或同类型的字段 `remember_token` 到你的 `users` 数据表中。

然后，如果你使用 Eloquent 认证驱动，依照下面更新你的 `User` 类别的三个方法：

```
public function getRememberToken()
{
    return $this->remember_token;
}

public function setRememberToken($value)
{
    $this->remember_token = $value;
}
```

```
public function getRememberTokenName()
{
    return 'remember_token';
}
```

> 附注: 所有现存的「记得我」sessions 在此更新后将会失效，所以应用程序的所有用户将会被迫重新登录。

组件管理者

两个新的方法被加入到 `Illuminate\Auth\UserProviderInterface` 接口。范例实现方式可以在默认驱动中找到：

```
public function retrieveByToken($identifier, $token);

public function updateRememberToken(UserInterface $user, $token);
```

`Illuminate\Auth\UserInterface` 也加了三个新方法描述在「升级路径」。

从 4.0 升级到 4.1

升级你的 Composer 依赖性

升级你的应用程序至 Laravel 4.1，将 `composer.json` 里的 `laravel/framework` 版本更改至 `4.1.*`。

文件置换

将你的 `public/index.php` 置换成 这个 repository 的干净版本。

同样的，将你的 `artisan` 置换成 这个 repository 的干净版本。

添加设置文件及选项

更新你在设置文件 `app/config/app.php` 里的 `aliases` 和 `providers` 数组。而更新的选项值可以在 这个文件 中找到。请确定将你后来加入自定和组件所需的 providers / aliases 加回数组中。

从 这个 repository 增加 `app/config/remote.php` 文件。

在你的 `app/config/session.php` 增加新的选项 `expire_on_close`。而默认值为 `false`。

在你的 `app/config/queue.php` 文件里添加 `failed` 设置区块。以下为区块的默认值：

```
'failed' => array(
    'database' => 'mysql', 'table' => 'failed_jobs',
),
```

（非必要）在你的 `app/config/view.php` 里，将 `pagination` 设置选项更新为 `pagination::slider-3`。

更新控制器（Controllers）

如果 `app/controllers/BaseController.php` 有 `use` 语句在最上面，将 `use Illuminate\Routing\Controllers\Controller;` 改为 `use Illuminate\Routing\Controller;`。

更新密码提醒

密码提醒功能已经大幅修正拥有更大的弹性。你可以执行 Artisan 指令 `php artisan auth:reminders-controller` 来检查新的存根控制器。你也可以浏览 更新文件 然后相应的更新你的应用程序。

更新你的 `app/lang/en/reminders.php` 语系文件来符合 这个新版文件。

更新环境侦测

为了安全因素，不再使用网域网址来侦测辨别应用程序的环境。因为这些直很容易被伪造欺骗，继而让攻击者透过请求来达到变更环境。所以你必须改为使用机器的 hostname（在 Mac & Ubuntu 下执行 `hostname` 出来的值）

（译按：的确原有方式有安全性考量，但对于现行 VirtualHost 大量使用下，反而这样改会造成不便）

更简单的日志文件

Laravel 目前只会产生单一的日志文件：`app/storage/logs/laravel.log`。然而，你还是可以透过设置你的 `app/start/global.php` 文件来更改他的行为。

删除重定向结尾的斜线

在你的 `bootstrap/start.php` 文件中，移除调用 `$app->redirectIfTrailingSlash()`。这个方法已不再需要了，因为之后将会改以框架内的 `.htaccess` 来处理。

然后，用 新版 替换掉你 Apache 中的 `.htaccess` 文件，来处理结尾的斜线问题。

取得目前路由

取得目前路由的方法由 `Route::getCurrentRoute()` 改为 `Route::current()`。

Composer 更新

一旦你完成以上的更新，你可以执行 `composer update` 来更新应用程序的核心文件。如果有 class load 错误，请在 `update` 之后加上 `--no-scripts`，如：`composer update --no-scripts`。

万用字符事件监听者

万用字符事件监听者不再添加事件为参数到你的处理函数。如果你需要寻找你触发的事件你应该用 `Event::firing()`.

贡献指南

缺陷报告

为了促进有效积极的合作，相对于仅提交 缺陷报告 来说， Laravel 团队更鼓励使用 GitHub的 Pull Request。 当然也可以用 Pull Request 的方式发送含有失败单元测试的「缺陷报告」。

当您在呈递缺陷报告的时候，请确保您所提交的问题含有标题和清晰的描述。同时应该附带尽可能详细的与问题相关的信息和代码示例。 缺陷报告的目标是尽可能的方便您与他人去重现错误并修复它。

请谨记，建立缺陷报告是希望您与其他遇到同样问题的人一起解决这个问题。 但请不要期望其他人会主动的过来修复它。 创建缺陷报告是为了给您和他人提供一个修复问题的切入点。

Laravel 框架的源代码托管在 Github， 以下列出了每个 Laravel 相关项目仓库的连接:

- Laravel Framework
- Laravel Application
- Laravel Documentation
- Laravel Cashier
- Laravel Envoy
- Laravel Homestead
- Laravel Homestead Build Scripts
- Laravel Website
- Laravel Art

核心开发讨论区

讨论区在 (Freenode) 上的 #laravel-dev IRC 频道， 讨论内容包括缺陷，新特性和计划实施的已有特性. Laravel 项目维护者 Taylor Otwell 通常会在周一至周五的美国芝加哥时间 8am-5am 上线 (UTC-06:00 or America/Chicago)， 当然其它时间他也会偶尔出现。

#laravel-dev IRC 频道是对所有人开放的，欢迎任何有兴趣的朋友参与进来讨论或哪怕只是围观!

如何选择分支?

所有的 缺陷修正都应该提交到最后一版的稳定分支。 永远 不要把缺陷修正提交到 `master` 分支除非这些正是在下个发行版本中他们要修复的特性。

那些 完全向后兼容 并随当前 Laravel 版发行的 非重要 特性也许可以提交到最后一版的稳定分支。

那些在下一个 Laravel 发行版中将要出现的 重要的 新特性应该总是被提交到 `master` 分支。

如果您也不确定你写的特性是否重要时，请到 (Freenode) 的 #laravel-dev IRC 频道 问一下 Taylor Otwell。

安全缺陷

如果你在 Laravel 中发现安全缺陷，烦请以电子邮件的方式发送给 Taylor Otwell taylorotwell@gmail.com。所有的安全缺陷都将会被及时的处理掉。

代码风格

Laravel 框架遵循 PSR-4 和 PSR-1 代码标准。除了这些以外，如下的代码标准也应该被遵守:

- 类命名空间的声明必须与 `<?php` 处在同一行。
- 类的起始花括号 `{` 必须与类名处在同一行。
- 函数和控制结构必须使用 Allman 样式 括起来。
- 缩进使用制表符，对齐使用空格。

Laravel 安装指南

安装 Composer

Laravel 框架使用 Composer 来管理其依赖性。所以，在你使用 Laravel 之前，你必须确认在你电脑上是否安装了 Composer。

安装 Laravel

通过 Laravel 安装工具

首先，使用 Composer 下载 Laravel 安装包：

```
composer global require "laravel/installer=~1.1"
```

请确定把 `~/.composer/vendor/bin` 路径放置于您的 `PATH` 里， 这样 `laravel` 执行文件就会存在你的系统。

一旦安装完成后，就可以使用 `laravel new` 命令建立一份全新安装的 `Laravel` 应用，例如： `laravel new blog` 将会在当前目录下建立一个名为 `blog` 的目录， 此目录里面存放着全新安装的 Laravel 相关代码，此方法跟其他方法不一样的地方在于会提前安装好所有相关代码，不需要再通过 `composer install` 安装相关依赖，速度会快许多。

```
laravel new blog
```

通过 Composer Create-Project

你一样可以通过 Composer 在命令行执行 `create-project` 来安装 Laravel：

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

脚手架

Laravel 自带了用户注册和认证的脚手架。如果你想要移除这个脚手架，使用 `fresh` 命令即可：

php artisan fresh

环境需求

Laravel 框架有一些系统上的需求：

- PHP 版本 >= 5.4
- Mcrypt PHP 扩展
- OpenSSL PHP 扩展
- Mbstring PHP 扩展
- Tokenizer PHP 扩展

在 PHP 5.5 之后， 有些操作系统需要手动安装 PHP JSON 扩展包。如果你是使用 Ubuntu，可以通过 `apt-get install php5-json` 来进行安装。

配置

在你安装完 Laravel 后，首先需要做的事情是配置一个随机字串作为应用程序密钥。假设你是通过 comoser 安装 Laravel ，这个密钥会自动通过 `key:generate` 命令帮你配置完成。

通常这个密钥应该有 32 字符长。这个密钥可以被配置在 `.env` 环境文件中。 如果这密钥没有被配置的话，你的用户 **sessions** 和其他的加密数据都是不安全的！

Laravel 几乎不需配置就可以马上使用。你可以自由的开始开发！然而，你可以查看 `config/app.php` 文件和其他的文档。你可能希望根据你的应用程序而做更改，文件包含数个选项如 `时区` 和 `语言环境`。

一旦 Laravel 安装完成，你应该同时 配置本地环境。

> 注意： 你不应该在正式环境中将 `app.debug` 配置为 `true`。绝对！千万不要！

权限

Laravel 框架有一个目录需要额外配置权限： `storage` 要让服务器有写入的权限。

优雅链接

Apache

Laravel 框架通过 `public/.htaccess` 文件来让网址中不需要 `index.php`。如果你的网页服务器是使用 Apache 的话，请确认是否有开启 `mod_rewrite` 模块。

假设 Laravel 附带的 `.htaccess` 文件在 Apache 无法生效的话，请尝试下面的方法：

```
Options +FollowSymLinks
RewriteEngine On

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

Nginx

在 Nginx，在你的网站配置中增加下面的配置，可以使用「优雅链接」：

```
location / {
```

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

当然，如果你使用 Homestead 的话，优雅链接会自动的帮你配置完成。

配置

简介

所有 Laravel 框架的配置文件都放置在 `config` 目录下。 每个选项都有说明，因此你可以轻松地浏览这些文档，并且熟悉这些选项配置。

完成安装后

命名你的应用程序

在安装 Laravel 后，你可以「命名」你的应用程序。默认情况下，`app` 的目录是在 `App` 的命名空间 下，通过 Composer 使用 PSR-4 自动载入规范 自动加载。不过，你可以轻松地通过 Artisan 命令 `app:name` 来修改命名空间，以配合你的应用程序名称。

举例来说，假设你的应用程序叫做「 Horsefly 」，你可以从安装的根目录执行下面的命令：

```
php artisan app:name Horsefly
```

重命名你的应用程序是完全可选的，你也可以保留原有的命名空间 `App` 。

其他配置

Laravel 几乎不需配置就可以马上使用。你可以自由的开始开发！然而，你可以浏览 `config/app.php` 文件和其他的文档。你可能希望依据你的本机而做更改，文件包含数个选项如 `时区` 和 `语言环境` 。

一旦 Laravel 安装完成，你应该同时 配置本机环境。

> 注意： 你不应该在正式环境中将 `app.debug` 配置为 `true` 。绝对！千万不要！

权限

Laravel 框架有一个目录需要额外权限： `storage` 目录必须让服务器有写入权限。

取得配置值

你可以很轻松的使用 `Config` facade 取得你的配置值：

```
$value = Config::get('app.timezone');

Config::set('app.timezone', 'America/Chicago');
```

你也可以使用 `config` 辅助方法：

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

环境配置

通常应用程序常常需要根据不同的执行环境而有不同的配置值。例如，你会希望在你的本机开发环境上会有与正式环境不同的缓存驱动（cache driver），通过配置文件，就可以轻松完成。

Laravel 通过 DotEnv Vance Lucas 写的一个 PHP 类库。 在全新安装好的 Laravel 里，你的应用程序的根目录下会包含一个 `.env.example` 文件。如果你通过 Composer 安装 Laravel，这个文件将自动被命名为 `.env`，不然你应该手动更改文件名。

当你的应用程序收到请求，这个文件所有的变量会被加载到 `$_ENV` 这个 PHP 超级全局变量里。你可以使用辅助方法 `env` 查看这些变量。事实上，如果你查看过 Laravel 配置文件，你会注意到几个选项已经在使用这个辅助方法！

根据你的本机服务器或者线上环境需求，你可以自由的修改你的环境变量。然而， 你的 `.env` 文件不应该被提交到应用程序的版本控制系统，因为每个开发人员或服务器使用你的应用程序可能需要不同的环境配置。

如果你是一个团队的开发者，不妨将 `.env.example` 文件包含到你的应用程序。通过例子配置文件里的预留值，你的团队中其他开发人员可以清楚地看到执行你的应用程序所需的哪些环境变量。

取得目前应用程序的环境

你可以通过 `Application` 实例中的 `environment` 方法取得目前应用程序的环境：

```
$environment = $app->environment();
```

你也可以传递参数至 `environment` 方法中，来确认目前的环境是否与参数相符合：

```
if ($app->environment('local'))
{
    // The environment is local
}

if ($app->environment('local', 'staging'))
{
    // The environment is either local OR staging...
}
```

如果想取得应用程序的实例，可以通过服务容器的 `Illuminate\Contracts\Foundation\Application` contract 来取得。当然，如果你想在服务提供者中使用，应用程序实例可以通过实例变量 `$this->app` 取

得。

也能通过 `App` facade 或者辅助方法 `app` 取得应用程序实例：

```
$environment = app()->environment();

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

配置缓存

为了让你的的应用程序提升一些速度，你可以使用 Artisan 命令 `config:cache` 将所有的配置文件缓存到单一文件。通过命令会将所有的配置选项合并成一个文件，让框架能够快速加载。

通常来说，你应该将执行 `config:cache` 命令作为部署工作的一部分。

维护模式

当你的应用程序处于维护模式时，所有的路由都会指向一个自定的视图。当你要更新或维护网站时，「关闭」整个网站是很简单的。维护模式会检查包含在应用程序的默认中间件堆栈。如果应用程序处于维护模式，`HttpException` 会抛出 503 的状态码。

启用维护模式，只需要执行 Artisan 命令 `down` ：

```
php artisan down
```

关闭维护模式，请使用 Artisan 命令 `up` ：

```
php artisan up
```

维护模式的响应模板

维护模式响应的默认模板放在 `resources/views/errors/503.blade.php` 。

维护模式与队列

当应用程序处于维护模式中，将不会处理任何队列工作。所有的队列工作将会在应用程序离开维护模式后继续被进行。

优雅链接

Apache

Laravel 框架通过 `public/.htaccess` 文件来让网址中不需要 `index.php` 。如果你的服务器是使用 Apache ，请确认是否有开启 `mod_rewrite` 模块。

假设 Laravel 附带的 `.htaccess` 文件在 Apache 无法生效的话，请尝试下面的方法：

```
Options +FollowSymLinks
RewriteEngine On

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

Nginx

若使用 Nginx ，可以在你的网站配置中增加下面的配置，以开启「优雅链接」：

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

当然，如果你使用 Homestead 的话，优雅链接会自动的帮你配置完成。

Laravel Homestead

介绍

Laravel 致力于让 PHP 开发体验更愉快，也包含你的本地开发环境。Vagrant 提供了一个简单、优雅的方式来管理与供应虚拟机。

Laravel Homestead 是一个官方预载的 Vagrant「封装包」，提供你一个美好的开发环境，你不需要在你的本机端安装 PHP、HHVM、网页服务器或任何服务器软件。不用担心搞乱你的系统！Vagrant 封装包可以搞定一切。如果有什么地方出现故障，你可以在几分钟内快速的销毁并重建虚拟机。

Homestead 可以在任何 Windows、Mac 或 Linux 上面运行，里面包含了 Nginx 网页服务器、PHP 5.6、MySQL、Postgres、Redis、Memcached 还有所有你要开发精彩的 Laravel 应用程序所需的软件。

> 附注： 如果您是 Windows 的用户，您可能需要启用硬件虚拟化（VT-x）。通常需要通过 BIOS 来启用它。

Homestead 目前是构建且测试于 Vagrant 1.6 版本。

内置软件

- Ubuntu 14.04
- PHP 5.6
- HHVM
- Nginx
- MySQL
- Postgres
- Node (With Bower, Grunt, and Gulp)
- Redis
- Memcached
- Beanstalkd
- Laravel Envoy
- Fabric + HipChat Extension
- Blackfire Profiler

安装与配置

安装 VirtualBox / VMWare 与 Vagrant

在启动你的 Homestead 环境之前，你必须先安装 VirtualBox 和 Vagrant. 两套软件在各平台都有提供易用

的可视化安装程序。

VMware

除了 VirtualBox 之外, Homestead 也支持 VMware. 如果使用 VMware 作为 provider, 你需要购买 VMware Fusion / Desktop 以及 VMware Vagrant plug-in. VMware 提供了更快、性能更好的共享文件夹。

增加 Vagrant 封装包

当 VirtualBox / VMware 和 Vagrant 安装完成后，你可以在终端机以下列命令将 'laravel/homestead' 封装包安装进你的 Vagrant 安装程序中。下载封装包会花你一点时间，时间长短将依据你的网络速度决定:

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

如果这个命令失败了, 你可能安装的是一个老版本的 Vagrant 需要指定一个完整的 URL：

```
vagrant box add laravel/homestead https://atlas.hashicorp.com/laravel/boxes/homestead
```

安装 Homestead

手动通过 Git 安装（本地端没有 PHP）

如果你不希望在你的本机上安装 PHP ，你可以简单地通过手动复制资源库的方式来安装 Homestead。将资源库复制至你的 "home" 目录中的 `Homestead` 文件夹，如此一来 Homestead 封装包将能提供主机服务给你所有的 Laravel（及 PHP）应用:

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

一旦你安装完 Homestead CLI 工具，即可执行 `bash init.sh` 命令来创建 `Homestead.yaml` 配置文件:

```
bash init.sh
```

此 `Homestead.yaml` 文件，将会被放置在你的 `~/.homestead` 目录中。

通过 Composer + PHP 工具

一旦封装包已经安装进你的 Vagrant 安装程序，你就可以准备通过 Composer `global` 命令来安装 Homestead CLI 工具：

```
composer global require "laravel/homestead=~2.0"
```

请务必确认 `homestead` 有被放置在目录 `~/.composer/vendor/bin` 之中，如此一来你才能在终端机中顺利执行 `homestead` 命令。

一旦你安装完 Homestead CLI 工具，即可执行 `init` 命令来创建 `Homestead.yaml` 配置文件:

```
homestead init
```

此 `Homestead.yaml` 将会被放置在你的 `~/.homestead` 文件夹中。如果你是使用 Mac 或 Linux，你可以直接在终端机执行 `homestead edit` 命令来编辑 `Homestead.yaml`：

```
homestead edit
```

配置你的 Provider

在 `Homestead.yaml` 文件中的 `provider` 键表明需要使用的 Vagrant prodiver：`virtualbox` 或者 `vmware_fusion`，你可以根据自己的喜好设定 provider。

```
provider: virtualbox
```

配置你的 SSH 密钥

然后你需要编辑 `Homestead.yaml`。可以在文件中配置你的 SSH 公开密钥，以及主要机器与 Homestead 虚拟机之间的共享目录。

如果没有 SSH 密钥的话， 在 Mac 和 Linux 下，你可以利用下面的命令来创建一个 SSH 密钥组:

```
ssh-keygen -t rsa -C "you@homestead"
```

在 Windows 下，你需要安装 Git 并且使用包含在 Git 里的 `Git Bash` 来执行上述的命令。另外你也可以使用 PuTTY 和 PuTTYgen。

一旦你创建了一个 SSH 密钥，记得在你的 `Homestead.yaml` 文件中的 `authorize` 属性指明密钥路径。

配置你的共享文件夹

`Homestead.yaml` 文件中的 `folders` 属性列出了所有你想在 Homestead 环境共享的文件夹列表。这些文件夹中的文件若有变动，他们将会同步在你的本机与 Homestead 环境里。你可以将你需要的共享文件夹都配置进去。

如果要开启 NFS，只需要在 `folders` 中加入一个标识：

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

配置你的 Nginx 站点

对 Nginx 不熟悉？没关系。 `sites` 属性允许你简单的对应一个 `域名` 到一个 homestead 环境中的目录。一个例子的站点被配置在 `Homestead.yaml` 文件中。同样的，你可以加任何你需要的站点到你的 Homestead 环境中。Homestead 可以为你每个进行中的 Laravel 应用提供方便的虚拟化环境。

你可以通过配置 `hhvm` 属性为 `true` 来让虚拟站点支持 HHVM:

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

Bash Aliases

如果要增加 Bash aliases 到你的 Homestead 封装包中，只要将内容添加到 `~/.homestead` 目录最上层的 `aliases` 文件中即可。

启动 Vagrant 封装包

当你根据你的喜好编辑完 `Homestead.yaml` 后，在终端机里进入你的 Homestead 文件夹并执行 `homestead up` 命令。

Vagrant 会将虚拟机开机，并且自动配置你的共享目录和 Nginx 站点。如果要移除虚拟机，可以使用 `vagrant destroy --force` 命令。

为了你的 Nginx 站点，别忘记在你的机器的 `hosts` 文件将「域名」加进去。`hosts` 文件会将你的本地域名的站点请求指向你的 Homestead 环境中。在 Mac 和 Linux，该文件放在 `/etc/hosts`。在 Windows 环境中，它被放置在 `C:\Windows\System32\drivers\etc\hosts`。你要加进去的内容类似如下：

```
192.168.10.10  homestead.app
```

务必确认 IP 地址与你的 `Homestead.yaml` 文件中的相同。一旦你将域名加进你的 `hosts` 文件中，你就可以通过网页浏览器访问到你的站点。

```
http://homestead.app
```

继续读下去，你会学到如何连接到数据库！

常见用法

通过 SSH 连接

要通过 SSH 连接上您的 Homestead 环境，在终端机里进入你的 Homestead 目录并执行 `vagrant ssh` 命令。

因为你可能会经常需要通过 SSH 进入你的 Homestead 虚拟机，可以考虑在你的主要机器上创建一个"别名":

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

一旦你创建了这个别名，无论你在主要机器的哪个目录，都可以简单地使用 "vm" 命令来通过 SSH 进入你

的 Homestead 虚拟机。

连接数据库

在 `Homestead` 封装包中，已经预了 MySQL 与 Postgres 两种数据库。为了更简便，Laravel 的 `local` 数据库配置已经默认将其配置完成。

如果想要从本机上通过 Navicat 或者 Sequel Pro 连接 MySQL 或者 Postgres 数据库，你可以连接 `127.0.0.1` 的端口 33060 (MySQL) 或 54320 (Postgres)。而帐号密码分别是 `homestead` / `secret`。

> 附注： 从本机端你应该只能使用这些非标准的连接端口来连接数据库。因为当 Laravel 运行在虚拟机时，在 Laravel 的数据库配置文件中依然是配置使用默认的 3306 及 5432 连接端口。

增加更多的站点

在 Homestead 环境上架且运行后，你可能会需要为 Laravel 应用程序增加更多的 Nginx 站点。你可以在单一个 Homestead 环境中运行非常多 Laravel 安装程序。有两种方式可以达成：第一种，在 `Homestead.yaml` 文件中增加站点然后执行 `homestead provision` 或者 `vagrant provision`。

另外，也可以使用存放在 Homestead 环境中的 `serve` 命令文件。要使用 `serve` 命令文件，请先 SSH 进入 Homestead 环境中，并执行下列命令：

```
serve domain.app /home/vagrant/Code/path/to/public/directory
```

> 附注： 在执行 `serve` 命令过后，别忘记将新的站点加进本机的 `hosts` 文件中。

连接端口

以下的端口将会被转发至 Homestead 环境：

- **SSH:** 2222 → Forwards To 22
- **HTTP:** 8000 → Forwards To 80
- **MySQL:** 33060 → Forwards To 3306
- **Postgres:** 54320 → Forwards To 5432

增加额外端口

你也可以自定义转发额外的端口至 Vagrant box，只需要指定协议：

```
ports:
    - send: 93000
      to: 9300
    - send: 7777
      to: 777
      protocol: udp
```

Blackfire Profiler

Blackfire Profiler 是由 SensioLabs 创建的一个分析工具，它会自动的收集代码执行期间的相关数据，比如 RAM, CPU time, 和 disk I/O. 如果你使用 Homestead ，那么使用这个分析工具会变得非常简单。

blackfire 所需的包已经安装在 Homestead box 中，你只需要在 `Homestead.yaml` 文件中设置 Server ID 和 token ：

```
blackfire:
    - id: your-id
      token: your-token
```

当你设定完 Blackfire 的凭证信息，使用 `homestead provision` 或者 `vagrant provision` 令配置生效。当然，你也需要通过阅读Blackfire 文档 来学习如何在你的浏览器中安装 Blackfire 扩展。

HTTP 路由

基本路由

您将在 `app/Http/routes.php` 中定义应用中的大多数路由，这个文件加载了 `App\Providers\RouteServiceProvider` 类。 大多数基本的 Laravel 路由都只接受一个 URI 和 一个 `闭包(Closure)` 参数：

基本 GET 路由

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

其他基础路由

```
Route::post('foo/bar', function()
{
    return 'Hello World';
});

Route::put('foo/bar', function()
{
    //
});

Route::delete('foo/bar', function()
{
    //
});
```

为多种请求注册路由

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

注册路由响应所有 HTTP 请求

```
Route::any('foo', function()
{
    return 'Hello World';
});
```

通常情况下，您将会需要为您的路由产生 URL，您可以使用 `url` 辅助函数来操作：

```
$url = url('foo');
```

CSRF 保护

Laravel 提供简易的方法，让您可以保护您的应用程序不受到 CSRF (跨网站请求伪造) 攻击。跨网站请求伪造是一种恶意的攻击，借以代表经过身份验证的用户执行未经授权的命令。

Laravel 会自动在每一位用户的 session 中放置随机的 `token` ，这个 token 将被用来确保经过验证的用户是实际发出请求至应用程序的用户：

插入 CSRF Token 到表单

```
<input type="hidden" name="_token" value="<?php echo csrf_token(); ?>">
```

当然也可以在 Blade 模板引擎使用：

```
<input type="hidden" name="_token" value="{{ csrf_token() }}">
```

您不需要手动验证在 POST、PUT、DELETE 请求的 CSRF token。 `VerifyCsrfToken` HTTP 中间件将保存在 session 中的请求输入的 token 配对来验证 token 。

X-CSRF-TOKEN

除了寻找 CSRF token 作为「POST」参数，中间件也检查 `X-XSRF-TOKEN` 请求头，比如，你可以把 token 存放在 meta 标签中, 然后使用 jQuery 将它加入到所有的请求头中：

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

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

现在所有的 AJAX 请求会自动加入 CSRF token：

```
$.ajax({
  url: "/foo/bar",
})
```

X-XSRF-TOKEN

Laravel 也在 cookie 中存放了名为 `XSRF-TOKEN` 的 CSRF token。你可以使用这个 cookie 值来设置 `X-XSRF-TOKEN` 请求头。一些 Javascript 框架，比如 Angular ，会自动设置这个值。

> 注意： `X-CSRF-TOKEN` 和 `X-XSRF-TOKEN` 的不同点在于前者使用的是纯文本而后者是一个加密的值，因为在 Laravel 中 cookies 始终是被加密过的。如果你使用 `csrf_token()` 函数来作为 token 的值， 你需要设置 `X-CSRF-TOKEN` 请求头。

方法欺骗

HTML 表单没有支持 `PUT` 或 `DELETE` 请求。所以当定义 `PUT` 或 `DELETE` 路由并在 HTML 表单中被调用的时候，您将需要添加隐藏 `_method` 字段在表单中。

发送的 `_method` 字段对应的值会被当做 HTTP 请求方法。举例来说：

```
<form action="/foo/bar" method="POST">
    <input type="hidden" name="_method" value="PUT">
    <input type="hidden" name="_token" value="<?php echo csrf_token(); ?>">
</form>
```

路由参数

当然，您可以获取请求路由的 URI 区段。

基础路由参数

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

可选择的路由参数

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

带默认值的路由参数

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

使用正则表达式限制参数

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

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

使用条件限制数组

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

定义全局模式

如果你想让特定路由参数总是遵询特定的正则表达式，可以使用 `pattern` 方法。在 `RouteServiceProvider` 的 `boot` 方法里定义模式：

```
$router->pattern('id', '[0-9]+');
```

定义模式之后，会作用在所有使用这个特定参数的路由上：

```
Route::get('user/{id}', function($id)
{
    // 只有 {id} 是数字才被调用。
});
```

取得路由参数

如果需要在路由外部取得其参数，使用 `input` 方法：

```
if ($route->input('id') == 1)
{
    //
}
```

你也可以使用 `Illuminate\Http\Request` 实体取得路由参数。当前请求的实例可以通过 `Request` facade 取得，或透过类型提示 `Illuminate\Http\Request` 注入依赖：

```
use Illuminate\Http\Request;

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

```
    if ($request->route('id'))
    {
        //
    }
});
```

命名路由

命名路由让你更方便于产生 URL 与重定向特定路由。您可以用 `as` 的数组键值指定名称给路由：

```
Route::get('user/profile', ['as' => 'profile', function()
{
    //
}]);
```

也可以为控制器动作指定路由名称：

```
Route::get('user/profile', [
    'as' => 'profile', 'uses' => 'UserController@showProfile'
]);
```

现在你可以使用路由名称产生 URL 或进行重定向：

```
$url = route('profile');

$redirect = redirect()->route('profile');
```

`currentRouteName` 方法会返回目前请求的路由名称：

```
$name = Route::currentRouteName();
```

路由群组

有时候您需要嵌套过滤器到群组的路由上。不需要为每个路由去嵌套过滤器，您只需使用路由群组：

```
Route::group(['middleware' => 'auth'], function()
{
    Route::get('/', function()
    {
        // Has Auth Filter
    });

    Route::get('user/profile', function()
    {
        // Has Auth Filter
    });
});
```

您一样可以在 `group` 数组中使用 `namespace` 参数，指定在这群组中控制器的命名空间：

```
Route::group(['namespace' => 'Admin'], function()
{
    //
});
```

> 注意： 在默认情况下， `RouteServiceProvider` 包含内置您命名空间群组的 `routes.php` 文件，让您不须使用完整的命名空间就可以注册控制器路由。

子域名路由

Laravel 路由一样可以处理通配符的子域名，并且从域名中传递您的通配符参数：

注册子域名路由

```
Route::group(['domain' => '{account}.myapp.com'], function()
{

    Route::get('user/{id}', function($account, $id)
    {
        //
    });

});
```

路由前缀

群组路由可以通过群组的描述数组中使用 `prefix` 选项，将群组内的路由加上前缀：

```
Route::group(['prefix' => 'admin'], function()
{

    Route::get('user', function()
    {
        //
    });

});
```

路由模型绑定

Laravel 模型绑定提供方便的方式将模型实体注入到您的路由中。例如，比起注入 User ID ，你可以选择注入符合给定 ID 的 User 类实体。

首先，使用路由的 `model` 方法指定特定参数要对应的类，您应该在 `RouteServiceProvider::boot` 方法定义您的模型绑定：

绑定参数至模型

```
public function boot(Router $router)
{
    parent::boot($router);

    $router->model('user', 'App\User');
}
```

然后定义一个有 `{user}` 参数的路由：

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

因为我们已经将 `{user}` 参数绑定到 `App\User` 模型，所以 `User` 实体将被注入到路由。所以举例来说，请求至 `profile/1` 将注入 ID 为 1 的 `User` 实体。

> 注意： 如果在数据库中找不到匹配的模型实体，将引发 404 错误。

如果您想要自定「没有找到」的行为，将闭包作为第三个参数传入 `model` 方法：

```
Route::model('user', 'User', function()
{
    throw new NotFoundHttpException;
});
```

如果您想要使用您自己决定的逻辑，您应该使用 `Router::bind` 方法。闭包通过 `bind` 方法将传递 URI 区段数值，并应该返回您想要被注入路由的类实体：

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

抛出 404 错误

这里有两种方法从路由手动触发 404 错误。首先，您可以使用 `abort` 辅助函数：

```
abort(404);
```

`abort` 辅助函数只是简单抛出带有特定状态代码的 `Symfony\Component\HttpFoundation\Exception\HttpException` 。

第二，您可以手动抛出 `Symfony\Component\HttpKernel\Exception\NotFoundHttpException` 的实体。

有关如何处理 404 异常状况和自定响应的更多信息，可以参考错误章节内的文档。

HTTP 中间件

简介

HTTP 中间件提供一个方便的机制来过滤进入应用程序的 HTTP 请求，例如，Laravel 默认包含了一个中间件来检验用户身份验证，如果用户没有经过身份验证，中间件会将用户导向登录页面，然而，如果用户通过身份验证，中间件将会允许这个请求进一步继续前进。

当然，除了身份验证之外，中间件也可以被用来执行各式各样的任务，CORS 中间件负责替所有即将离开程序的响应加入适当的响应头，一个日志中间件可以记录所有传入应用程序的请求。 Laravel 框架已经内置一些中间件，包括维护、身份验证、CSRF 保护，等等。所有的中间件都位于 `app/Http/Middleware` 目录内。

建立中间件

要建立一个新的中间件，可以使用 `make:middleware` 这个 Artisan 命令：

```
php artisan make:middleware OldMiddleware
```

此命令将会 在 `app/Http/Middleware` 目录内置立一个名称为 `OldMiddleware` 的类。在这个中间件内我们只允许 `年龄` 大于 200 的才能访问路由，否则，我们会将用户重新导向 「home」 的 URI 。

```
<?php namespace App\Http\Middleware;

class OldMiddleware {

    /**
     * Run the request filter.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if ($request->input('age') < 200)
        {
            return redirect('home');
        }

        return $next($request);
    }

}
```

如你所见，若是 `年龄` 小于 `200` ，中间件将会返回 HTTP 重定向给客户端，否则，请求将会进一步传递到应用程序。只需调用带有 `$request` 的 `$next` 方法，即可将请求传递到更深层的应用程序(允许跳过中间件) HTTP 请求在实际碰触到应用程序之前，最好是可以层层通过许多中间件，每一层都可以对请求进行检查，甚至是完全拒绝请求。

Before* / *After 中间件

在一个请求前后指定某个中间件取决于这个中间件自身。这个中间件可以执行在请求前执行一些 前置 操作：

```
<?php namespace App\Http\Middleware;

class BeforeMiddleware implements Middleware {

    public function handle($request, Closure $next)
    {
        // Perform action

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

然后，这个中间件也可以在请求后执行一些 后置 操作：

```
<?php namespace App\Http\Middleware;

class AfterMiddleware implements Middleware {

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

        // Perform action

        return $response;
    }
}
```

注册中间件

全局中间件

若是希望中间件被所有的 HTTP 请求给执行，只要将中间件的类加入到 `app/Http/Kernel.php` 的 `$middleware` 属性清单列表中。

指派中间件给路由

如果你要指派中间件给特定的路由，你得先将中间件在 `app/Http/Kernel.php` 配置一个键值，默认情况下，这个文件内的 `$routeMiddleware` 属性已包含了 Laravel 目前配置的中间件，你只需要在清单列表中加上一组自定义的键值即可。 中间件一旦在 HTTP kernel 文件内被定义，你即可在路由选项内使用 `middleware` 键值来指派：

```
Route::get('admin/profile', ['middleware' => 'auth', function()
{
    //
}]);
```

可终止中间件

有些时候中间件需要在 HTTP 响应已被发送到用户端之后才执行，例如，Laravel 内置的「session」中间件，保存 session 数据是在响应已被发送到用户端 之后 才执行。为了做到这一点，你需要定义中间件为「可终止的」。

```
use Illuminate\Contracts\Routing\TerminableMiddleware;

class StartSession implements TerminableMiddleware {

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

    public function terminate($request, $response)
    {
        // Store the session data...
    }

}
```

如你所见，除了定义 `handle` 方法之外， `TerminableMiddleware` 定义一个 `terminate` 方法。这个方法接收请求和响应。一旦定义了 terminable 中间件，你需要将它增加到 HTTP kernel 文件的全局中间件清单列表中。

HTTP 控制器

简介

除了在单一的 `routes.php` 文件中定义所有的请求处理逻辑之外，你可能希望使用控制器类来组织此行为。控制器可将相关的 HTTP 请求处理逻辑组成一个类。控制器通常存放在 `app/Http/Controllers` 此目录中。

基础控制器

这里是一个基础控制器类的例子：

```
<?php namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

class UserController extends Controller {

    /**
     * 显示所给定的用户个人数据。
     *
     * @param  int  $id
     * @return Response
     */
    public function showProfile($id)
    {
        return view('user.profile', ['user' => User::findOrFail($id)]);
    }

}
```

我们可以通过如下方式引导路由至对应的控制器动作：

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

> 注意： 所有的控制器都应该扩展基础控制器类。

控制器和命名空间

有一点非常重要，那就是我们无需指明完整的控制器命名空间，在类名称中 `App\Http\Controllers` 之后的部分即可用于表示「根」命名空间。 `RouteServiceProvider` 默认会在包含根控制器命名空间的路由群

组中，加载 `routes.php` 此文件。

若你要在 `App\Http\Controllers` 此目录深层使用 PHP 命名空间以嵌套化或组织你的控制器，只要使用相对于 `App\Http\Controllers` 根命名空间的特定类名称即可。因此，若你的控制器类全名为 `App\Http\Controllers\Photos\AdminController` ，你可以像这样注册一个路由：

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

命名控制器路由

和闭包路由一样，你也可以指定控制器路由的名称。

```
Route::get('foo', ['uses' => 'FooController@method', 'as' => 'name']);
```

指向控制器行为的 URL

要产生一个指向控制器行为的 URL，可使用 `action` 辅助方法。

```
$url = action('App\Http\Controllers\FooController@method');
```

若你想仅使用相对于控制器命名空间的类名称中的一部分，来产生指向控制器行为的 URL，可用 URL 产生器注册控制器的根命名空间。

```
URL::setRootControllerNamespace('App\Http\Controllers');

$url = action('FooController@method');
```

你可以使用 `currentRouteAction` 方法来获取正在执行的控制器行为名称：

```
$action = Route::currentRouteAction();
```

控制器中间件

中间件 可在控制器路由中指定，例如：

```
Route::get('profile', [
    'middleware' => 'auth',
    'uses' => 'UserController@showProfile'
]);
```

此外，你也可以在控制器构造器中指定中间件 ：

```
class UserController extends Controller {

    /**
     * 建立一个新的 UserController 实例。
```

```
     */
    public function __construct()
    {
        $this->middleware('auth');

        $this->middleware('log', ['only' => ['fooAction', 'barAction']]);

        $this->middleware('subscribed', ['except' => ['fooAction', 'barAction']]);
    }

}
```

隐式控制器

Laravel 让你能轻易地定义单一路由来处理控制器中的每一项行为。首先，用 `Route::controller` 方法定义一个路由：

```
Route::controller('users', 'UserController');
```

`Controller` 方法接受两个参数。第一个参数是控制器欲处理的 base URI，第二个是控制器的类名称。接着只要在你的控制器中加入方法，并在名称前加上它们所对应的 HTTP 请求。

```
class UserController extends BaseController {

    public function getIndex()
    {
        //
    }

    public function postProfile()
    {
        //
    }

    public function anyLogin()
    {
        //
    }

}
```

`index` 方法会响应控制器处理的根 URI ，在这个例子中是 `users` 。

如果你的控制器行为包含多个字词，你可以在 URI 中使用「破折号」语法来访问此行为。例如，下面这个在 `UserController` 中的控制器动作会响应 `users/admin-profile` 此一 URI ：

```
public function getAdminProfile() {}
```

设定路由名字

如果你想“命名”一些控制器的路由，你可以给 `controller` 方法传入第三个参数：

```
Route::controller('users', 'UserController', [
    'anyLogin' => 'user.login',
]);
```

RESTful 资源控制器

资源控制器可让你无痛建立和资源相关的 RESTful 控制器。例如，你可能希望创建一个控制器，它可用来处理针对你的应用程序所保存相片的 HTTP 请求。我们可以使用 `make:controller` Artisan 命令，快速创建这样的控制器：

```
php artisan make:controller PhotoController
```

接着，我们注册一个指向此控制器的资源路由：

```
Route::resource('photo', 'PhotoController');
```

此单一路由声明创建了多个路由，用来处理各式各样和相片资源相关的 RESTful 行为。同样地，产生的控制器已有各种和这些行为绑定的方法，包含用来通知你它们处理了那些 URI 及动词。

由资源控制器处理的行为

动词	路径	行为	路由名称
GET	/photo	索引	photo.index
GET	/photo/create	创建	photo.create
POST	/photo	保存	photo.store
GET	/photo/{photo}	显示	photo.show
GET	/photo/{photo}/edit	编辑	photo.edit
PUT/PATCH	/photo/{photo}	更新	photo.update
DELETE	/photo/{photo}	删除	photo.destroy

自定义资源路由

除此之外，你也可以指定让路由仅处理一部分的行为：

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

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

所有的资源控制器行为默认都有个路由名称。然而你可在选项中传递一个 `names` 数组来重载这些名称：

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

处理嵌套资源控制器

在你的路由声明中使用「点」号来「嵌套化」资源控制器：

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

此路由会注册一个「嵌套的」资源，可透过像 `photos/{photos}/comments/{comments}` 这样的 URL 来访问。

```
class PhotoCommentController extends Controller {

    /**
     * 显示指定照片的评论。
     *
     * @param  int  $photoId
     * @param  int  $commentId
     * @return Response
     */
    public function show($photoId, $commentId)
    {
        //
    }

}
```

在资源控制器中加入其他的路由

除了默认的资源路由外，若你还需要在资源控制器中加入其他路由，应该在调用 `Route::resource` 之前先定义它们：

```
Route::get('photos/popular', 'PhotoController@method');

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

依赖注入和控制器

构造器注入

Laravel 服务容器 用于解析所有的 Laravel 控制器。因此，你可以在控制器所需要的构造器中，对依赖作任何的类型限制。

```
<?php namespace App\Http\Controllers;

use Illuminate\Routing\Controller;
use App\Repositories\UserRepository;

class UserController extends Controller {

    /**
     * 用户保存库实例。
     */
    protected $users;
```

```
    /**
     * 创建新的控制器实例。
     *
     * @param  UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }

}
```

当然了，你也可以对任何的 Laravel contract 作类型限制。只要容器能解析它，你就可以对它作类型限制。

方法注入

除了建构器注入外，你也可以对控制器方法的依赖作类型限制。例如，让我们对某个方法的 `Request` 实例作类型限制：

```
<?php namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

class UserController extends Controller {

    /**
     * 保存一个新的用户。
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        $name = $request->input('name');

        //
    }

}
```

如果你的控制器方法预期由路由参数取得输入，只要在其他的依赖之后列出路由参数即可：

```
<?php namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

class UserController extends Controller {

    /**
     * 保存一个新的用户。
     *
     * @param  Request  $request
     * @param  int  $id
```

```
     * @return Response
     */
    public function update(Request $request, $id)
    {
        //
    }

}
```

> 注意： 方法注入和 模型绑定 是完全兼容的。容器可智能地判断那些参数和模型相关以及那些参数应该被注入。

路由缓存

若您的应用只使用了控制器路由，你可利用 Laravel 的路由缓存。使用路由缓存，将大幅降低注册应用程序所有路由所需要的时间。某些情况下，路由注册甚至可以快上 100 倍。要产生路由缓存，只要执行 `route:cache` Artisan 命令：

```
php artisan route:cache
```

就是这样！你的缓存路由文件将会被用来代替 `app/Http/routes.php` 此一文件。记住，若你增加了任何新的路由，你就 必须产生一个新的路由缓存。因此在应用部署时，你可能会希望只要执行 `route:cache` 命令：

要移除路由缓存文件，但不希望产生新的缓存，可使用 `route:clear` 命令：

```
php artisan route:clear
```

HTTP 请求

取得请求实例

通过 Facade

`Request` facade 允许你访问当前绑定容器的请求。例如：

```
$name = Request::input('name');
```

切记，如果你在一个命名空间中，你必须导入 `Request` facade，接着在类的上方声明 `use Request;`。

通过依赖注入

要通过依赖注入的方式取得 HTTP 请求的实例，你必须在控制器中的构造函数或方法对该类使用类型提示。当前请求的实例将会自动由服务容器注入：

```
<?php namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

class UserController extends Controller {

    /**
     * Store a new user.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        $name = $request->input('name');

        //
    }

}
```

如果你的控制器也有从路由参数传入的输入数据，只需要将路由参数置于其他依赖之后：

```
<?php namespace App\Http\Controllers;
```

```
use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

class UserController extends Controller {

    /**
     * Store a new user.
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function update(Request $request, $id)
    {
        //
    }

}
```

取得输入数据

取得特定输入数据

你可以通过 `Illuminate\Http\Request` 的实例，经由几个简洁的方法取得所有的用户输入数据。不需要担心发出请求时使用的 HTTP 请求，取得输入数据的方式都是相同的。

```
$name = Request::input('name');
```

取得特定输入数据，若没有则取得默认值

```
$name = Request::input('name', 'Sally');
```

确认是否有输入数据

```
if (Request::has('name'))
{
    //
}
```

取得所有发出请求时传入的输入数据

```
$input = Request::all();
```

取得部分发出请求时传入的输入数据

```
$input = Request::only('username', 'password');

$input = Request::except('credit_card');
```

如果是「数组」形式的输入数据，可以使用「点」语法取得数组：

```
$input = Request::input('products.0.name');
```

旧输入数据

Laravel 可以让你保留这次的输入数据，直到下一次请求发送前。例如，你可能需要在表单验证失败后重新填入表单值。

将输入数据存成一次性 Session

`flash` 方法会将当前的输入数据存进 session中，所以下次用户发出请求时可以使用保存的数据：

```
Request::flash();
```

将部分输入数据存成一次性 Session

```
Request::flashOnly('username', 'email');

Request::flashExcept('password');
```

快闪及重定向

你很可能常常需要在重定向至前一页，并将输入数据存成一次性 Session。只要在重定向方法后的链式调用方法中传入输入数据，就能简单地完成。

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

return redirect('form')->withInput(Request::except('password'));
```

取得旧输入数据

若想要取得前一次请求所保存的一次性 Session，你可以使用 `Request` 实例中的 `old` 方法。

```
$username = Request::old('username');
```

如果你想在 Blade 模板显示旧输入数据，可以使用更加方便的辅助方法 `old` ：

```
{{ old('username') }}
```

Cookies

Laravel 所建立的 cookie 会加密并且加上认证记号，这代表着被用户擅自更改的 cookie 会失效。

取得 Cookie 值

```
$value = Request::cookie('name');
```

加上新的 **Cookie** 到响应

辅助方法 `cookie` 提供一个简易的工厂方法来产生新的 `Symfony\Component\HttpFoundation\Cookie` 实例。可以在 `Response` 实例之后连接 `withCookie` 方法带入 cookie 至响应：

```
$response = new Illuminate\Http\Response('Hello World');

$response->withCookie(cookie('name', 'value', $minutes));
```

建立永久有效的 **Cookie***

虽然说是「永远」，但真正的意思是五年。

```
$response->withCookie(cookie()->forever('name', 'value'));
```

上传文件

取得上传文件

```
$file = Request::file('photo');
```

确认文件是否有上传

```
if (Request::hasFile('photo'))
{
    //
}
```

`file` 方法返回的对象是 `Symfony\Component\HttpFoundation\File\UploadedFile` 的实例，`UploadedFile` 继承了 PHP 的 `SplFileInfo` 类并且提供了很多和文件交互的方法。

确认上传的文件是否有效

```
if (Request::file('photo')->isValid())
{
    //
}
```

移动上传的文件

```
Request::file('photo')->move($destinationPath);

Request::file('photo')->move($destinationPath, $fileName);
```

其他上传文件的方法

`UploadedFile` 的实例还有许多可用的方法，可以至该对象的 API 文档了解有关这些方法的详细信息。

其他的请求信息

`Request` 类提供很多方法检查 HTTP 请求，它继承了 `Symfony\Component\HttpFoundation\Request` 类，下面是一些使用方式。

取得请求 URI

```
$uri = Request::path();
```

取得请求方法

```
$method = Request::method();

if (Request::isMethod('post'))
{
    //
}
```

确认请求路径是否符合特定格式

```
if (Request::is('admin/*'))
{
    //
}
```

取得请求 URL

```
$url = Request::url();
```

HTTP 响应

基本响应

从路由返回字串

最基本的响应就是从 Laravel 的路由返回字串：

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

建立自定义响应

但是以大部分的路由及控制器所执行的动作来说，你需要返回完整的 `Illuminate\Http\Response` 实例或是一个视图。返回一个完整的 `Response` 实例时，你能够自定义响应的 HTTP 状态码以及响应头。`Response` 实例继承了 `Symfony\Component\HttpFoundation\Response` 类，它提供了很多方法来建立 HTTP 响应。

```
use Illuminate\Http\Response;

return (new Response($content, $status))
              ->header('Content-Type', $value);
```

为了方便起见，你可以使用辅助方法 `response`：

```
return response($content, $status)
              ->header('Content-Type', $value);
```

> 提示： 有关 `Response` 方法的完整列表可以参照 API 文档 以及 Symfony API 文档.

在响应送出视图

如果想要使用 `Response` 类的方法，但最终返回视图给用户，你可以使用简便的 `view` 方法：

```
return response()->view('hello')->header('Content-Type', $type);
```

附加 **Cookies** 到响应

```
return response($content)->withCookie(cookie('name', 'value'));
```

链式方法

切记，大多数的 `Response` 方法都是可以链式调用的，用以建立流畅的响应：

```
return response()->view('hello')->header('Content-Type', $type)
                 ->withCookie(cookie('name', 'value'));
```

重定向

重定向响应通常是类 `Illuminate\Http\RedirectResponse` 的实例，并且包含用户要重定向至另一个 URL 所需的响应头。

返回重定向

有几种方法可以产生 `RedirectResponse` 的实例，最简单的方式就是透过辅助方法 `redirect`。当在测试时，建立一个模拟重定向响应的测试并不常见，所以使用辅助方法通常是可行的：

```
return redirect('user/login');
```

返回重定向并且加上快闪数据（ **Flash Data** ）

通常重定向至新的 URL 时会一并将数据存进一次性 Session。所以为了方便，你可以利用方法连接的方式创建一个 `RedirectResponse` 的实例并将数据存进一次性 Session：

```
return redirect('user/login')->with('message', 'Login Failed');
```

返回根据前一个 **URL** 的重定向

你可能希望将用户重定向至前一个位置，例如当表单提交之后。你可以使用 `back` 方法来达成这个目的：

```
return redirect()->back();

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

返回根据路由名称的重定向

当你调用辅助方法 `redirect` 且不带任何参数时，将会返回 `Illuminate\Routing\Redirector` 的实例，你可以对该实例调用任何的方法。举个例子，要产生一个 `RedirectResponse` 到一个路由名称，你可以使用 `route` 方法：

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

返回根据路由名称的重定向，并给予路由参数赋值

如果你的路由有参数，你可以放进 `route` 方法的第二个参数。

```
// 路由的 URI 为：profile/{id}

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

如果你要重定向至路由且路由的参数为 Eloquent 模型的「ID」，你可以直接将模型传入，ID 将会自动被提取：

```
return redirect()->route('profile', [$user]);
```

返回根据路由名称的重定向，并给予特定名称路由参数赋值

```
// 路由的 URI 为：profile/{user}

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

返回根据控制器动作的重定向

既然可以产生 `RedirectResponse` 的实例并重定向至路由名称，同样的也可以重定向至控制器动作：

```
return redirect()->action('App\Http\Controllers\HomeController@index');
```

> 提示：如果你已经通过 `URL::setRootControllerNamespace` 注册了根控制器的命名空间，那么就不需要对 `action()` 方法内的控制器指定完整的命名空间。

返回根据控制器动作的重定向，并给予参数赋值

```
return redirect()->action('App\Http\Controllers\UserController@profile', [1]);
```

返回根据控制器动作的重定向，并给予特定名称参数赋值

```
return redirect()->action('App\Http\Controllers\UserController@profile', ['user' => 1]);
```

其他响应

使用辅助方法 `response` 可以轻松的产生其他类型的响应实例。当你调用辅助方法 `response` 且不带任何参数时，将会返回 `Illuminate\Contracts\Routing\ResponseFactory` Contract 的实做。Contract 提供了一些有用的方法来产生响应。

建立 JSON 响应

`json` 方法会自动将响应头的 `Content-Type` 配置为 `application/json`：

```
return response()->json(['name' => 'Abigail', 'state' => 'CA']);
```

建立 JSONP 响应

```
return response()->json(['name' => 'Abigail', 'state' => 'CA'])
                 ->setCallback($request->input('callback'));
```

建立文件下载的响应

```
return response()->download($pathToFile);

return response()->download($pathToFile, $name, $headers);

return response()->download($pathToFile)->deleteFileAfterSend(true);
```

> 提醒：管理文件下载的扩展包，Symfony HttpFoundation，要求下载文件名必须为 ASCII。

响应宏

如果你想要自定义可以在很多路由和控制器重复使用的响应，你可以使用 `Illuminate\Contracts\Routing\ResponseFactory` 实做的方法 `macro`。

举个例子，来自服务提供者的 `boot` 方法:

```
<?php namespace App\Providers;

use Response;
use Illuminate\Support\ServiceProvider;

class ResponseMacroServiceProvider extends ServiceProvider {

    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Response::macro('caps', function($value) use ($response)
        {
            return $response->make(strtoupper($value));
        });
    }

}
```

`macro` 函数第一个参数为宏名称，第二个参数为闭包函数。闭包函数会在 `ResponseFactory` 的实做或者辅助方法 `response` 调用宏名称的时候被执行：

```
return response()->caps('foo');
```

视图 (View)

基本用法

视图里面包含了你应用程序所提供的 HTML 代码，并且提供一个简单的方式来分离控制器和网页呈现上的逻辑。视图被保存在 `resources/views` 文件夹内。

一个简单的视图看起来可能像这样：

```
<!-- 视图被保存在 resources/views/greeting.php -->

<html>
    <body>
        <h1>Hello, <?php echo $name; ?></h1>
    </body>
</html>
```

这个视图可以使用以下的代码传递到用户的浏览器：

```
Route::get('/', function()
{
    return view('greeting', ['name' => 'James']);
});
```

如你所见，`view` 辅助方法的第一个参数会对应到 `resources/views` 文件夹内视图文件的名称；传递到 `view` 辅助方法的第二个参数是一个能够在视图内取用的数据数组。

当然，视图文件也可以被存放在 `resources/views` 的子文件夹内。举例来说，如果你的视图文件保存在 `resources/views/admin/profile.php`，你可以用以下的代码来返回：

```
return view('admin.profile', $data);
```

传递数据到视图

```
// 使用传统的方法
$view = view('greeting')->with('name', 'Victoria');

// 使用魔术方法
$view = view('greeting')->withName('Victoria');
```

在上面的例子代码中，视图将可以使用 `$name` 来取得数据，其值为 `Victoria`。

如果你想的话，还有一种方式就是直接在 `view` 辅助方法的第二个参数直接传递一个数组：

```
$view = view('greetings', $data);
```

把数据共享给所有视图

有时候你可能需要共享一些数据给你的所有视图，你有很多个选择：`view` 辅助方法；`Illuminate\Contracts\View\Factory` 合约 (contract)；在 视图组件 (view composer) 内使用通配符。

这里有个 `view` 辅助方法的例子：

```
view()->share('data', [1, 2, 3]);
```

你也可以使用 `view` 的 Facade：

```
View::share('data', [1, 2, 3]);
```

通常你应该在服务提供者的 `boot` 方法内使用 `share` 方法。你可以选择加在 `AppServiceProvider` 或者是新建一个单独的服务提供者来容纳这些代码。

> 备注：当 `view` 辅助方法没有带入任何参数调用时，它将会返回一个的 `Illuminate\Contracts\View\Factory` 合约 (contract) 的实现 (implementation)。

确认视图是否存在

如果你需要确认视图是否存在，使用 `exists` 方法：

```
if (view()->exists('emails.customer'))
{
    //
}
```

从一个文件路径产生视图

你可以从一个完整的文件路径来产生一个视图：

```
return view()->file($pathToFile, $data);
```

视图组件

视图组件就是在视图被渲染前，会调用的闭包或类方法。如果你想在每次渲染某些视图时绑定数据，视图组件可以把这样的程序逻辑组织在同一个地方。

定义一个视图组件

让我们在 服务提供者 内组织我们的视图组件。底下例子将使用 `View` Facade 来取得底层 `Illuminate\Contracts\View\Factory` 合约的实现：

```php
<?php namespace App\Providers;

use View;
use Illuminate\Support\ServiceProvider;

class ComposerServiceProvider extends ServiceProvider {

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function boot()
    {
        // 使用类来指定视图组件
        View::composer('profile', 'App\Http\ViewComposers\ProfileComposer');

        // 使用闭包来指定视图组件
        View::composer('dashboard', function()
        {

        });
    }

    /**
     * Register
     *
     * @return void
     */
    public function register()
    {
        //
    }

}
```

> 备注： Laravel 没有默认的文件夹来放置类形式的视图组件。你可以自由的把它们放在你想要的地方。举例来说，你可以放在 `App\Http\ViewComposers` 文件夹内。

记得要把这个服务提供者添加到 `config/app.php` 配置文件的 `providers` 数组中。

现在我们已经注册了视图组件，并且在每次 `profile` 视图渲染的时候， `ProfileComposer@compose` 都将会被执行。接下来我们来看看这个类要如何定义：

```php
<?php namespace App\Http\ViewComposers;

use Illuminate\Contracts\View\View;
use Illuminate\Users\Repository as UserRepository;

class ProfileComposer {

    /**
     * The user repository implementation.
     *
     * @var UserRepository
     */
    protected $users;
```

```
    /**
     * Create a new profile composer.
     *
     * @param  UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        // service container 会自动解析所需的参数
        $this->users = $users;
    }

    /**
     * Bind data to the view.
     *
     * @param  View  $view
     * @return void
     */
    public function compose(View $view)
    {
        $view->with('count', $this->users->count());
    }

}
```

在视图被渲染之前，视图组件的 `compose` 方法就会被调用，并且传入一个 `Illuminate\Contracts\View\View` 实例。你可以使用 `with` 方法来把数据绑定到 `view`。

> 备注： 所有的视图组件会被 服务容器 (service container) 解析，所以你需要在视图组件的构造器类型限制你所需的任何依赖参数。

在视图组件内使用通配符

`View` 的 `composer` 方法可以接受 `*` 作为通配符，所以你可以对所有视图附加 `composer` 如下：

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

同时对多个视图附加视图组件

你也可以同时针对多个视图附加同一个视图组件：

```
View::composer(['profile', 'dashboard'], 'App\Http\ViewComposers\MyViewComposer');
```

定义多个视图组件

你可以使用 `composers` 方法来同时定义一群视图组件：

```
View::composers([
    'App\Http\ViewComposers\AdminComposer' => ['admin.index', 'admin.profile'],
    'App\Http\ViewComposers\UserComposer' => 'user',
    'App\Http\ViewComposers\ProductComposer' => 'product'
```

```
]);
```

视图创建者

视图 创建者 几乎和视图组件运作方式一样；只是视图创建者会在视图初始化后就立刻执行。要注册一个创建者，只要使用 `creator` 方法：

```
View::creator('profile', 'App\Http\ViewCreators\ProfileCreator');
```

服务提供者

简介

服务提供者是所有 Laravel 应用程序的启动中心。你的应用程序，以及所有 Laravel 的核心服务，都是透过服务提供者启动。

但我们所说的「启动」指的是什么？一般而言，我们指注册事物，包括注册服务容器绑定、事件监听器、过滤器，甚至路由。服务提供者是你的应用程序配置中心所在。

如果你开启包含于 Laravel 中 `config/app.php` 此一文件，你会看到 `providers` 数组。这些是所有将加载至你的应用程序里的服务提供者类。当然，它们之中有很多属于「缓载」提供者，意思是除非真正需要它们所提供的服务，否则它们并不会在每一个请求中都被加载。

在这份概述中，你会学到如何编写你自己的服务提供者，并将它们注册于你的 Laravel 应用程序。

基本提供者例子

所有的服务提供者都应继承 `Illuminate\Support\ServiceProvider` 此一类。在这个抽象类中，至少必须定义一个方法： `register` 。在 `register` 方法中，应该只绑定服务到服务容器之中。你永远不该试图在 `register` 方法中注册任何事件监听器、路由或任何其他功能。

Artisan 命令行接口可以很容易地通过 `make:provider` 产生新的提供者：

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

注册者方法

现在，让我们来看看基本的服务提供者：

```
<?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()
    {
```

```
        $this->app->singleton('Riak\Contracts\Connection', function($app)
        {
            return new Connection($app['config']['riak']);
        });
    }

}
```

这个服务提供者只定义了一个 `register` 方法，并在服务容器中使用此方法定义了一份 `Riak\Contracts\Connection` 的实现。若你还不了解服务容器是如何运作的，不用担心，我们很快会提到它。

此类位于 `App\Providers` 命名空间之下，因为这是 Laravel 中默认服务提供者所在的位置。然而，你可以随自己的需要改变它。你的服务提供者可被置于任何 Composer 能自动加载的位置。

启动方法

所以，若我们需要在服务提供者中注册一个事件监听器，该怎么做？它应该在 `boot` 方法中完成。这个方法会在所有的服务提供者注册后才被调用，这让你能取用框架中所有其他已注册过的服务。

```
<?php namespace App\Providers;

use Event;
use Illuminate\Support\ServiceProvider;

class EventServiceProvider extends ServiceProvider {

    /**
     * 执行注册后的启动服务。
     *
     * @return void
     */
    public function boot()
    {
        Event::listen('SomeEvent', 'SomeEventHandler');
    }

    /**
     * 在容器中注册绑定。
     *
     * @return void
     */
    public function register()
    {
        //
    }

}
```

我们可以对 `boot` 方法中的依赖作类型提示。服务容器会自动注入任何你所需要的依赖：

```
use Illuminate\Contracts\Events\Dispatcher;

public function boot(Dispatcher $events)
{
    $events->listen('SomeEvent', 'SomeEventHandler');
```

```
}
```

注册提供者

所有的服务提供者都在 `config/app.php` 此一配置文件中被注册。此文件包含了一个 `providers` 数组，你可以在其中列出你所有服务提供者的名称。此数组默认会列出一组 Laravel 的核心服务提供者。这些提供者启动了 Laravel 的核心组件，例如邮件发送者、队列、缓存及其他等等。

要注册你的提供者，只要把它加入此数组：

```
'providers' => [
    // 其他的服务提供者

    'App\Providers\AppServiceProvider',
],
```

缓载提供者

若你的提供者仅仅用于绑定注册到服务容器，你可以选择延缓其注册，直到真正需要其中注册的绑定才加载。延缓像这样的提供者加载可增进应用程序的性能，因为这样就不用每个请求都从文件系统中将其加载。

要延缓提供者加载，将 `defer` 性质设为 `true` ，并定义一个 `provides` 方法。 `provides` 方法应返回提供者所注册的服务容器绑定。

```
<?php namespace App\Providers;

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

class RiakServiceProvider extends ServiceProvider {

    /**
     * 指定是否延缓提供者加载。
     *
     * @var bool
     */
    protected $defer = true;

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

    /**
     * 取得提供者所提供的服务。
```

```
     *
     * @return array
     */
    public function provides()
    {
        return ['Riak\Contracts\Connection'];
    }

}
```

Laravel 编译并保存所有由延缓服务提供者所提供的服务清单，以及其服务提供者的类名称。只有在当你尝试解析其中的服务时， Laravel 才会加载服务提供者。

服务容器

介绍

Laravel 服务容器是管理类依赖的强力工具。依赖注入是比较专业的说法，真正意思是将类依赖透过构造器或「setter」方法注入。

让我们来看一个简单的例子：

```
<?php namespace App\Handlers\Commands;

use App\User;
use App\Commands\PurchasePodcast;
use Illuminate\Contracts\Mail\Mailer;

class PurchasePodcastHandler {

    /**
     * 一个发信功能的实现
     */
    protected $mailer;

    /**
     * 创建一个新的实例
     *
     * @param  Mailer  $mailer
     * @return void
     */
    public function __construct(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    /**
     * 购买一个播客节目
     *
     * @param  PurchasePodcastCommand  $command
     * @return void
     */
    public function handle(PurchasePodcastCommand $command)
    {
        //
    }

}
```

在这个例子中，当播客被购买时， `PurchasePodcast` 命令处理器需要发送一封电子邮件。所以，我们将注入一个服务来提供这个能力。当这个服务被注入以后，我们就可以轻易地切换到不同的实现。当测试我们的应用程序时，我们同样也可以轻易地「模拟」，或者创建一个虚拟的发信服务实现，来帮助我们进行测试。

如果要创建一个强大并且大型的应用，或者对 Laravel 的内核做贡献，首先必须对 Laravel 的服务容器进行深入了解。

基本用法

绑定

几乎你所有服务容器将与已注册的服务提供者绑定，这些例子都在情境(context)使用容器做说明，如果应用程序其它地方需要容器实例，如工厂(factory)，能以类型提示 `Illuminate\Contracts\Container\Container` 注入一个容器实例。另外，你可以使用 `App` facade 访问容器。

注册基本解析器

在一个服务提供者内部，你总是可以通过 `$this->app` 实例变量来访问到容器。

在服务提供者里，总是通过 `$this->app` 实例变量使用容器。

服务容器注册依赖有几种方式，包括闭包回调和绑定实例的接口。首先，我们来探讨闭包回调的方式。被注册至容器的闭包解析器包含一个 key (通常用类名称) 和一个有返回值的闭包:

```
$this->app->bind('FooBar', function($app)
{
    return new FooBar($app['SomethingElse']);
});
```

注册一个单例

有时候，你可能希望绑定到容器的对象只会被解析一次，之后的调用都返回相同的实例：

```
$this->app->singleton('FooBar', function($app)
{
    return new FooBar($app['SomethingElse']);
});
```

绑定一个已经存在的实例

你也可以使用 `instance` 方法，绑定一个已经存在的实例到容器，接下来将总是返回该实例：

```
$fooBar = new FooBar(new SomethingElse);

$this->app->instance('FooBar', $fooBar);
```

解析

从容器解析出实例有几种方式。
一、可以使用 `make` 方法：

```
$fooBar = $this->app->make('FooBar');
```

二、你可以像「访问数组」一样对容器进行访问，因为它实现了PHP的 `ArrayAccess` 接口：

```
$fooBar = $this->app['FooBar'];
```

最后，也是最重要的一点，你可以在构造函数中简单地「类型指定（type-hint）」你所需要的依赖，包括在控制器、事件监听器、队列任务，过滤器等等之中。容器将自动注入你所需的所有依赖：

```
<?php namespace App\Http\Controllers;

use Illuminate\Routing\Controller;
use App\Users\Repository as UserRepository;

class UserController extends Controller {

    /**
     * The user repository instance.
     */
    protected $users;

    /**
     * Create a new controller instance.
     *
     * @param  UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }

    /**
     * Show the user with the given ID.
     *
     * @param  int  $id
     * @return Response
     */
    public function show($id)
    {
        //
    }

}
```

将接口绑定到实现

注入具体依赖

服务容器有个非常强大特色，能够绑定特定实例的接口。举例，假设我们应用程序要集成 Pusher 服务去收发即时事件，如果使用 Pusher 的 PHP SDK，可以在类注入一个 Pusher 客户端实例：

```
<?php namespace App\Handlers\Commands;

use App\Commands\CreateOrder;
use Pusher\Client as PusherClient;

class CreateOrderHandler {

    /**
     * Pusher SDK 客户端实例
     */
    protected $pusher;

    /**
     * 创建一个实例
     *
     * @param  PusherClient  $pusher
     * @return void
     */
    public function __construct(PusherClient $pusher)
    {
        $this->pusher = $pusher;
    }

    /**
     * 执行命令
     *
     * @param  CreateOrder  $command
     * @return void
     */
    public function execute(CreateOrder $command)
    {
        //
    }

}
```

在上面这个例子中，注入类的依赖到类中已经能够满足需求；但同时，我们也紧密耦合于 Pusher 的 SDK。如果 Pusher 的 SDK 方法发生改变，或者我们要切换到别的事件服务，那我们也需要同时修改 `CreateOrderHandler` 的代码。

为接口编程

为了将 `CreateOrderHandler` 和事件推送的修改「隔离」，我们可以定义一个 `EventPusher` 接口和一个 `PusherEventPusher` 实现：

```
<?php namespace App\Contracts;

interface EventPusher {

    /**
     * Push a new event to all clients.
     *
     * @param  string  $event
     * @param  array  $data
```

```
     * @return void
     */
    public function push($event, array $data);

}
```

一旦 `PusherEventPusher` 实现这接口，就可以在服务容器像这样注册它：

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

当有类需要 `EventPusher` 接口时，会告诉容器应该注入 `PusherEventPusher`，现在就可以在构造器中「类型指定」一个 `EventPusher` 接口：

```
    /**
     * Create a new order handler instance.
     *
     * @param  EventPusher  $pusher
     * @return void
     */
    public function __construct(EventPusher $pusher)
    {
        $this->pusher = $pusher;
    }
```

上下文绑定

有时候，你可能会有两个类需要用到同一个接口，但是你希望为每个类注入不同的接口实现。例如当我们的系统收到一个新的订单时，我们需要使用 PubNub 来代替 Pusher 发送消息。Laravel 提供了一个简单便利的接口来定义以上的行为：

```
$this->app->when('App\Handlers\Commands\CreateOrderHandler')
          ->needs('App\Contracts\EventPusher')
          ->give('App\Services\PubNubEventPusher');
```

标签

偶尔你可能需要解析绑定中的某个「类」。例如你正在建设一个汇总报表，它需要接收实现了 `Report` 接口的不同实现的数组。在注册了 `Report` 的这些实现之后，你可以用 `tag` 方法来给他们赋予一个标签：

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

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

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

一旦服务打上标签，可以通过 `tagged` 方法轻易地解析它们：

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

实际应用

Laravel 提供了几个机会来使用服务容器以提高应用程序的灵活性和可测试性。解析控制器是一个最主要的案例。所有的控制器都通过服务容器来进行解析，意味着你可以在控制器的构造函数中「类型指定」所需依赖，而且它们将被自动注入。

```
<?php namespace App\Http\Controllers;

use Illuminate\Routing\Controller;
use App\Repositories\OrderRepository;

class OrdersController extends Controller {

    /**
     * The order repository instance.
     */
    protected $orders;

    /**
     * Create a controller instance.
     *
     * @param  OrderRepository  $orders
     * @return void
     */
    public function __construct(OrderRepository $orders)
    {
        $this->orders = $orders;
    }

    /**
     * Show all of the orders.
     *
     * @return Response
     */
    public function index()
    {
        $orders = $this->orders->all();

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

}
```

在这个例子中，`OrderRepository` 类将被自动注入到控制器中。这意味着在进行 单元测试 时，我们可以绑定一个假的 `OrderRepository` 到容器中来代替我们对数据库的真实操作，避免对真实数据库的影响。

使用容器的其他几个例子

当然，在上面提到过的，控制器并不是 Laravel 通过服务容器进行解析的唯一类。你也可以在路由的闭包中、过滤器中、队列任务中、事件监听器中来「类型指定」你所需要的依赖。对于在这些情境中如何使用服务容器，请参考相关文档。

容器事件

注册一个解析事件监听器

容器在解析每一个对象时就会触发一个事件。你可以用 `resolving` 方法来监听此事件：

```
$this->app->resolving(function($object, $app)
{
    // 当容器解析任意类型的依赖时被调用
});

$this->app->resolving(function(FooBar $fooBar, $app)
{
    // 当容器解析 `FooBar` 类型的依赖时被调用
});
```

被解析的对象将被传入到闭包方法中。

Contracts

简介

Laravel 的 Contracts 是一组定义了框架核心服务的接口（ interfaces ）。例如，`Queue` contract 定义了队列任务所需要的方法，而 `Mailer` contract 定义了发送 e-mail 需要的方法。

在 Laravel 框架里，每个 contract 都提供了一个对应的实现。例如， Laravel 提供了有多种驱动的 `Queue` 的实现，而根据 SwiftMailer 实现了 `Mailer`。

Laravel 所有的 contracts 都放在各自的 Github repository。除了提供了所有可用的 contracts 一个快速的参考，也可以单独作为一个低耦合的扩展包让其他扩展包开发者使用。

为什么用 **Contracts**？

你可能有很多关于 contracts 的问题。如为什么要使用接口？使用接口会不会变的更复杂？

让我们用下面的标题来解释为什么要使用接口：低耦合和简单性。

低耦合

首先，看一些强耦合的缓存实现代码。如下：

```
<?php namespace App\Orders;

class Repository {

    /**
     * The cache.
     */
    protected $cache;

    /**
     * Create a new repository instance.
     *
     * @param  \SomePackage\Cache\Memcached  $cache
     * @return void
     */
    public function __construct(\SomePackage\Cache\Memcached $cache)
    {
        $this->cache = $cache;
    }

    /**
     * Retrieve an Order by ID.
     *
     * @param  int  $id
```

```
     * @return Order
     */
    public function find($id)
    {
        if ($this->cache->has($id))
        {
            //
        }
    }

}
```

在上面的类里，代码跟缓存实现之间是强耦合。理由是它会依赖于扩展包库（ package vendor ）的特定缓存类。一旦这个扩展包的 API 更改了，我们的代码也要跟着改变。

同样的，如果想要将底层的缓存技术（比如 Memcached ）抽换成另一种（像 Redis ），又一次的我们必须修改这个 repository 类。我们的 repository 不应该知道这么多关于谁提供了数据，或是如何提供等等细节。

比起上面的做法，我们可以改用一个简单、和扩展包无关的接口来改进代码：

```
<?php namespace App\Orders;

use Illuminate\Contracts\Cache\Repository as Cache;

class Repository {

    /**
     * Create a new repository instance.
     *
     * @param  Cache  $cache
     * @return void
     */
    public function __construct(Cache $cache)
    {
        $this->cache = $cache;
    }

}
```

现在上面的代码没有跟任何扩展包耦合，甚至是 Laravel。既然 contracts 扩展包没有包含实现和任何依赖，你可以很简单的对任何 contract 进行实现，你可以很简单的写一个替换的实现，甚至是替换 contracts，让你可以替换缓存实现而不用修改任何用到缓存的代码。

简单性

当所有的 Laravel 服务都简洁的使用简单的接口定义，就能够很简单的决定一个服务需要提供的功能。 可以将 **contracts** 视为说明框架特色的简洁文档。

除此之外，当你依赖简洁的接口，你的代码能够很简单的被了解和维护。比起搜索一个大型复杂的类里有哪些可用的方法，你有一个简单，干净的接口可以参考。

Contract 参考

以下是大部分 Laravel Contracts 的参考，以及相对应的 "facade"

Contract	Laravel 4.x Facade
Illuminate\Contracts\Auth\Guard	Auth
Illuminate\Contracts\Auth\PasswordBroker	Password
Illuminate\Contracts\Bus\Dispatcher	Bus
Illuminate\Contracts\Cache\Repository	Cache
Illuminate\Contracts\Cache\Factory	Cache::driver()
Illuminate\Contracts\Config\Repository	Config
Illuminate\Contracts\Container\Container	App
Illuminate\Contracts\Cookie\Factory	Cookie
Illuminate\Contracts\Cookie\QueueingFactory	Cookie::queue()
Illuminate\Contracts\Encryption\Encrypter	Crypt
Illuminate\Contracts\Events\Dispatcher	Event
Illuminate\Contracts\Filesystem\Cloud	
Illuminate\Contracts\Filesystem\Factory	File
Illuminate\Contracts\Filesystem\Filesystem	File
Illuminate\Contracts\Foundation\Application	App
Illuminate\Contracts\Hashing\Hasher	Hash
Illuminate\Contracts\Logging\Log	Log
Illuminate\Contracts\Mail\MailQueue	Mail::queue()
Illuminate\Contracts\Mail\Mailer	Mail
Illuminate\Contracts\Queue\Factory	Queue::driver()
Illuminate\Contracts\Queue\Queue	Queue
Illuminate\Contracts\Redis\Database	Redis
Illuminate\Contracts\Routing\Registrar	Route
Illuminate\Contracts\Routing\ResponseFactory	Response
Illuminate\Contracts\Routing\UrlGenerator	URL
Illuminate\Contracts\Support\Arrayable	
Illuminate\Contracts\Support\Jsonable	
Illuminate\Contracts\Support\Renderable	
Illuminate\Contracts\Validation\Factory	Validator::make()
Illuminate\Contracts\Validation\Validator	
Illuminate\Contracts\View\Factory	View::make()
Illuminate\Contracts\View\View	

如何使用 **Contracts**

所以，要如何实现一个 contract？实际上非常的简单。很多 Laravel 的类都是经由 service container 解析，包含控制器，事件监听，过滤器，队列任务，甚至是闭包。所以，要实现一个 contract，你可以在类的构造器使用「类型提示」解析类。例如，看下面的事件处理程序：

```php
<?php namespace App\Handlers\Events;

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

class CacheUserInformation {

    /**
     * Redis 数据库实现
     */
    protected $redis;

    /**
     * 建立新的事件处理实例
     *
     * @param  Database  $redis
     * @return void
     */
    public function __construct(Database $redis)
    {
        $this->redis = $redis;
    }

    /**
     * 处理事件
     *
     * @param  NewUserRegistered  $event
     * @return void
     */
    public function handle(NewUserRegistered $event)
    {
        //
    }

}
```

当事件监听被解析时，服务容器会经由类构造器参数的类型提示，注入适当的值。要知道怎么注册更多服务容器，参考这个文档。

Facades

介绍

Facades 提供一个静态接口给在应用程序的 服务容器 中可以取用的类。Laravel 附带许多 facades，甚至你可能已经在不知情的状况下使用过它们！Laravel 的「facades」作为在 IoC 容器里面的基础类的静态代理，提供的语法有简洁、易表达的优点，同时维持比传统的静态方法更高的可测试性和弹性。

有时，你或许会希望为应用程序和扩展包建立自己的 facades，所以让我们来探索这些类的概念、开发和用法。

> 注意： 在深入了解 facades 之前，强烈建议你先熟悉 Laravel IoC 容器.

解释

在 Laravel 应用程序的环境中，facade 是个提供从容器访问对象的类。`Facade` 类是让这个机制可以运作的原因。Laravel 的 facades 和你建立的任何自定义 facades，将会继承基本的 `Facade` 类。

你的 facade 类只需要去实现一个方法：`getFacadeAccessor`。`getFacadeAccessor` 方法的工作是定义要从容器解析什么。基本的 `Facade` 类利用 `__callStatic()` 魔术方法来从你的 facade 调用到解析出来的对象。

所以当你对 facade 调用，例如 `Cache::get`，Laravel 从 IoC 容器解析缓存管理类出来，并对该类调用 `get` 方法。用专业口吻来说，Laravel Facades 是使用 Laravel IoC 容器作为服务定位器的便捷语法。

实际用法

在下面的例子，对 Laravel 缓存系统进行调用。简单看过去这代码，有人可能会以为静态方法 `get` 是对 `Cache` 类调用。

```
$value = Cache::get('key');
```

然而，如果我们去看 `Illuminate\Support\Facades\Cache` 类，你将会看到它没有静态方法 `get`：

```
class Cache extends Facade {

    /**
     * 取得组件的注册名称
     *
     * @return string
```

```
     */
    protected static function getFacadeAccessor() { return 'cache'; }

}
```

Cache 类继承基本的 `Facade` 类并定义一个 `getFacadeAccessor()` 方法。记住，这个方法的工作是返回 IoC 绑定的名称。

当用户在 `Cache` 的 facade 上参考任何的静态方法，Laravel 会从 IoC 容器解析被绑定的 `cache` ，并对该对象执行被请求的方法 (在这个例子中， `get`)。

所以我们的 `Cache::get` 调用可以被重写成像这样：

```
$value = $app->make('cache')->get('key');
```

导入 **Facades**

记住，如果你在控制器有使用命名空间的情况下使用 facade，你会需要导入 facade 类进入命名空间。所有的 facades 存在于全局命名空间：

```
<?php namespace App\Http\Controllers;

use Cache;

class PhotosController extends Controller {

    /**
     * 取得所有的应用程序相片。
     *
     * @return Response
     */
    public function index()
    {
        $photos = Cache::get('photos');

        //
    }

}
```

建立 **Facades**

为你自己的应用程序或扩展包建立 facade 是很简单的。你只需要 3 个东西：

- 一个 IoC 绑定。
- 一个 facade 类。
- 一个 facade 别名配置。

让我们来看个例子。这里有一个定义为 `PaymentGateway\Payment` 的类。

```
namespace PaymentGateway;
```

```
class Payment {

    public function process()
    {
        //
    }

}
```

我们需要可以从 IoC 容器解析出这个类。所以，让我们来加上一个绑定到服务提供者：

```
App::bind('payment', function()
{
    return new \PaymentGateway\Payment;
});
```

注册这个绑定的好方式是建立新的 服务提供者 命名为 `PaymentServiceProvider`，并把这个绑定加到 `register` 方法。然后你可以配置 Laravel 从 `config/app.php` 配置文件加载你的服务提供者。

接下来，我们可以建立我们自己的 facade 类：

```
use Illuminate\Support\Facades\Facade;

class Payment extends Facade {

    protected static function getFacadeAccessor() { return 'payment'; }

}
```

最后，如果我们希望，可以在 `config/app.php` 配置文件为 facade 加个别名到 `aliases` 数组。现在我们可以在 `Payment` 类的实例上调用 `process` 方法。

```
Payment::process();
```

自动加载别名的附注

在 `aliases` 数组中的类在某些实例中不能使用，因为 PHP 将不会尝试去自动加载未定义的类型提示类。如果 `\ServiceWrapper\ApiTimeoutException` 命别名为 `ApiTimeoutException`，即便有异常被抛出，在 `\ServiceWrapper` 命名空间外面的 `catch(ApiTimeoutException $e)` 将永远捕捉不到异常。类似的问题在有类型提示的别名类一样会发生。唯一的替代方案就是放弃别名并用 `use` 在每一个文件的最上面引入你希望类型提示的类。

模拟 Facades

单元测试是为什么现在 facades 采用这样的工作方式的主要因素。事实上，可测试性甚至是 facades 存在的主要理由。想要获得更多信息，请查看文档的 模拟 facades 章节。

Facade 类参考

你将会在下面找到每一个 facade 和它的基础类。这是个可以从一个给定的 facade 根源快速地深入 API 文档的有用工具。可应用的 IoC 绑定 关键字也包含在里面。

Facade	Class	IoC Binding
App	Illuminate\Foundation\Application	`app`
Artisan	Illuminate\Console\Application	`artisan`
Auth	Illuminate\Auth\AuthManager	`auth`
Auth (实例)	Illuminate\Auth\Guard	
Blade	Illuminate\View\Compilers\BladeCompiler	`blade.compiler`
Bus	Illuminate\Contracts\Bus\Dispatcher	
Cache	Illuminate\Cache\Repository	`cache`
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	
Event	Illuminate\Events\Dispatcher	`events`
File	Illuminate\Filesystem\Filesystem	`files`
Hash	Illuminate\Contracts\Hashing\Hasher	`hash`
Input	Illuminate\Http\Request	`request`
Lang	Illuminate\Translation\Translator	`translator`
Log	Illuminate\Log\Writer	`log`
Mail	Illuminate\Mail\Mailer	`mailer`
Password	Illuminate\Auth\Passwords\PasswordBroker	`auth.password`
Queue	Illuminate\Queue\QueueManager	`queue`
Queue (实例)	Illuminate\Queue\QueueInterface	
Queue (基础类)	Illuminate\Queue\Queue	
Redirect	Illuminate\Routing\Redirector	`redirect`
Redis	Illuminate\Redis\Database	`redis`
Request	Illuminate\Http\Request	`request`
Response	Illuminate\Contracts\Routing\ResponseFactory	
Route	Illuminate\Routing\Router	`router`
Schema	Illuminate\Database\Schema\Blueprint	
Session	Illuminate\Session\SessionManager	`session`
Session (实例)	Illuminate\Session\Store	
Storage	Illuminate\Contracts\Filesystem\Factory	`filesystem`
URL	Illuminate\Routing\UrlGenerator	`url`
Validator	Illuminate\Validation\Factory	`validator`
Validator (实例)	Illuminate\Validation\Validator	
View	Illuminate\View\Factory	`view`

View (实例)	Illuminate\View\View	

请求的生命周期

简介

当您使用「真实世界」中的任何工具时，若能了解它是如何运作的，您会更具信心。开发应用程序也是一样。当您明白您的开发工具运作的方式，使用它们时，您会感到更舒适、更有信心。

这份文档的目的在给予您一个优良且高端的概述，关于 Laravel 框架是如何「运作」的。当您越了解整个框架，这些事情便不再那么令人感到「神奇」，而您在建立应用程序时也会更具信心。

若您目前还无法了解所有的术语，不要灰心！只要试着对现在提到的东西有个基本掌握，您的知识将会随着您探索这份文档其他章节的同时跟着成长。

生命周期概要

首要之事

`public/index.php` 这个文件是对 Laravel 应用程序所有请求的进入点。所有的请求都通过您网页服务器（Apache / Ngix）的配置导向这个文件。 `index.php` 这个文件并没有太多的代码。更确切地说，它只是个起始点，用来加载框架其他的部分。

`index.php` 加载由 Composer 产生的自动加载器定义，并接收由 `bootstrap/app.php` 文件所产生的 Laravel 应用程序实例。Laravel 自身的第一个动作就是创建一个应用程序 / 服务容器的实例。

HTTP / 终端核心

接下来，进入应用程序的请求的会被送往 HTTP 核心或终端核心，视该请求的种类而定。这两种核心是所有请求流向的中心位置。现在开始，我们只将焦点放在 HTTP 核心，它位于 `app/Http/Kernel.php`。

HTTP 核心扩展了 `Illuminate\Foundation\Http\Kernel` 类，它定义了一个 `bootstrappers` 数组，在请求被执行前会执行。这些启动器（bootstrappers）会进行配置错误处理，日志记录，侦测应用程序环境，以及其他在请求真正被处理之前，需要完成的工作。

HTTP 核心也定义了一份 HTTP 中间件清单，所有的请求在被应用程序处理之前都必须经过它们。这些中间件有负责处理 HTTP session 的读写，决定应用程序是否处于维护模式，查验跨站请求伪造（CSRF）标记，以及其他更多的功能。

HTTP 核心 `handle` 方法的方法签名相当简单：它接收一个 `Request` 并返回一个 `Response`。把核心想像成一个大的黑盒子，用来代表你整个的应用程序。对它输入 HTTP 请求，它将返回 HTTP 响应。

服务提供者

最重要的核心启动行为之一，是加载您的应用程序的服务提供者。所有应用程序的服务提供者，都在 `config/app.php` 配置文件的 `providers` 数组中被配置。首先，对所有的提供者调用 `register` 方法，一旦所有的提供者都被注册之后，`boot` 方法也会被调用。

请求分派

当应用程序启动且所有的服务提供者都被注册之后，`Request` 将被移转给路由器进行分派。路由器会将请求分派给路由或控制器，并执行任何特定路由的中间件。

聚焦于服务提供者

服务提供者是启动 Laravel 应用程序的真正关键。创建应用程序实例，注册服务提供者，并将请求移转至已启动的应用程序。真的就是这么简单！

能确实掌握 Laravel 应用程序是如何建立，并通过服务提供者启动是很有价值的。当然，您应用程序的默认服务提供者存放在 `app/Providers` 这个目录中。

`AppServiceProviders` 默认几乎是空的。此提供者是一个很好的地方，可让您加入您应用程序自身的启动及对服务容器的绑定。当然，对大型应用程序而言，您可能希望创建若干个服务提供者，每一个都具备更精细的启动类型。

应用程序结构

介绍

默认的 Laravel 应用程序结构是为了提供一个良好的开始，无论是构建大型还是小型应用。当然，你可以依照喜好自由地组织应用程序。Laravel 几乎没有强加限制任何类的放置位置 - 只要 Composer 可以自动加载这些类即可。

根目录

一个新安装的 Laravel 根目录包含许多个目录：

`app` 目录，如你所料，包含应用程序的核心代码。我们之后将会很快深入探讨这个目录的细节。

`bootstrap` 目录包含几个框架启动跟自动加载配置的文件。

`config` 目录，顾名思义，包含所有应用程序的配置文件。

`database` 目录包含你的数据库迁移与数据填充文件。

`public` 目录包含前面的控制器和你的资源文件 (图片、JavaScript、CSS，等等)。

`resources` 目录包含你的视图、原始的资源文件 (LESS、SASS、CoffeeScript) 和「语言」文件。

`storage` 目录包含编译后的 Blade 模板、基于文件的 session、文件缓存和其他框架产生的文件。

`tests` 目录包含你的自动化测试。

`vendor` 目录包含你的 Composer 依赖模块。

App 目录

应用程序的「内容」存在于 `app` 目录中。默认情况下，这个目录在 `App` 命名空间下并通过 Composer 使用 PSR-4 自动加载标准 自动加载。 你可以使用 `app:name` **Artisan** 命令变更这个命名空间.

`app` 目录附带许多个额外的目录，例如：`Console`、 `Http` 和 `Providers`。考虑 `Console` 和 `Http` 目录用作提供 API 进入应用程序的「核心」。HTTP 协定和 CLI 都是跟应用程序交互的机制，但实际上并不包含应用程序逻辑。换句话说，它们是两种简单地发布命令给应用程序的方法。 `Console` 目录包含你全部的 Artisan 命令，而 `Http` 目录包含你的控制器、过滤器和请求。

`Commands` 目录当然是用来放置应用程序的命令。命令代表可以被应用程序放到队列的任务，以及可以在当前请求生命周期内同步运行的任务。

`Events` 目录，如你所料，是用来放置事件类。当然，使用类来代表事件不是必须的；然而，如果你选择

使用它们，这个目录将会是通过 Artisan 命令行创建它们时的默认位置。

`Handlers` 目录包含命令和事件的处理类。处理进程接收命令或事件，并针对该命令或事件执行逻辑。

`Services` 目录包含各种「辅助」服务，囊括应用程序需要的功能。例如，Laravel 引入的 `Registrar` 服务负责验证 并创建应用程序的新用户。其他的例子可能是服务跟外部 API、评价系统或甚至是跟从你的应用程序汇集数据的服务交互。

`Exceptions` 目录包含应用程序的异常处理进程，也是个处置应用程序抛出的任何异常的好地方。

> 注意： 在 `app` 目录中的许多类可以用 Artisan 命令产生。要查看可以使用的命令，在终端机执行 `php artisan list make` 命令。

为应用程序配置命名空间

如前面所提到的，默认的应用程序命名空间为 `App` ；然而，你可以变更这个命名空间成跟应用程序的名称一样，这可以简单地通过 `app:name` Artisan 命令完成。例如：如果你的应用程序叫做「SocialNet」，你将会执行下面的命令：

```
php artisan app:name SocialNet
```

认证

介绍

Laravel 让实现认证机制变得非常简单。事实上，几乎所有的设置默认就已经完成了。有关认证的配置文件都放在 `config/auth.php` 里，而在这些文件里也都包含了良好的注释描述每一个选项的所对应的认证服务。

Laravel 默认在 `app` 文件夹内就包含了一个使用默认 Eloquent 认证驱动的 `App\User` 模型。

注意：当为这个认证模型设计数据库结构时，密码字段至少有60个字符宽度。同样，在开始之前，请先确认您的 `users` (或其他同义) 数据库表包含一个名为 `remember_token` 长度为 100 的 `string` 类型、可接受 null 的字段。这个字段将会被用来储存「记住我」的 session token。也可以通过在迁移文件中使用 `$table->rememberToken();` 方法。 当然， Laravel 5 自带的 migrations 里已经设定了这些字段。

假如您的应用程序并不是使用 Eloquent ，您也可以使用 Laravel 的查询构造器做 `database` 认证驱动。

用户认证

Laravel 已经预设了两个认证相关的控制器。 `AuthController` 处理新的用户注册和「登陆」，而 `PasswordController` 可以帮助已经注册的用户重置密码。

每个控制器使用 trait 引入需要的方法。在大多数应用上，你不需要修改这些控制器。这些控制器用到的视图放在 `resources/views/auth` 目录下。你可以依照需求修改這些视图。

用户注册

要修改应用注册新用户时所用到的表单字段，可以修改 `App\Services\Registrar` 类。这个类负责验证和建立应用的新用户。

`Registrar` 的 `validator` 方法包含新用户时的验证规则，而 `Registrar` 的 `create` 方法负责在数据库中建立一条新的 `User` 记录。你可以自由的修改这些方法。 `Registrar` 方法是通过 `AuthenticatesAndRegistersUsers` trait 的中的 `AuthController` 调用的。

手动认证

如果你不想使用预设的 `AuthController` ，你需要直接使用 Laravel 的身份验证类来管理用户认证。别担心，这也很简单的！首先，让我们看看 `attempt` 方法：

```php
<?php namespace App\Http\Controllers;

use Auth;
use Illuminate\Routing\Controller;

class AuthController extends Controller {

    /**
     * Handle an authentication attempt.
     *
     * @return Response
     */
    public function authenticate()
    {
        if (Auth::attempt(['email' => $email, 'password' => $password]))
        {
            return redirect()->intended('dashboard');
        }
    }

}
```

`attempt` 方法可以接受由键值对组成的数组作为第一个参数。`password` 的值会先进行 哈希。数组中的其他 值会被用来查询数据表里的用户。所以，在上面的示例中，会根据 `email` 列的值找出用户。如果找到该用户，会比对数据库中存储的哈希过的密码以及数组中的哈希过后的 `password` 值。假设两个哈希后的密码相同，会重新为用户启动认证通过的 session。

如果认证成功， `attempt` 将会返回 `true`。否则则返回 `false`。

> 注意：在上面的示例中，并不一定要使用 `email` 字段，这只是作为示例。你应该使用对应到数据表中的「username」的任何键值。

`intended` 方法会重定向到用户尝试要访问的 URL ， 其值会在进行认证过滤前被存起来。也可以给这个方法传入一个预设的 URI，防止重定向的网址无法使用。

以特定条件验证用户

在认证过程中，你可能会想要加入额外的认证条件：

```php
if (Auth::attempt(['email' => $email, 'password' => $password, 'active' => 1]))
{
    // The user is active, not suspended, and exists.
}
```

判断用户是否已验证

判断一个用户是否已经登录，你可以使用 `check` 方法：

```php
if (Auth::check())
{
    // The user is logged in...
}
```

认证用户并且「记住」他

假如你想要在应用中提供「记住我」的功能，你可以传入布尔值作为 `attempt` 方法的第二个参数，这样就可以保留用户的认证身份（或直到他手动登出为止）。当然，你的 `users` 数据表必需包括一个字符串类型的 `remember_token` 列來储存「记住我」的标识。

```
if (Auth::attempt(['email' => $email, 'password' => $password], $remember))
{
    // The user is being remembered...
}
```

假如有使用「记住我」功能，可以使用 `viaRemember` 方法判定用户是否拥有「记住我」的 cookie 來判定用户认证：

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

以 **ID** 认证用户

要通过 ID 来认证用户，使用 `loginUsingId` 方法：

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

验证用户信息而不登陆

`validate` 方法可以让你验证用户凭证信息而不用真的登陆应用：

```
if (Auth::validate($credentials))
{
    //
}
```

在单一请求内登陆用户

你也可以使用 `once` 方法來让用户在单一请求内登陆。不会有任何 session 或 cookie 产生：

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

手动登陆用户

假如你需要将一个已经存在的用户实例登陆应用，你可以调用 `login` 方法并且传入用户实例：

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

这个方式和使用 `attempt` 方法验证用户凭证信息是一样的。

用户登出

```
Auth::logout();
```

当然，假设你使用 Laravel 内建的认证控制器，预设提供了让用户登出的方法。

认证事件

当 `attempt` 方法被调用时，`auth.attempt` 事件 会被触发。假设用户尝试认证成功并且登陆了，`auth.login` 事件会被触发。

取得经过认证的用户

当用户通过认证后，有几种方式取得用户实例。

首先， 你可以从 `Auth` facade 取得用户：

```
<?php namespace App\Http\Controllers;

use Illuminate\Routing\Controller;

class ProfileController extends Controller {

    /**
     * Update the user's profile.
     *
     * @return Response
     */
    public function updateProfile()
    {
        if (Auth::user())
        {
            // Auth::user() returns an instance of the authenticated user...
        }
    }

}
```

第二种，你可以使用 `Illuminate\Http\Request` 实例取得认证过的用户：

```
<?php namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

class ProfileController extends Controller {

    /**
     * Update the user's profile.
     *
     * @return Response
```

```
     */
    public function updateProfile(Request $request)
    {
        if ($request->user())
        {
            // $request->user() returns an instance of the authenticated user...
        }
    }

}
```

第三，你可以使用 `Illuminate\Contracts\Auth\Authenticatable` contract 类型提示。这个类型提示可以用在控制器的构造方法，控制器的其他方法，或是其他可以通过服务容器 解析的类的构造方法：

```
<?php namespace App\Http\Controllers;

use Illuminate\Routing\Controller;
use Illuminate\Contracts\Auth\Authenticatable;

class ProfileController extends Controller {

    /**
     * Update the user's profile.
     *
     * @return Response
     */
    public function updateProfile(Authenticatable $user)
    {
        // $user is an instance of the authenticated user...
    }

}
```

保护路由

路由中间件 只允许通过认证的用户访问指定的路由。Laravel 默认提供了 `auth` 中间件，放在 `app\Http\Middleware\Authenticate.php`。 你需要做的只是将其加到一个路由定义中：

```
// With A Route Closure...

Route::get('profile', ['middleware' => 'auth', function()
{
    // Only authenticated users may enter...
}]);

// With A Controller...

Route::get('profile', ['middleware' => 'auth', 'uses' => 'ProfileController@show']);
```

HTTP 基本认证

HTTP 基本认证提供了一个快速的方式来认证用户而不用特定设置一个「登入」页。在您的路由内设定 `auth.basic` 中间件则可启动这个功能：

用 **HTTP** 基本认证保护路由

```
Route::get('profile', ['middleware' => 'auth.basic', function()
{
    // Only authenticated users may enter...
}]);
```

默认情况下 `basic` 中间件会使用用户的 `email` 列当做「 username 」。

设定无状态的 **HTTP** 基本过滤器

你可能想要使用 HTTP 基本认证，但不会在 session 里设置用户身份的 cookie，这在 API 认证时特别有用。如果要這樣做，定义一个中间件并调用 `onceBasic` 方法：

```
public function handle($request, Closure $next)
{
    return Auth::onceBasic() ?: $next($request);
}
```

如果你使用 PHP FastCGI，HTTP 基本认证可能无法正常运行。请在你的 `.htaccess` 文件内新增以下代码：

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

忘记密码与重设

模型与数据表

大多数的 web 应用程序都会提供用户忘记密码的功能。为了不让开发者重复实现这个功能，Laravel 提供了方便的方法来发送忘记密码通知及密码重设的功能。

在开始之前，请先确认您的 `User` 模型实现了 `Illuminate\Contracts\Auth\CanResetPassword` 接口。当然，默认 Laravel 的 `User` 模型本身就已实现，并且引入 `Illuminate\Auth\Passwords\CanResetPassword` 来包括所有需要实现的接口方法。

生成 **Reminder** 数据表迁移

接下来，我们需要生成一个数据库表来储存重设密码标志。Laravel 默认已经包含了这个迁移表，放在 `database/migrations` 的目录下。你所需要作的只有执行迁移：

```
php artisan migrate
```

密码重设控制器

Laravel 还包含了 `Auth\PasswordController` 其中包含重设用户密码的功能。甚至一些视图，可以让你直接开始使用！视图放在 `resources/views/auth` 目录下。你可以按照你的应用程序设计，自由的修改這些

视图。

你的使用者会收到一封 e-mail，内含连接指向 `PasswordController` 中的 `getReset` 方法。这个方法会显示密码重设表单，允许用户重新设定密码。在密码重新设定完之后，用户将会自动登录到应用中，然后被重定向到 `/home`。你可以通过 `PasswordController` 中的 `redirectTo` 来定义重设密码后要重定向的位置：

```
protected $redirectTo = '/dashboard';
```

> 注意：默认情况下，密码重设 tokens 会在一小时后过期。你可以修改 `config/auth.php` 文件中的 `reminder.expire` 更改 这个设定。

第三方登陆认证

除了传统的以表单进行的认证，Laravel 还提供了简单、易用的方式，使用 Laravel Socialite 进行 OAuth 认证。**Socialite** 目前支持的认证有 **Facebook**、 **Twitter**、**Google**、以及**GitHub** 和 **Bitbucket** 。

如果要开始使用第三方认证，请将下面的代码加入到你的 `composer.json` 文件内：

```
"laravel/socialite": "~2.0"
```

接下來，在你的 `config/app.php` 配置文件中注册 `Laravel\Socialite\SocialiteServiceProvider`。也可以注册 facade：

```
'Socialize' => 'Laravel\Socialite\Facades\Socialite',
```

你需要在应用程序中加入 OAuth 服务所需的凭证。这些凭证都放在 `config/services.php` 配置文件里，并根据应用的需求使用 `facebook`、 `twitter`、 `google` 或 `github` 作为对应的键值。例如：

```
'github' => [
    'client_id' => 'your-github-app-id',
    'client_secret' => 'your-github-app-secret',
    'redirect' => 'http://your-callback-url',
],
```

接下來就准备认证用户了！你会需要两个路由：一个用于将用户重定向至认证提供网站，另一个用于认证之后，从认证服务接收回调。下面是一个使用 `Socialize` facade 的示例：

```
public function redirectToProvider()
{
    return Socialize::with('github')->redirect();
}

public function handleProviderCallback()
{
    $user = Socialize::with('github')->user();

    // $user->token;
```

```
}
```

`redirect` 方法将用户重定向到认证 OAuth 的网站，而 `user` 方法会获取返回的请求，以及从认证网站取得的用户信息。在重定向至用户之前，你也可以设定请求的「 scopes 」：

```
return Socialize::with('github')->scopes(['scope1', 'scope2'])->redirect();
```

一旦你取得用户实例，你能获取到更多的用户详细信息：

获取用户资料

```
$user = Socialize::with('github')->user();

// OAuth Two Providers
$token = $user->token;

// OAuth One Providers
$token = $user->token;
$tokenSecret = $user->tokenSecret;

// All Providers
$user->getId();
$user->getNickname();
$user->getName();
$user->getEmail();
$user->getAvatar();
```

Laravel Cashier

介绍

Laravel Cashier 提供语义化，流畅的接口和 Stripe 的订购管理服务集成。它几乎处理了所有让人退步三舍的订购管理相关逻辑。除了基本的订购管理，Cashier 还可以处理折价券，订购转换，管理订购「数量」、服务有效期限，甚至产生收据的 PDF。

配置文件

Composer

首先，把 Cashier 扩展包加到 `composer.json`：

```
"laravel/cashier": "~3.0"
```

注册服务

然后，在 `app` 配置文件注册 `Laravel\Cashier\CashierServiceProvider`。

迁移

使用 Cashier 前，我们需要增加几个字段到数据库。别担心，你可以使用 `cashier:table` Artisan 命令，建立迁移文件来添加必要字段。例如，要增加字段到 users 数据表，使用 `php artisan cashier:table users`。建立完迁移文件后，只要执行 `migrate` 命令即可。

配置模型

然后，把 `Billable` trait 和相关的日期字段参数加到模型里：

```
use Laravel\Cashier\Billable;
use Laravel\Cashier\Contracts\Billable as BillableContract;
```

```
class User extends Eloquent implements BillableContract {

    use Billable;

    protected $dates = ['trial_ends_at', 'subscription_ends_at'];

}
```

Stripe Key

最后，在初始化文件或服务注册里（如 `AppServiceProvider` ）加入 Stripe key：

```
User::setStripeKey('stripe-key');
```

订购方案

当有了模型实例，你可以很简单的处理客户订购的 Stripe 里的方案：

```
$user = User::find(1);

$user->subscription('monthly')->create($creditCardToken);
```

如果你想在建立订购的时候使用折价券，可以使用 `withCoupon` 方法：

```
$user->subscription('monthly')
     ->withCoupon('code')
     ->create($creditCardToken);
```

`subscription` 方法会自动建立与 Stripe 的交易，以及将 Stripe customer ID 和其他相关帐款信息更新到数据库。如果你的方案有在 Stripe 配置试用期，试用到日期也会自动记录起来。

如果你的方案有试用期间，但是没有在 Stripe 里配置，你必须在处理订购后手动保存试用到日期。

```
$user->trial_ends_at = Carbon::now()->addDays(14);

$user->save();
```

自定义额外用户详细数据

如果你想自定义额外的顾客详细数据，你可以将数据数组作为 `create` 方法的第二个参数传入：

```
$user->subscription('monthly')->create($creditCardToken, [
    'email' => $email, 'description' => 'Our First Customer'
]);
```

想知道更多 Stripe 支持的额外字段，可以查看 Stripe 的在线文档 建立客户。

一次性付款

如果你想使用一次性付款而不是信用卡订购方案，你可以使用 `charge` 方法：

```
$user->charge(100);
```

`charge` 方法接受一个用最低单位货币的数量参数。 比如，上面的例子中会付款 100 美分 或者 1.00 美元，而不是用户的信用卡。

`charge` 方法也接受一个数组作为第二个参数，允许你传入一些创建 Stripe 订购的选项：

```
$user->charge(100, [
    'source' => $token,
    'receipt_email' => $user->email,
]);
```

如果付款失败， `charge` 方法会返回 `false` 。一般来说，这意味着付款被拒绝：

```
if ( ! $user->charge(100))
{
    // The charge was denied...
}
```

如果付款成功, 一个完整的 Stripe 响应会从这个方法返回。

免信用卡试用

如果你提供免信用卡试用服务，把 `cardUpFront` 属性设为 `false` ：

```
protected $cardUpFront = false;
```

建立帐号时，记得把试用到日期记录起来：

```
$user->trial_ends_at = Carbon::now()->addDays(14);

$user->save();
```

订购转换

使用 `swap` 方法可以把用户转换到新的订购：

```
$user->subscription('premium')->swap();
```

如果用户还在试用期间，试用服务会跟之前一样可用。如果订单有「数量」，也会和之前一样。

订购数量

有时候订购行为会跟「数量」有关。例如，你的应用程序可能会依照帐号的用户人数，每人每月收取 $10 元。你可以使用 `increment` 和 `decrement` 方法简单的调整订购数量：

```
$user = User::find(1);

$user->subscription()->increment();

// Add five to the subscription's current quantity...
$user->subscription()->increment(5);

$user->subscription()->decrement();

// Subtract five to the subscription's current quantity...
$user->subscription()->decrement(5);
```

取消订购

取消订购相当简单：

```
$user->subscription()->cancel();
```

当客户取消订购时，Cashier 会自动更新数据库的 `subscription_ends_at` 字段。这个字段会被用来判断 `subscribed` 方法是否该返回 `false`。例如，如果顾客在三月一号取消订购，但是服务可以使用到三月五号为止，那么 `subscribed` 方法在三月五号前都会传回 `true`。

恢复订购

如果你想要恢复客户之前取消的订购，使用 `resume` 方法：

```
$user->subscription('monthly')->resume($creditCardToken);
```

如果客户取消订购后，在服务过期前恢复，他们不用在当下付款。他们的服务会立刻重启，而付款则会遵循平常的流程。

确认订购状态

要确认用户是否订购了服务，使用 `subscribed` 方法：

```
if ($user->subscribed())
{
    //
}
```

`subscribed` 方法很适合用在 路由中间件:

```
public function handle($request, Closure $next)
{
    if ($request->user() && ! $request->user()->subscribed())
    {
        return redirect('billing');
    }

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

你可以使用 `onTrial` 方法，确认用户是否还在试用期间：

```
if ($user->onTrial())
{
    //
}
```

要确认用户是否曾经订购但是已经取消了服务，可以使用 `cancelled` 方法：

```
if ($user->cancelled())
{
    //
}
```

你可能想确认用户是否已经取消订单，但是服务还没有到期。例如，如果用户在三月五号取消了订购，但是服务会到三月十号才过期。那么用户到三月十号前都是有效期间。注意， `subscribed` 方法在过期前都会返回 `true` 。

```
if ($user->onGracePeriod())
{
    //
}
```

`everSubscribed` 方法可以用来确认用户是否订购过应用程序里的方案：

```
if ($user->everSubscribed())
{
    //
}
```

`onPlan` 方法可以用方案 ID 来确认用户是否订购某方案：

```
if ($user->onPlan('monthly'))
{
    //
}
```

处理交易失败

如果顾客的信用卡过期了呢？无需担心，Cashier 包含了 Webhook 控制器，可以帮你简单的取消顾客的订单。只要在路由注册控制器：

```
Route::post('stripe/webhook', 'Laravel\Cashier\WebhookController@handleWebhook');
```

这样就成了！失败的交易会经由控制器捕捉并进行处理。控制器会进行至多三次再交易尝试，都失败后才会取消顾客的订单。上面的 `stripe/webhook` URI 只是一个例子，你必须使用配置在 Stripe 里的 URI 才行。

处理其它 Stripe Webhooks

如果你想要处理额外的 Stripe webhook 事件，可以继承 Webhook 控制器。你的方法名称要对应到 Cashier 预期的名称，尤其是方法名称应该使用 `handle` 前缀，后面接着你想要处理的 Stripe webhook 。例如，如果你想要处理 `invoice.payment_succeeded` webhook ，你应该增加一个 `handleInvoicePaymentSucceeded` 方法到控制器。

```
class WebhookController extends Laravel\Cashier\WebhookController {

    public function handleInvoicePaymentSucceeded($payload)
    {
        // Handle The Event
    }

}
```

> 注意： 除了更新你数据库里的订购信息以外， Webhook 控制器也会经由 Stripe API 取消你的订购。

收据

你可以很简单的经由 `invoices` 方法拿到客户的收据数据数组：

```
$invoices = $user->invoices();
```

你可以使用这些辅助方法，列出收据的相关信息给客户看：

```
{{ $invoice->id }}

{{ $invoice->dateString() }}

{{ $invoice->dollars() }}
```

使用 `downloadInvoice` 方法产生收据的 PDF 下载。是的，它非常容易：

```
return $user->downloadInvoice($invoice->id, [
    'vendor'  => 'Your Company',
    'product' => 'Your Product',
]);
```

缓存

配置

Laravel 为各种不同的缓存系统提供一致的 API 。缓存配置文件位在 `config/cache.php` 。您可以在此为应用程序指定使用哪一种缓存系统， Laravel 支持各种常见的后端缓存系统，如 Memcached 和 Redis 。

缓存配置文件也包含多个其他选项，在文件里都有说明，所以请务必先阅读过。 Laravel 默认使用 `文件` 缓存系统，该系统会保存序列化、缓存对象在文件系统中。在大型应用程序上，建议使用保存在内存内的缓存系统，如 Memcached 或 APC 。你甚至可以以同一个缓存系统配置多个缓存配置。

在 Laravel 中使用 Redis 缓存系统前， 必须先使用 Composer 安装 `predis/predis` 扩展包 (~1.0) 。

缓存用法

保存对象到缓存中

```
Cache::put('key', 'value', $minutes);
```

使用 **Carbon** 对象配置缓存过期时间

```
$expiresAt = Carbon::now()->addMinutes(10);

Cache::put('key', 'value', $expiresAt);
```

若是对象不存在，则将其存入缓存中

```
Cache::add('key', 'value', $minutes);
```

当对象确实被加入缓存时，使用 `add` 方法将会返回 `true` 否则会返回 `false` 。

确认对象是否存在

```
if (Cache::has('key'))
{
    //
}
```

从缓存中取得对象

```
$value = Cache::get('key');
```

取得对象或是返回默认值

```
$value = Cache::get('key', 'default');

$value = Cache::get('key', function() { return 'default'; });
```

永久保存对象到缓存中

```
Cache::forever('key', 'value');
```

有时候您会希望从缓存中取得对象，而当此对象不存在时会保存一个默认值，您可以使用 `Cache::remember` 方法：

```
$value = Cache::remember('users', $minutes, function()
{
    return DB::table('users')->get();
});
```

您也可以结合 `remember` 和 `forever` 方法：

```
$value = Cache::rememberForever('users', function()
{
    return DB::table('users')->get();
});
```

请注意所有保存在缓存中的对象皆会被序列化，所以您可以任意保存各种类型的数据。

从缓存拉出对象

如果您需要从缓存中取得对象后将它删除，您可以使用 `pull` 方法：

```
$value = Cache::pull('key');
```

从缓存中删除对象

```
Cache::forget('key');
```

获取特定的缓存存储

当使用多种缓存存储时，你可以通过 `store` 方法来访问它们：

```
$value = Cache::store('foo')->get('key');
```

递增与递减

除了 `文件` 与 `数据库` 以外的缓存系统都支持 `递增` 和 `递减` 操作：

递增值

```
Cache::increment('key');

Cache::increment('key', $amount);
```

递减值

```
Cache::decrement('key');

Cache::decrement('key', $amount);
```

缓存标签

> 注意： `文件` 或 `数据库` 这类缓存系统均不支持缓存标签。此外，使用带有「forever」的缓存标签时，挑选 `memcached` 这类缓存系统将获得最好的性能，它会自动清除过期的纪录。

访问缓存标签

缓存标签允许您标记缓存内的相关对象，然后使用特定名称更新所有缓存标签。要访问缓存标签可以使用 `tags` 方法。

您可以保存缓存标签，通过将有序标签列表当作参数传入，或者作为标签名称的有序数组：

```
Cache::tags('people', 'authors')->put('John', $john, $minutes);

Cache::tags(array('people', 'artists'))->put('Anne', $anne, $minutes);
```

您可以结合使用各种缓存保存方法与标签，包含 `remember`，`forever`，和 `rememberForever` 。您也可以从已标记的缓存中访问对象，以及使用其他缓存方法如 `increment` 和 `decrement` 。

从已标记的缓存中访问对象

要访问已标记的缓存，可传入相同的有序标签列表。

```
$anne = Cache::tags('people', 'artists')->get('Anne');

$john = Cache::tags(array('people', 'authors'))->get('John');
```

您可以更新所有已标记的对象，使用指定名称或名称列表。例如，以下例子将会移除带有 `people` 或 `authors` 或者两者皆有的所有缓存标签，所以「Anne」和「John」皆会从缓存中被移除:

```
Cache::tags('people', 'authors')->flush();
```

对照来看，以下例子将只会移除带有 `authors` 的标签，所以「John」会被移除，但是「Anne」不会。

```
Cache::tags('authors')->flush();
```

数据库缓存

当使用 `数据库` 缓存系统时，您必须配置一张数据表来保存缓存对象。数据表的 `Schema` 声明例子如下：

```
Schema::create('cache', function($table)
{
    $table->string('key')->unique();
    $table->text('value');
    $table->integer('expiration');
});
```

集合

介绍

`Illuminate\Support\Collection` 类提供一个流畅、方便的封装来操作数组数据。举个例子，查看下面的代码。我们将会使用 `collect` 辅助方法来用数组建立一个新的集合实例：

```
$collection = collect(['taylor', 'abigail', null])->map(function($name)
{
    return strtoupper($name);
})
->reject(function($name)
{
    return empty($name);
});
```

可以看到，`Collection` 类允许你链式调用它的方法对背后的数组执行流畅的映射和归纳。一般说来，每一个 `Collection` 的方法都返回一个全新的 `Collection` 实例。为了更深一步的了解，请继续阅读！

基本用法

建立集合

如上述，`collect` 辅助方法将会用给定的数组返回一个新的 `Illuminate\Support\Collection` 实例。你也可以在 `Collection` 类上使用 `make` 命令：

```
$collection = collect([1, 2, 3]);

$collection = Collection::make([1, 2, 3]);
```

当然，Eloquent 的对象集合总是以 `Collection` 实例返回；然而，你可以在应用程序的任何地方方便的使用 `Collection` 类。

探索集合

作为列出集合可以用的所有方法 (有很多) 的替代，请查看 类的 API 文档！

Command Bus

简介

Command bus 提供一个简便的方法来封装任务，使你的程序更加容易阅读与执行，为了帮助我们更加了解使用「命令」的目的，让我们来模拟建立一个可以购买 podcast 的网站。

用户购买 podcasts 的过程中需要做很多事。例如，我们需要从用户的信用卡扣款，将纪录添加到数据库以表示购买，并发送购买确认的电子邮件，或许，我们还需要进行许多验证来确认用户是否可以购买。

我们可以将这些逻辑通通放在控制器的方法内，然而，这样做会有一些缺点，首先，控制器可能还需要处理许多其他的 HTTP 请求，包含复杂的逻辑，这会让控制器变得很臃肿且难易阅读，第二点，这些逻辑无法在这个控制器以外被重复使用，第三，这些命令无法被单元测试，为此我们还得额外产生一个 HTTP 请求，并向网站进行完整购买 podcast 的流程。

比起将逻辑放在控制器内，我们可以选择使用一个「命令」对象来封装它，如 `PurchasePodcast` 命令。

建立命令

使用 `make:command` 这个 Artisan 命令可以产生一个新的命令类 ：

```
php artisan make:command PurchasePodcast
```

新产生的类会被放在 `app/Commands` 目录中，命令默认包含了两个方法：构造器和 `handle` 。当然，`handle` 方法执行命令时，你可以使用构造器传入相关的对象到这个命令中。例如：

```
class PurchasePodcast extends Command implements SelfHandling {

    protected $user, $podcast;

    /**
     * Create a new command instance.
     *
     * @return void
     */
    public function __construct(User $user, Podcast $podcast)
    {
        $this->user = $user;
        $this->podcast = $podcast;
    }

    /**
     * Execute the command.
     *
```

```
     * @return void
     */
    public function handle()
    {
        // Handle the logic to purchase the podcast...

        event(new PodcastWasPurchased($this->user, $this->podcast));
    }

}
```

`handle` 方法也可以使用类型提示依赖，并且通过 服务容器 机制自动进行依赖注入。例如：

```
    /**
     * Execute the command.
     *
     * @return void
     */
    public function handle(BillingGateway $billing)
    {
        // Handle the logic to purchase the podcast...
    }
```

调用命令

所以，我们建立的命令该如何调用它呢？当然，我们可以直接调用 `handle` 方法，然而使用 Laravel 的 "command bus" 来调用命令将会有许多优点，待会我们会讨论这个部分。

如果你有浏览过内置的基本控制器，将会发现 `DispatchesCommands` trait ，它将允许我们在控制器内调用 `dispatch` 方法，例如：

```
public function purchasePodcast($podcastId)
{
    $this->dispatch(
        new PurchasePodcast(Auth::user(), Podcast::findOrFail($podcastId))
    );
}
```

Command bus 将会负责执行命令和调用 IoC 容器来将所需的依赖注入到 `handle` 方法。

你也可以将 `Illuminate\Foundation\Bus\DispatchesCommands` trait 加入任何要使用的类内。若你想要在任何类的构造器内接收 command bus 的实体 ，你可以使用类型提示 `Illuminate\Contracts\Bus\Dispatcher` 这个接口。 最后，你也可以使用 `Bus` facade 来快速派发命令：

```
    Bus::dispatch(
        new PurchasePodcast(Auth::user(), Podcast::findOrFail($podcastId))
    );
```

从请求映射要注入命令的属性

映射 HTTP 请求到命令是很常见的，所以，与其要你针对每个请求苦命地进行手动对应，Laravel 则提供一些有用的方法来轻松达到，让我们来看一下 `DispatchesCommands` trait 提供的 `dispatchFrom` 方法：

```
$this->dispatchFrom('Command\Class\Name', $request);
```

这个方法将会检查这个被传入的命令类的构造器，并取出来自于 HTTP 请求的变量(或其他任何的 `ArrayAccess` 对象) 并将其填入构造器，所以，若命令类在构造器接受 `firstName` 参数，command bus 将会试图从 HTTP 请求取出 `firstName` 参数。

`dispatchFrom` 方法的第三个参数允许你传入数组，那些不在 HTTP 请求内的参数可用这个数组来填入构造器：

```
$this->dispatchFrom('Command\Class\Name', $request, [
    'firstName' => 'Taylor',
]);
```

命令队列

Command bus 不仅仅作为当下请求的同步作业，也可以作为 Laravel 队列任务的主要方法，所以，我们要如何指示 command bus 在背景作业而不是同步处理呢？非常简单，首先，在建立新的命令时加上 `--queued` 参数：

```
php artisan make:command PurchasePodcast --queued
```

正如你所见的，这让命令增加了一点功能，即 `Illuminate\Contracts\Queue\ShouldBeQueued` 接口和 `SerializesModels` trait。 他们指示 command bus 使用队列来执行命令，以及优雅的序列化和反序列化任何在命令内被保存的 Eloquent 模型。

若你想将已存在的命令转换为队列命令，只需手动修改让命令类实现 `Illuminate\Contracts\Queue\ShouldBeQueued` 接口，它不包含方法，而是仅仅给调用员作为"标记接口"。

然后，一如往常撰写你的命令，当你将命令派发到 bus，它将会自动将命令丢到背景队列执行，没有比这个更容易的方法了。

想了解更多关于队列命令的方法，请见队列文档.

命令管道

在命令被派发到处理器之前，你也可以将它通过"命令管道"传递到其他类去。命令管道操作上如 HTTP 中间件，除了是专门来给命令用的，例如，一个命令管道能够在数据库事务处理期间包装全部的命令操作，或者仅作为执行纪录。

要将管道添加到 bus，只要从 `App\Providers\BusServiceProvider::boot` 方法调用调用员的 `pipeThrough` 方法：

```
$dispatcher->pipeThrough(['UseDatabaseTransactions', 'LogCommand']);
```

一个命令管道被定义在 `handle` 方法，就如个中间件：

```
class UseDatabaseTransactions {

    public function handle($command, $next)
    {
        return DB::transaction(function() use ($command, $next)
        {
            return $next($command);
        });
    }

}
```

命令管道是透过 IoC 容器来达成，所以请自行在构造器类型提示所需的依赖。

你甚至可以定义一个 `闭包` 来作为命令管道：

```
$dispatcher->pipeThrough([function($command, $next)
{
    return DB::transaction(function() use ($command, $next)
    {
        return $next($command);
    }
}]);
```

扩展框架

管理者和工厂

Laravel 有几个 `Manager` 类，用来管理创建基于驱动的组件。这些类包括缓存、session 、认证和队列组件。管理者类负责基于应用程序的配置建立一个特定的驱动实现。例如，`CacheManager` 类可以建立 APC 、 Memcached 、文件和各种其他的缓存驱动实现。

这些管理者都拥有 `extend` 方法，可以简单地用它来注入新的驱动解析功能到管理者。我们将会在下面的例子，随着讲解如何为它们注入自定义驱动支持，涵盖这些管理者的内容。

> 注意： 建议花点时间来探索 Laravel 附带的各种 `Manager` 类，例如： `CacheManager` 和 `SessionManager` 。看过这些类将会让你更彻底了解 Laravel 表面下是如何运作。所有的管理者类继承 `Illuminate\Support\Manager` 基础类，它提供一些有用、常见的功能给每一个管理者。

缓存

为了扩展 Laravel 缓存功能，我们将会使用 `CacheManager` 的 `extend` 方法，这方法可以用来绑定一个自定义驱动解析器到管理者，并且是全部的管理者类通用的。例如，注册一个新的缓存驱动名为「mongo」，我们将执行以下操作：

```
Cache::extend('mongo', function($app)
{
    return Cache::repository(new MongoStore);
});
```

传递到 `extend` 方法的第一个参数是驱动的名称。这将会对应到你的 `config/cache.php` 配置文件里的 `driver` 选项。第二个参数是个应该返回 `Illuminate\Cache\Repository` 实例的闭包。 `$app` 将会被传递到闭包，它是 `Illuminate\Foundation\Application` 和服务容器的实例。

`Cache::extend` 的调用可以在新的 Laravel 应用程序默认附带的 `App\Providers\AppServiceProvider` 的 `boot` 方法中完成，或者你可以建立自己的服务提供者来放置这个扩展 - 记得不要忘记在 `config/app.php` 的提供者数组注册提供者。

要建立自定义缓存驱动，首先需要实现 `Illuminate\Contracts\Cache\Store` contract 。所以，我们的 MongoDB 缓存实现将会看起来像这样：

```
class MongoStore implements Illuminate\Contracts\Cache\Store {

    public function get($key) {}
    public function put($key, $value, $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() {}

}
```

我们只需要使用 MongoDB 连接来实现这些方法。当实现完成，就可以完成自定义驱动注册：

```
Cache::extend('mongo', function($app)
{
    return Cache::repository(new MongoStore);
});
```

如果你正在考虑要把自定义缓存驱动代码放在哪里，请考虑把它放上 Packagist ！或者，你可以在 `app` 的目录中建立 `Extensions` 命名空间。记得 Laravel 没有严格的应用程序架构，你可以依照喜好自由的组织应用程序。

Session

自定义 session 驱动来扩展 Laravel 和扩展缓存系统一样简单。我们将会再一次使用 `extend` 方法来注册自定义代码：

```
Session::extend('mongo', function($app)
{
    // Return implementation of SessionHandlerInterface
});
```

在哪里扩展 Session

你应该把 session 扩展代码放置在 `AppServiceProvider` 的 `boot` 方法里。

实现 Session 扩展

要注意我们的自定义缓存驱动应该要实现 `SessionHandlerInterface` 。这个接口只包含少数需要实现的简单方法。一个基本的 MongoDB 实现会看起来像这样：

```
class MongoHandler implements SessionHandlerInterface {

    public function open($savePath, $sessionName) {}
    public function close() {}
    public function read($sessionId) {}
    public function write($sessionId, $data) {}
    public function destroy($sessionId) {}
    public function gc($lifetime) {}

}
```

因为这些方法不像缓存的 `StoreInterface` 一样容易理解，让我们快速地看过这些方法做些什么：

- `open` 方法通常会被用在基于文件的 session 保存系统。因为 Laravel 附带一个 `file` session 驱动，几乎不需要在这个方法放任何东西。你可以让它留空。PHP 要求我们去实现这个方法，事实上明显是个差劲的接口设计 (我们将会晚点讨论它)。
- `close` 方法，就像 `open` 方法，通常也可以忽略。对大部份的驱动来说，并不需要它。
- `read` 方法应该返回与给定 `$sessionId` 关联的 session 数据的字串形态。当你的驱动取回或保存 session 数据时不需要做任何序列化或进行其他编码，因为 Laravel 将会为你进行序列化
- `write` 方法应该写入给定 `$data` 字串与 `$sessionId` 的关联到一些永久存储系统，例如：MongoDB、 Dynamo、等等。
- `destroy` 方法应该从永久存储移除与 `$sessionId` 关联的数据。
- `gc` 方法应该销毁所有比给定 `$lifetime` UNIX 时间戳记还旧的 session 数据。对于会自己过期的系统如 Memcached 和 Redis，这个方法可以留空。

当 `SessionHandlerInterface` 实现完成，我们准备好要用 Session 管理者注册它：

```
Session::extend('mongo', function($app)
{
    return new MongoHandler;
});
```

当 session 驱动已经被注册，我们可以在 `config/session.php` 配置文件使用 `mongo` 驱动。

> 注意： 记住，如果你写了个自定义 session 处理器，请在 Packagist 分享它！

认证

认证可以用与缓存和 session 功能相同的方法扩展。再一次的，使用我们已经熟悉的 `extend` 方法：

```
Auth::extend('riak', function($app)
{
    // 返回 Illuminate\Contracts\Auth\UserProvider 的实现
});
```

`UserProvider` 实现只负责从永久存储系统抓取 `Illuminate\Contracts\Auth\Authenticatable` 实现，存储系统例如： MySQL 、 Riak ，等等。这两个接口让 Laravel 认证机制无论用户数据如何保存或用什么种类的类来代表它都能继续运作。

让我们来看一下 `UserProvider` contract ：

```
interface UserProvider {

    public function retrieveById($identifier);
    public function retrieveByToken($identifier, $token);
    public function updateRememberToken(Authenticatable $user, $token);
    public function retrieveByCredentials(array $credentials);
    public function validateCredentials(Authenticatable $user, array $credentials);

}
```

`retrieveById` 函数通常接收一个代表用户的数字键，例如：MySQL 数据库的自动递增 ID。这方法应该

取得符合 ID 的 `Authenticatable` 实现并返回。

`retrieveByToken` 函数用用户唯一的 `$identifier` 和保存在 `remember_token` 字段的「记住我」 `$token` 来取得用户。跟前面的方法一样，应该返回 `Authenticatable` 的实现。

`updateRememberToken` 方法用新的 `$token` 更新 `$user` 的 `remember_token` 字段。新 token 可以是在「记住我」成功地登录时，传入一个新的 token，或当用户注销时传入一个 null。

`retrieveByCredentials` 方法接收当尝试登录应用程序时，传递到 `Auth::attempt` 方法的凭证数组。这个方法应该接着「查找」底层使用的永久存储，找到符合凭证的用户。这个方法通常会对 `$credentials['username']` 用「 where 」条件查找。 并且应该返回一个 `UserInterface` 接口的实现。这个方法不应该尝试做任何密码验证或认证。

`validateCredentials` 方法应该通过比较给定的 `$user` 与 `$credentials` 来验证用户。举例来说，这个方法可以比较 `$user->getAuthPassword()` 字串跟 `Hash::make` 后的 `$credentials['password']`。这个方法应该只验证用户的凭证数组并且返回布尔值。

现在我们已经看过 `UserProvider` 的每个方法，接着来看一下 `Authenticatable`。记住，提供者应该从 `retrieveById` 和 `retrieveByCredentials` 方法返回这个接口的实现：

```
interface Authenticatable {

    public function getAuthIdentifier();
    public function getAuthPassword();
    public function getRememberToken();
    public function setRememberToken($value);
    public function getRememberTokenName();

}
```

这个接口很简单。 The `getAuthIdentifier` 方法应该返回用户的「主键」。在 MySQL 后台，同样，这将会是个自动递增的主键。 `getAuthPassword` 应该返回用户哈希过的密码。这个接口让认证系统可以与任何用户类一起运作，无论你使用什么 ORM 或保存抽象层。默认，Laravel 包含一个实现这个接口的 `User` 类在 `app` 文件夹里，所以你可以参考这个类当作实现的例子。

最后，当我们已经实现了 `UserProvider`，我们准备好用 `Auth` facade 来注册扩展：

```
Auth::extend('riak', function($app)
{
    return new RiakUserProvider($app['riak.connection']);
});
```

用 `extend` 方法注册驱动之后，在你的 `config/auth.php` 配置文件切换到新驱动。

基于服务容器的扩展

几乎每个 Laravel 框架引入的服务提供者都会绑定对象到服务容器中。你可以在 `config/app.php` 配置文件中找到应用程序的服务提供者清单。如果你有时间，你应该浏览过这里面每一个提供者的源代码。通过这样做，你将会更了解每一个提供者添加什么到框架，以及用什么键值来绑定各种服务到服务容器。

例如， `HashServiceProvider` 绑定 `hash` 做为键值到服务容器，它将解析成 `Illuminate\Hashing\BcryptHasher` 实例。你可以在应用程序中覆写这个 IoC 绑定，轻松地扩展并覆写这个类。例如：

```
<?php namespace App\Providers;

class SnappyHashProvider extends \Illuminate\Hashing\HashServiceProvider {

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

        $this->app->bindShared('hash', function()
        {
            return new \Snappy\Hashing\ScryptHasher;
        });
    }

}
```

要注意的是这个类扩展 `HashServiceProvider`，不是默认的 `ServiceProvider` 基础类。当你扩展了服务提供者，在 `config/app.php` 配置文件把 `HashServiceProvider` 换成你扩展的提供者名称。

这是被绑定在容器的所有核心类的一般扩展方法。实际上，每个以这种方式绑定在容器的核心类都可以被覆写。再次强调，看过每个框架引入的服务提供者将会使你熟悉：每个类被绑在容器的哪里、它们是用什么键值绑定。这是个好方法可以了解更多关于 Laravel 如何结合它们。

Laravel Elixir

简介

Laravel Elixir 提供了简洁流畅的 API，让你能够为你的 Laravel 应用程序定义基本的 Gulp 任务。Elixir 支持许多常见的 CSS 与 JavaScrtip 预处理器，甚至包含了测试工具。

如果你曾经对于使用 Gulp 及编译资源感到困惑，那么你绝对会爱上 Laravel Elixir！

安装与配置

安装 Node

在开始使用 Elixir 之前，你必须先确定你的开发环境上有安装 Node.js。

```
node -v
```

默认情况下，Laravel Homestead 会包含你所需的一切；当然，如果你没有使用 Vagrant，那么你可以浏览 Node 的下载页进行安装。别担心，安装是很简单又快速的！

Gulp

接着你需要全局安装 Gulp 的 NPM 安装包，如这样：

```
npm install --global gulp
```

Laravel Elixir

最后的步骤就是安装 Elixir！伴随着新安装的 Laravel，你会发现根目录有个名为 `package.json` 的文件。想像它就如同你的 `composer.json` 文件，只是它定义的是 Node 的依赖，而不是 PHP。你可以使用以下的命令进行安装依赖的动作：

```
npm install
```

使用方式

现在你已经安装好 Elixir，未来任何时候你都能进行编译及合并文件！ 在项目根目录下的 `gulpfile.js` 已

经包含了所有的 Elixir 任务。

编译 Less

```
elixir(function(mix) {
    mix.less("app.less");
});
```

在上述例子中，Elixir 会假设你的 Less 文件保存在 `resources/assets/less` 里。

编译多个 Less 文件

```
elixir(function(mix) {
    mix.less([
        'app.less',
        'something-else.less'
    ]);
});
```

编译 Sass

```
elixir(function(mix) {
    mix.sass("app.scss");
});
```

在上述例子中，Elixir 会假设你的 Sass 文件保存在 `resources/assets/sass` 里。

编译 CoffeeScript

```
elixir(function(mix) {
    mix.coffee();
});
```

在上述例子中，Elixir 会假设你的 CoffeeScript 文件保存在 `resources/assets/coffee` 里。

编译所有的 Less 及 CoffeeScript

```
elixir(function(mix) {
    mix.less()
       .coffee();
});
```

触发 PHPUnit 测试

```
elixir(function(mix) {
    mix.phpUnit();
});
```

触发 PHPSpec 测试

```
elixir(function(mix) {
    mix.phpSpec();
});
```

合并样式文件

```
elixir(function(mix) {
    mix.styles([
        "normalize.css",
        "main.css"
    ]);
});
```

传递给此方法的文件路径均相对于 `resources/css` 目录。

合并样式文件且保存在自定义的路径

```
elixir(function(mix) {
    mix.styles([
        "normalize.css",
        "main.css"
    ], 'public/build/css/everything.css');
});
```

从特定相对目录合并样式文件

```
elixir(function(mix) {
    mix.styles([
        "normalize.css",
        "main.css"
    ], 'public/build/css/everything.css', 'public/css');
});
```

`styles` 与 `scrtips` 方法可以通过传入第三个参数来决定来源文件的相对目录。

合并指定目录里所有的样式文件

```
elixir(function(mix) {
    mix.stylesIn("public/css");
});
```

合并脚本文件

```
elixir(function(mix) {
    mix.scripts([
        "jquery.js",
        "app.js"
    ]);
```

```
});
```

同样的，传递给此方法的文件路径均相对于 `resources/js` 目录

合并指定目录里所有的脚本文件

```
elixir(function(mix) {
    mix.scriptsIn("public/js/some/directory");
});
```

合并多组脚本文件

```
elixir(function(mix) {
    mix.scripts(['jquery.js', 'main.js'], 'public/js/main.js')
       .scripts(['forum.js', 'threads.js'], 'public/js/forum.js');
});
```

压缩文件并加上哈希的版本号

```
elixir(function(mix) {
    mix.version("css/all.css");
});
```

这个动作会为你的文件名称加上独特的哈希值，以防止文件被缓存。举例来说，产生出来的文件名称可能像这样：`all-16d570a7.css`。

接着在你的视图中，你能够使用 `elixir()` 函数来正确加载名称被哈希后的文件。举例如下：

```
<link rel="stylesheet" href="{{ elixir("css/all.css") }}">
```

程序的作用下，`elixir()` 函数会将参数内的源文件名转换成被哈希后的文件名并加载。是否有如释重担的感觉呢？

您也可以传给一个数组给 `version` 方法来为多个文件进行版本管理：

```
elixir(function(mix) {
    mix.version(["css/all.css", "js/app.js"]);
});
```

```
<link rel="stylesheet" href="{{ elixir("css/all.css") }}">
<script src="{{ elixir("js/app.js") }}"></script>
```

复制文件到新的位置

```
elixir(function(mix) {
    mix.copy('vendor/foo/bar.css', 'public/css/bar.css');
```

```
});
```

将整个目录都复制到新的位置

```
elixir(function(mix) {
    mix.copy('vendor/package/views', 'resources/views');
});
```

方法连接

当然，你能够串连 Elixir 大部份的方法来建立一连串的任务：

```
elixir(function(mix) {
    mix.less("app.less")
       .coffee()
       .phpUnit()
       .version("css/bootstrap.css");
});
```

Gulp

现在你已经告诉 Elixir 要执行的任务，接着只需要在命令行执行 Gulp。

执行一次所有注册的任务

```
gulp
```

监控文件变更

```
gulp watch
```

仅编译 **javascript**

```
gulp scripts
```

仅编译 **css** 样式

```
gulp styles
```

监控测试以及 **PHP** 类的变更

```
gulp tdd
```

> 提示： 所有的任务都会使用开发环境进行，所以压缩功能不会被执行。如果要使用上线环境，可以使

> 用 `gulp --production`。

功能扩充

你甚至能够建立自己的 Gulp 任务至 Elixir 里。想像一下，你想加入一个有趣的任务，使用终端机后会打打印一些消息。看起来可能会如下：

```
var gulp = require("gulp");
var shell = require("gulp-shell");
var elixir = require("laravel-elixir");

elixir.extend("message", function(message) {

    gulp.task("say", function() {
        gulp.src("").pipe(shell("say " + message));
    });

    return this.queueTask("say");

});
```

请注意我们 `扩增( extend )` Elixir 的 API 时所使用的第一个参数，稍后我们需要在 Gulpfile 中使用它，以及建立 Gulp 任务所使用的回调函数。

如果你想要让你的自定义任务能被监控，只要在监控器注册就行了。

```
this.registerWatcher("message", "**/*.php");
```

这行程序的意思是指，当符合正则表达式的文件一经修改，就会触发 `message` 任务。

很好！接着你可以将这行程序写在 Gulpfile 的顶端，或者将它放到自定义任务的文件里。如果你选择后者，那么你必须将它加载至你的 Gulpfile，例如：

```
require("./custom-tasks")
```

大功告成！最后你只需要将他们结合。

```
elixir(function(mix) {
    mix.message("Tea, Earl Grey, Hot");
});
```

加入之后，每当你触发 Gulp，Picard 就会要求一些茶。

加密

介绍

Laravel 通过 Mcrypt PHP 扩充扩展包提供功能强大的 AES 加密功能。

基本用法

加密

```
$encrypted = Crypt::encrypt('secret');
```

> 注意： 请确保 `config/app.php` 文件中的 `key` 选项配置了 16, 24, 或 32 字符的随机字串，否则加密的数值不会安全。

解密

```
$decrypted = Crypt::decrypt($encryptedValue);
```

配置密码与模式

您也可以使用加密器来配置密码和模式：

```
Crypt::setMode('ctr');

Crypt::setCipher($cipher);
```

Envoy 任务执行器

简介

Laravel Envoy 提供了简洁、轻量的语法用于定义在远程服务器上可执行的通用任务。通过 Blade 风格的语法，你可以很容易地设置任务从而完成部署、执行 Artisan 命令或其他更多工作。

> 注意： Envoy 依赖 PHP 5.4 或更高版本，并且只能运行在 Mac / Linux 操作系统中。

安装

首先，通过 Composer 的 `global` 命令来安装 Envoy：

```
composer global require "laravel/envoy=~1.0"
```

请务必将 `~/.composer/vendor/bin` 目录加入到 PATH 环境变量中，这样才能在命令行中执行 `envoy` 命令时找到可执行文件。

接下来，在项目的根目录下创建 `Envoy.blade.php` 文件。下面给出的实例代码你可以当做模板使用：

```
@servers(['web' => '192.168.1.1'])

@task('foo', ['on' => 'web'])
    ls -la
@endtask
```

如上所示，在文件的开头首先定义了 `@servers` 数组。后续的任务声明中，你可以在 `on` 选项中直接引用。在 `@task` 声明里，你可以直接填写需要在服务器上执行的 Bash 脚本代码。

`init` 命令可以很方便地用来创建一个包含基本内容的 Envoy 文件：

```
envoy init user@192.168.1.1
```

执行任务

使用 `run` 命令来执行任务：

```
envoy run foo
```

如有需要，你还可以通过命令行向 Envoy 文件传递参数：

```
envoy run deploy --branch=master
```

你可以通过 Blade 语法引用这些参数：

```
@servers(['web' => '192.168.1.1'])

@task('deploy', ['on' => 'web'])
    cd site
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

Bootstrapping

你可以在 `@setup` 指令中声明变量，并在 Envoy 文件中执行普通的 PHP 代码:

```
@setup
    $now = new DateTime();

    $environment = isset($env) ? $env : "testing";
@endsetup
```

还可以通过 `@include` 指令引入任意的 PHP 文件：

```
@include('vendor/autoload.php');
```

多服务器

在多台服务器上执行一个任务是非常简单的，只需在声明任务时列出服务器名称即可：

```
@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
```

默认情况下，任务将以串行的方式依次在每台服务器上执行。也就是说，任务在第一台服务器上执行完成后才会切换到下一台服务器上执行。

并行执行

如果你希望在多个服务器上并行执行一个任务，只需在任务声明处添加 `parallel` 选项即可：

```
@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
```

任务宏

“宏”可以让你只用一条命令就能顺序执行一组任务。例如：

```
@servers(['web' => '192.168.1.1'])

@macro('deploy')
    foo
    bar
@endmacro

@task('foo')
    echo "HELLO"
@endtask

@task('bar')
    echo "WORLD"
@endtask
```

`deploy` 宏可以通过一条简单地命令启动并执行：

```
envoy run deploy
```

通知

HipChat

任务执行完后，你可能希望发送一条通知信息到团队的 HipChat 聊天室，这一功能可以通过 `@hipchat` 指令实现：

```
@servers(['web' => '192.168.1.1'])

@task('foo', ['on' => 'web'])
    ls -la
@endtask

@after
    @hipchat('token', 'room', 'Envoy')
@endafter
```

你还可以定制发送到 hipchat 聊天室的消息内容。任何在 `@setup` 或通过 `@include` 引入的变量都可以在

消息中直接引用：

```
@after
    @hipchat('token', 'room', 'Envoy', "$task ran on [$environment]")
@endafter
```

Envoy 让你的团队时刻掌握服务器上任务执行的情况变得惊人的简单。

Slack

下面的代码实例可以将通知发送到 Slack 聊天室：

```
@after
    @slack('team', 'token', 'channel')
@endafter
```

更新 Envoy

通过 Composer 来更新 Envoy：

```
composer global update
```

错误与日志

配置

应用程序的日志功能配置在 `Illuminate\Foundation\Bootstrap\ConfigureLogging` 启动类中。这个类使用 `config/app.php` 配置文件的 `log` 配置选项。

日志工具默认使用每天的日志文件；然而，你可以依照需求自定义这个行为。因为 Laravel 使用流行的 Monolog 日志函数库，你可以利用很多 Monolog 提供的处理进程。

例如，如果你想要使用单一日志文件，而不是每天一个日志文件，你可以对 `config/app.php` 配置文件做下面的变更：

```
'log' => 'single'
```

Laravel 提供立即可用的 `single` 、 `daily` 和 `syslog` 日志模式。然而，你可以通过覆写 `ConfigureLogging` 启动类，依照需求自由地自定义应用程序的日志。

错误细节

`config/app.php` 配置文件的 `app.debug` 配置选项控制应用程序透过浏览器显示错误细节。配置选项默认参照 `.env` 文件的 `APP_DEBUG` 环境变量。

进行本地开发时，你应该配置 `APP_DEBUG` 环境变量为 `true` 。 在上线环境，这个值应该永远为 **`false`** 。

错误处理

所有的异常都由 `App\Exceptions\Handler` 类处理。这个类包含两个方法： `report` 和 `render` 。

`report` 方法用来记录异常或把异常传递到外部服务，例如： BugSnag 。默认情况下， `report` 方法只基本实现简单地传递异常到父类并于父类记录异常。然而，你可以依你所需自由地记录异常。如果你需要使用不同的方法来报告不同类型的异常，你可以使用 PHP 的 `instanceof` 比较运算符：

```
/**
 * 报告或记录异常。
 *
 * 这是一个发送异常到 Sentry、Bugsnag 等服务的好地方。
 *
 * @param  \Exception  $e
 * @return void
 */
public function report(Exception $e)
```

```
{
    if ($e instanceof CustomException)
    {
        //
    }

    return parent::report($e);
}
```

`render` 方法负责把异常转换成应该被传递回浏览器的 HTTP 响应。默认情况下，异常会被传递到基础类并帮你产生响应。然而，你可以自由的检查异常类型或返回自定义的响应。

异常处理进程的 `dontReport` 属性是个数组，包含应该不要被纪录的异常类型。由 404 错误导致的异常默认不会被写到日志文件。你可以依照需求添加其他类型的异常到这个数组。

HTTP 异常

有一些异常是描述来自服务器的 HTTP 错误码。例如，这可能是个「找不到页面」错误 (404)、「未授权错误」(401)，或甚至是工程师导致的 500 错误。使用下面的方法来返回这样一个响应：

```
abort(404);
```

或是你可以选择提供一个响应：

```
abort(403, 'Unauthorized action.');
```

你可以在请求的生命周期中任何时间点使用这个方法。

自定义 404 错误页面

要让所有的 404 错误返回自定义的视图，请建立一个 `resources/views/errors/404.blade.php` 文件。应用程序将会使用这个视图处理所有发生的 404 错误。

日志

Laravel 日志工具在强大的 Monolog 函数库上提供一层简单的功能。Laravel 默认为应用程序建立每天的日志文件在 `storage/logs` 目录。你可以像这样把信息写到日志：

```
Log::info('This is some useful information.');

Log::warning('Something could be going wrong.');

Log::error('Something is really going wrong.');
```

日志工具提供定义在 RFC 5424 的七个级别：**debug**、**info**、**notice**、**warning**、**error**、**critical** 和 **alert**。

也可以传入上下文相关的数据数组到日志方法里：

```
Log::info('Log message', ['context' => 'Other helpful information']);
```

Monolog 有很多其他的处理方法可以用在日志上。如有需要，你可以取用 Laravel 底层使用的 Monolog 实例：

```
$monolog = Log::getMonolog();
```

你也可以注册事件来捕捉所有传到日志的消息：

注册日志事件监听器

```
Log::listen(function($level, $message, $context)
{
    //
});
```

事件

基本用法

Laravel 的 event 功能提供一个简单的观察者实现，允许你在应用程序里订阅与监听事件。事件类通常被保存在 `app/Events` 目录下，而它们的处理程序则被保存在 `app/Handlers/Events` 目录下。

你可以使用 Artisan 命令行工具产生一个新的事件类：

```
php artisan make:event PodcastWasPurchased
```

订阅事件

Laravel 里的 `EventServiceProvider` 提供了一个方便的地方注册所有的事件处理程序。`listen` 属性包含一个所有的事件 (键) 和相对应的处理程序 (值) 的 数组。当然，你可以依应用程序的需求添加任何数量的事件到这个数组。举个例子，让我们来加上 `PodcastWasPurchased` 事件：

```
/**
 * 应用程序的事件处理程序对照。
 *
 * @var array
 */
protected $listen = [
    'App\Events\PodcastWasPurchased' => [
        'App\Handlers\Events\EmailPurchaseConfirmation',
    ],
];
```

使用 Artisan 命令行命令 `handler:event`，来产生一个事件的处理程序：

```
php artisan handler:event EmailPurchaseConfirmation --event=PodcastWasPurchased
```

当然，在每次你需要一个处理程序或是事件时，手动地执行 `make:event` 和 `handler:event` 命令很麻烦。作为替代，简单地添加处理程序跟事件到你的 `EventServiceProvider` 并使用 `event:generate` 命令。这个命令将会产生任何在你的 `EventServiceProvider` 列出的事件跟处理程序：

```
php artisan event:generate
```

触发事件

现在我们准备好使用 `Event` facade 触发我们的事件：

```
$response = Event::fire(new PodcastWasPurchased($podcast));
```

`fire` 方法返回一个响应的数组，让你可以用来控制你的应用程序接下来要有什么反应。

你也可以使用 `event` 辅助方法来触发事件：

```
event(new PodcastWasPurchased($podcast));
```

监听器闭包

你甚至可以不需对事件建立对应的处理类。举个例子，在你的 `EventServiceProvider` 的 `boot` 方法里，你可以做下面这件事：

```
Event::listen('App\Events\PodcastWasPurchased', function($event)
{
    // 处理事件...
});
```

停止继续传递事件

有时候你会希望停止继续传递事件到其他监听器。你可以通过从处理程序返回 `false` 来做到这件事：

```
Event::listen('App\Events\PodcastWasPurchased', function($event)
{
    // 处理事件...

    return false;
});
```

事件处理队列

需要把事件处理程序放到 队列 吗？这不能变得再更简单了。当你产生处理程序，简单地使用 `--queued` 旗标：

```
php artisan handler:event SendPurchaseConfirmation --event=PodcastWasPurchased --queued
```

这将会产生一个实现了 `Illuminate\Contracts\Queue\ShouldBeQueued` 接口的处理程序类。这样就可以了！现在当这个处理程序因为事件发生被调用，它将会被事件配送器自动地排进队列。

当处理程序被队列执行，如果没有异常被丢出，在执行后该队列中的任务将会自动被删除。你也可以手动取用队列中的任务的 `delete` 和 `release` 方法。队列处理程序默认会引入的 `Illuminate\Queue\InteractsWithQueue` trait，让你可以取用这些方法：

```
public function handle(PodcastWasPurchased $event)
{
    if (true)
```

```
    {
        $this->release(30);
    }
}
```

如果你想要把一个已存在的处理程序转换成一个队列的处理程序，简单地手动添加 `ShouldBeQueued` 接口到类。

事件订阅者

定义事件订阅者

事件订阅者是个可以从类自身里面订阅多个事件的类。订阅者应该定义 `subscribe` 方法，事件配送器实体将会被传递到这个方法：

```
class UserEventHandler {

    /**
     * 处理用户登录事件。
     */
    public function onUserLogin($event)
    {
        //
    }

    /**
     * 处理用户注销事件。
     */
    public function onUserLogout($event)
    {
        //
    }

    /**
     * 注册监听器给订阅者。
     *
     * @param  Illuminate\Events\Dispatcher  $events
     * @return array
     */
    public function subscribe($events)
    {
        $events->listen('App\Events\UserLoggedIn', 'UserEventHandler@onUserLogin');

        $events->listen('App\Events\UserLoggedOut', 'UserEventHandler@onUserLogout');
    }

}
```

注册事件订阅者

当定义了订阅者后，可以使用 `Event` 类注册。

```
$subscriber = new UserEventHandler;

Event::subscribe($subscriber);
```

你也可以使用 服务容器 自动解析订阅者。简单地传递订阅者的名字给 `subscribe` 方法就可以做到：

```
Event::subscribe('UserEventHandler');
```

文件系统 / 云存储

简介

Laravel 有很棒的文件系统抽象层，是基于 Frank de Jonge 的 Flysystem 扩展包。 Laravel 集成的 Flysystem 提供了简单的接口，可以操作本地端空间、 Amazon S3 、 Rackspace Cloud Storage 。更好的是，它可以非常简单的切换不同保存方式，但仍使用相同的 API 操作！

配置文件

文件系统的配置文件放在 `config/filesystems.php` 。在这个文件内你可以配置所有的「硬盘」。每个硬盘代表一种保存方式和地点。默认的配置文件内已经包含了所有保存方式的例子。所以只要修改保存配置和认证即可！

在使用 S3 或 Rackspace 之前，你必须先用 Composer 安装相对应的扩展包：

- Amazon S3: `league/flysystem-aws-s3-v2 ~1.0`
- Rackspace: `league/flysystem-rackspace ~1.0`

当然，你可以加入任意数量的硬盘配置文件，甚至配置多个硬盘都使用同一种保存方式。

使用本地端空间时，要注意所有的操作路径都是相对于配置文件里 `local` 的 `root` ，默认的路径是 `storage/app` 。因此下列的操作将会保存一个文件在 `storage/app/file.txt` ：

```
Storage::disk('local')->put('file.txt', 'Contents');
```

基本用法

可以用 `Storage` facade 操作所有写在配置文件里的硬盘。或者是，你也可以将 `Illuminate\Contracts\Filesystem\Factory` 类型提示写到任何类里，经由 Laravel 的 服务容器 解析。

取得一个特定硬盘

```
$disk = Storage::disk('s3');

$disk = Storage::disk('local');
```

确认文件是否存在

```
$exists = Storage::disk('s3')->exists('file.jpg');
```

使用默认硬盘调用方法

```
if (Storage::exists('file.jpg'))
{
    //
}
```

取得文件内容

```
$contents = Storage::get('file.jpg');
```

配置文件内容

```
Storage::put('file.jpg', $contents);
```

附加内容到文件结尾

```
Storage::prepend('file.log', 'Prepended Text');
```

加入内容到文件开头

```
Storage::append('file.log', 'Appended Text');
```

删除文件

```
Storage::delete('file.jpg');

Storage::delete(['file1.jpg', 'file2.jpg']);
```

复制文件到新的路径

```
Storage::copy('old/file1.jpg', 'new/file1.jpg');
```

移动文件到新的路径

```
Storage::move('old/file1.jpg', 'new/file1.jpg');
```

取得文件大小

```
$size = Storage::size('file1.jpg');
```

取得最近修改时间 **(UNIX)**

```
$time = Storage::lastModified('file1.jpg');
```

取得目录下所有文件

```
$files = Storage::files($directory);

// Recursive...
$files = Storage::allFiles($directory);
```

取得目录下所有子目录

```
$directories = Storage::directories($directory);

// Recursive...
$directories = Storage::allDirectories($directory);
```

建立目录

```
Storage::makeDirectory($directory);
```

删除目录

```
Storage::deleteDirectory($directory);
```

自定义文件系统

Laravel 的文件系统默认已经集成了不少驱动。不过，文件系统并不仅限于这些，还有针对其他存储系统的一些适配器。如果你想使用这些适配器，你可以创建一个自定义的驱动。不用担心，它没有那么复杂！

如果要创建一个自定义的文件系统，你需要创建一个服务提供者，比如 `DropboxFilesystemServiceProvider`。在提供者的 `boot` 方法中，你可以注入一个实现了 `Illuminate\Contracts\Filesystem\Factory` 接口的实例并且调用注入实例的 `extend` 方法。 或者你也可以使用 `Disk` facade 的 `extend` 方法。

`extend` 方法的第一个参数是驱动的名字，第二个参数是一个闭包，接受 `$app` 和 `$config` 变量。这个闭包的返回值必须是 `League\Flysystem\Filesystem` 的一个实例。

> 注意： $config 变量已经包含了定义在 `config/filesystems.php` 中特定硬盘的配置。

Dropbox 示例

```
<?php namespace App\Providers;

use Storage;
use League\Flysystem\Filesystem;
use Dropbox\Client as DropboxClient;
use League\Flysystem\Dropbox\DropboxAdapter;
```

```
class DropboxFilesystemServiceProvider {

    public function boot()
    {
        Storage::extend('dropbox', function($app, $config)
        {
            $client = new DropboxClient($config['accessToken'], $config['clientIdentifier

            return new Filesystem(new DropboxAdapter($client));
        });
    }

}
```

哈希

简述

在 Laravel `Hash` 内保存的密码使用 Bcrypt 加密方式。 如果您在 Laravel 使用认证控制器，控制器也会帮助未使用 Bcrypt 加密的密码进行 Bcrypt 验证。 同样，在用户注册服务内 Laravel 也提供 `bcrypt` 密码加密的方式保存密码。

基本用法

对 **A** 密码使用**Bcrypt** 加密

```
$password = Hash::make('secret');
```

你也可直接使用 bcrypt 的 function

```
$password = bcrypt('secret');
```

对加密的 **A** 密码进行验证

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

检查 **A** 密码是否需要重新加密

```
if (Hash::needsRehash($hashed))
{
    $hashed = Hash::make('secret');
}
```

辅助方法

数组

array_add

如果给定的键不在数组中，`array_add` 函数会把给定的键值对加到数组中。

```
$array = array('foo' => 'bar');

$array = array_add($array, 'key', 'value');
```

array_divide

`array_divide` 函数返回两个数组，一个包含原本数组的键，另一个包含原本数组的值。

```
$array = array('foo' => 'bar');

list($keys, $values) = array_divide($array);
```

array_dot

`array_dot` 函数把多维数组扁平化成一维数组，并用「点」符号表示深度。

```
$array = array('foo' => array('bar' => 'baz'));

$array = array_dot($array);

// array('foo.bar' => 'baz');
```

array_except

`array_except` 函数从数组移除给定的键值对。

```
$array = array_except($array, array('keys', 'to', 'remove'));
```

array_fetch

`array_fetch` 函数返回包含被选择的嵌套元素的扁平化数组。

```
$array = array(
    array('developer' => array('name' => 'Taylor')),
    array('developer' => array('name' => 'Dayle')),
);

$array = array_fetch($array, 'developer.name');

// array('Taylor', 'Dayle');
```

array_first

`array_first` 函数返回数组中第一个通过给定的测试为真的元素。

```
$array = array(100, 200, 300);

$value = array_first($array, function($key, $value)
{
    return $value >= 150;
});
```

也可以传递默认值当作第三个参数：

```
$value = array_first($array, $callback, $default);
```

array_last

`array_last` 函数返回数组中最后一个通过给定的测试为真的元素。

```
$array = array(350, 400, 500, 300, 200, 100);

$value = array_last($array, function($key, $value)
{
    return $value > 350;
});

// 500
```

也可以传递默认值当作第三个参数：

```
$value = array_last($array, $callback, $default);
```

array_flatten

`array_flatten` 函数将会把多维数组扁平化成一维。

```
$array = array('name' => 'Joe', 'languages' => array('PHP', 'Ruby'));
```

```
$array = array_flatten($array);

// array('Joe', 'PHP', 'Ruby');
```

array_forget

`array_forget` 函数将会用「点」符号从深度嵌套数组移除给定的键值对。

```
$array = array('names' => array('joe' => array('programmer')));

array_forget($array, 'names.joe');
```

array_get

`array_get` 函数将会使用「点」符号从深度嵌套数组取回给定的值。

```
$array = array('names' => array('joe' => array('programmer')));

$value = array_get($array, 'names.joe');

$value = array_get($array, 'names.john', 'default');
```

> 注意: 想要把 `array_get` 用在对象上？ 请使用 `object_get`。

array_only

`array_only` 函数将会只从数组返回给定的键值对。

```
$array = array('name' => 'Joe', 'age' => 27, 'votes' => 1);

$array = array_only($array, array('name', 'votes'));
```

array_pluck

`array_pluck` 函数将会从数组拉出给定键值对的清单。

```
$array = array(array('name' => 'Taylor'), array('name' => 'Dayle'));

$array = array_pluck($array, 'name');

// array('Taylor', 'Dayle');
```

array_pull

`array_pull` 函数将会从数组返回给定的键值对，并移除它。

```
$array = array('name' => 'Taylor', 'age' => 27);
```

```
$name = array_pull($array, 'name');
```

array_set

`array_set` 函数将会使用「点」符号在深度嵌套数组中指定值。

```
$array = array('names' => array('programmer' => 'Joe'));

array_set($array, 'names.editor', 'Taylor');
```

array_sort

`array_sort` 函数通过给定闭包的结果来排序数组。

```
$array = array(
    array('name' => 'Jill'),
    array('name' => 'Barry'),
);

$array = array_values(array_sort($array, function($value)
{
    return $value['name'];
}));
```

array_where

使用给定的闭包过滤数组。

```
$array = array(100, '200', 300, '400', 500);

$array = array_where($array, function($key, $value)
{
    return is_string($value);
});

// Array ( [1] => 200 [3] => 400 )
```

head

返回数组中第一个元素。对 PHP 5.3.x 的方法链很有用。

```
$first = head($this->returnsArray('foo'));
```

last

返回数组中最后一个元素。对方法链很有用。

```
$last = last($this->returnsArray('foo'));
```

路径

app_path

取得 `app` 文件夹的完整路径。

```
$path = app_path();
```

base_path

取得应用程序安装根目录的完整路径。

public_path

取得 `public` 文件夹的完整路径。

storage_path

取得 `app/storage` 文件夹的完整路径。

路由

get

注册一个 GET 路由。

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

post

注册一个 POST 路由。

```
post('foo/bar', 'FooController@action');
```

put

注册一个 PUT 路由。

```
put('foo/bar', 'FooController@action');
```

patch

注册一个 PATCH 路由。

```
patch('foo/bar', 'FooController@action');
```

delete

注册一个 DELETE 路由。

```
delete('foo/bar', 'FooController@action');
```

resource

注册一个 RESTful 的资源路由。

```
resource('foo', 'FooController');
```

字串

camel_case

把给定的字串转换成 `驼峰式命名`。

```
$camel = camel_case('foo_bar');

// fooBar
```

class_basename

取得给定类的类名称，不含任何命名空间的名称。

```
$class = class_basename('Foo\Bar\Baz');

// Baz
```

e

对给定字串执行 `htmlentities`，并支持 UTF-8。

```
$entities = e('<html>foo</html>');
```

ends_with

判断句子结尾是否有给定的字串。

```
$value = ends_with('This is my name', 'name');
```

snake_case

把给定的字串转换成 `蛇形命名` 。

```
$snake = snake_case('fooBar');

// foo_bar
```

str_limit

限制字串的字符数量。

```
str_limit($value, $limit = 100, $end = '...')
```

例子：

```
$value = str_limit('The PHP framework for web artisans.', 7);

// The PHP...
```

starts_with

判断句子是否开头有给定的字串。

```
$value = starts_with('This is my name', 'This');
```

str_contains

判断句子是否有给定的字串。

```
$value = str_contains('This is my name', 'my');
```

str_finish

加一个给定字串到句子结尾。多余一个的给定字串则移除。

```
$string = str_finish('this/string', '/');

// this/string/
```

str_is

判断字串是否符合给定的模式。星号可以用来当作通配符。

```
$value = str_is('foo*', 'foobar');
```

str_plural

把字串转换成它的多数形态 (只有英文)。

```
$plural = str_plural('car');
```

str_random

产生给定长度的随机字串。

```
$string = str_random(40);
```

str_singular

把字串转换成它的单数形态 (只有英文)。

```
$singular = str_singular('cars');
```

str_slug

从给定字串产生一个对网址友善的「slug」。

```
str_slug($title, $separator);
```

例子：

```
$title = str_slug("Laravel 5 Framework", "-");

// laravel-5-framework
```

studly_case

把给定字串转换成 `首字大写命名` 。

```
$value = studly_case('foo_bar');

// FooBar
```

trans

翻译给定的语句。等同 `Lang::get`。

```
$value = trans('validation.required'):
```

trans_choice

随着词形变化翻译给定的语句。等同 `Lang::choice`。

```
$value = trans_choice('foo.bar', $count);
```

网址

action

产生给定控制器行为的网址。

```
$url = action('HomeController@getIndex', $params);
```

route

产生给定路由名称的网址。

```
$url = route('routeName', $params);
```

asset

产生资源的网址。

```
$url = asset('img/photo.jpg');
```

secure_asset

产生给定资源的 HTTPS HTML 链接。

```
echo secure_asset('foo/bar.zip', $title, $attributes = array());
```

secure_url

产生给定路径的 HTTPS 完整网址。

```
echo secure_url('foo/bar', $parameters = array());
```

url

产生给定路径的完整网址。

```
echo url('foo/bar', $parameters = array(), $secure = null);
```

其他

csrf_token

返回 取得现在 CSRF token 的值。

```
$token = csrf_token();
```

dd

打印给定变量并结束脚本执行。

```
dd($value);
```

elixir

获取带版本号的 Elixir 文件的路径

```
elixir($file);
```

env

获取一个环境变量的值，如果没有则返回一个默认值。

```
env('APP_ENV', 'production')
```

event

触发一个事件。

```
event('my.event');
```

value

如果给定的值是个 `闭包` ，返回 `闭包` 的返回值。不是的话，则返回值。

```
$value = value(function() { return 'bar'; });
```

view

用给定的视图路径取得一个视图实例。

```
return view('auth.login');
```

with

返回给定对象。对 PHP 5.3.x 的构造器方法链很有用。

```
$value = with(new Foo)->doWork();
```

本地化

介绍

Laravel 的 `Lang` facade 提供方便的方法来取得多种语言的字串，让你简单地在应用进程里支持多种语言。

语言文件

语言字串保存在 `resources/lang` 文件夹的文档里。在这个文件夹里应该要给每一个应用进程支持的语言一个子文件夹。

```
/resources
    /lang
        /en
            messages.php
        /es
            messages.php
```

语言文件例子

语言文件简单地返回键跟字串的数组。例如：

```
<?php

return array(
    'welcome' => 'Welcome to our application'
);
```

在执行时变换默认语言

应用进程的默认语言被保存在 `config/app.php` 配置文件。你可以在任何时候用 `App::setLocale` 方法变换现行语言：

```
App::setLocale('es');
```

配置备用语言

你也可以配置「备用语言」，它将会在当现行语言没有给定的语句时被使用。就像默认语言，备用语言也可以在 `config/app.php` 配置文件配置：

```
'fallback_locale' => 'en',
```

基本用法

从语言文件取得句子

```
echo Lang::get('messages.welcome');
```

传递给 `get` 方法的字串的第一个部分是语言文件的名称，第二个部分是应该被取得的句子的名称。

> 注意：如果语句不存在，`get` 方法将会返回键的名称。

你也可以使用 `trans` 辅助方法，它是 `Lang::get` 方法的别名。

```
echo trans('messages.welcome');
```

在句子中做替代

你也可以在语句中定义占位符：

```
'welcome' => 'Welcome, :name',
```

接着，传递替代用的第二个参数给 `Lang::get` 方法：

```
echo Lang::get('messages.welcome', array('name' => 'Dayle'));
```

判断语言文件是否有指定的句子

```
if (Lang::has('messages.welcome'))
{
    //
}
```

复数

复数是个复杂的问题，不同语言对于复数有很多种复杂的规则。你可以简单地在你的语言文件里管理它。你可以用「管道」字符区分字串的单数和复数形态：

```
'apples' => 'There is one apple|There are many apples',
```

接着你可以用 `Lang::choice` 方法取得语句：

```
echo Lang::choice('messages.apples', 10);
```

你也可以提供一个地区参数来指定语言。举个例，如果你想要使用俄语 (ru)：

```
echo Lang::choice('товар|товара|товаров', $count, array(), 'ru');
```

因为 Laravel 的翻译器由 Symfony 翻译组件提供，你也可以很容易地建立更明确的复数规则：

```
'apples' => '{0} There are none|[1,19] There are some|[20,Inf] There are many',
```

验证

要验证本地化的错误和消息，可以看一下验证的文档.

覆写扩展包的语言文件

许多扩展包附带它们自有的语句。你可以通过放置文档在 `resources/lang/packages/{locale}/{package}` 文件夹来覆写它们，而不是改变扩展包的核心文档来调整这些句子。所以，举个例子，如果你需要覆写 `skyrim/hearthfire` 扩展包在 `messages.php` 的英文语句，你可以放置语言文件在： `resources/lang/packages/en/hearthfire/messages.php` 。你可以只定义你想要覆写的语句在这个文档里，任何你没有覆写的语句将会仍从扩展包的语言文件加载。

邮件

配置

Laravel 基于热门的 SwiftMailer 函数库之上，提供了一个简洁的 API。邮件配置文件为 `config/mail.php` ，包含若干选项，让您可以更改 SMTP 主机、连接端口、凭证，也可以让您对函数库发送出去的所有消息配置全局的 `from` 地址。您可使用任何您想要的 SMTP 服务器。如果想使用 PHP `mail` 函数来发送邮件，您可以将配置文件中的 `driver` 更改为 `mail` 。您也可以使用 `sendmail` 驱动器。

API 驱动

Laravel 也包含了 Mailgun 及 Mandrill HTTP API 的驱动。这些 API 通常比 SMTP 服务器更简单快速。这两套驱动都需要在应用程序中安装 Guzzle 4 HTTP 函数库。您可在 `composer.josn` 中加入下列代码， 以便在应用中加入 Guzzle 4：

```
"guzzlehttp/guzzle": "~4.0"
```

Mailgun 驱动

要使用 Mailgun 驱动，请将 `config/mail.php` 配置文件中的 `driver` 选项配置为 `mailgun` 。接下来，若 `config/service.php` 配置文件还不存在于您的应用中，请建立此文件，并确认其包含下列选项：

```
'mailgun' => array(
    'domain' => 'your-mailgun-domain',
    'secret' => 'your-mailgun-key',
),
```

Mandrill 驱动

要使用 Mandrill 驱动，将 `config/mail.php` 配置文件中的 `driver` 选项配置为 `mandrill` 。接下来，若 `config/service.php` 配置文件还不存在于您的应用中，请建立此文件，并确认其包含下列选项：

```
'mandrill' => array(
    'secret' => 'your-mandrill-key',
),
```

日志驱动

若您的 `config/mail.php` 配置文件中的 `driver` 选项配置为 `log` ，所有的电子邮件都会被写入日志文件，而不会真正寄给任何收件者。这主要用于快速的本地端除错及内容验证。

基本用法

您可使用 `Mail::send` 方法来发送电子邮件消息：

```
Mail::send('emails.welcome', array('key' => 'value'), function($message)
{
    $message->to('foo@example.com', 'John Smith')->subject('Welcome!');
});
```

传入 `send` 方法的第一个参数为邮件视图的名称。第二个是传递给该视图的数据，通常是一个关联式数组，让视图可通过 `$key` 来取得数据对象。第三个参数是一个闭包，可以对 message 进行各种配置。

> 注意： `$message` 变量总是会被传入邮件视图中，并且允许内嵌附件。因此最好避免在您的视图本体中传入 `message` 变量。

除了 HTML 视图外，您也可以指定使用纯文本视图：

```
Mail::send(array('html.view', 'text.view'), $data, $callback);
```

或者，您可使用 `html` 或 `text` 作为键值来指定单一类型的视图：

```
Mail::send(array('text' => 'view'), $data, $callback);
```

您也可以在邮件消息中指定其他选项，例如副本收件者或附件：

```
Mail::send('emails.welcome', $data, function($message)
{
    $message->from('us@example.com', 'Laravel');

    $message->to('foo@example.com')->cc('bar@example.com');

    $message->attach($pathToFile);
});
```

要附加文件至 message 时，可以指定 MIME 的类型、显示名称：

```
$message->attach($pathToFile, array('as' => $display, 'mime' => $mime));
```

若您只需发送一个简单的字串而非完整的视图，可使用 `raw` 方法：

```
Mail::raw('Text to e-mail', function($message)
{
    $message->from('us@example.com', 'Laravel');

    $message->to('foo@example.com')->cc('bar@example.com');
```

```
});
```

> 注意： 传递至 `Mail::send` 闭包的 message 实例是继承了 SwiftMailer 的 message 类，你可以调用该类的任何方法来建立电子邮件消息。

内嵌附件

在电子邮件中嵌入内部图像通常很麻烦；然而 Laravel 提供一个便利的方法让您对电子邮件附加图像，并取得相应的 CID。

在邮件视图中嵌入图像

```
<body>
    这是一张图像：

    <img src="<?php echo $message->embed($pathToFile); ?>">
</body>
```

在邮件视图中嵌入原始数据

```
<body>
    这是一张从原始数据来的图像：

    <img src="<?php echo $message->embedData($data, $name); ?>">
</body>
```

请注意 `Mail` 类总是会将 `$message` 变量传递给电子邮件视图。

邮件队列

将邮件消息加入队列

发送电子邮件消息会大幅延长应用程序的响应时间，因此许多开发者选择将邮件消息加入队列并于后台发送。 Laravel 使用内置 统一的 queue API ，让您轻松地完成此工作。要将邮件消息加入队列，只要使用 `Mail` 类的 `queue` 方法：

```
Mail::queue('emails.welcome', $data, function($message)
{
    $message->to('foo@example.com', 'John Smith')->subject('Welcome!');
});
```

您也可以使用 `later` 方法来指定您希望延迟发送邮件消息的秒数：

```
Mail::later(5, 'emails.welcome', $data, function($message)
{
    $message->to('foo@example.com', 'John Smith')->subject('Welcome!');
});
```

若您想要指定特定的队列或「管道」来加入消息，您可使用 `queueOn` 以及 `laterOn` 方法：

```
Mail::queueOn('queue-name', 'emails.welcome', $data, function($message)
{
    $message->to('foo@example.com', 'John Smith')->subject('Welcome!');
});
```

邮件与本地端开发

当开发发送电子邮件的应用程序时，我们通常希望不要真的从本地端或开发环境发送邮件。您可以使用 `Mail::pretend` 方法或将 `config/mail.php` 配置文件中的 `pretend` 选项配置为 `true`。在 `pretend` 模式下，消息会改而写入应用程序的日志文件，而不会真的发送给收件者。

若您想要实际阅览测试的邮件，可考虑使用如 MailTrap 的服务。

扩展包开发

介绍

开发扩展包是添加功能到 Laravel 最主要的方法。扩展包可以是任何处理日期的方式。例如，Carbon，或是一个全套的 BDD testing 框架。例如，Behat

当然，有非常多不同类型的扩展包。有些扩展包是独立的，意思是此扩展包运作且兼容于任何的框架，不只有 Laravel。Carbon 以及 Behat 都是这类的扩展包。任何这类的扩展包只需要在您的 `composer.json` 文件里配置就可以使用。

另一方面，其他的扩展包所设计的目的是只要在 Laravel 上使用。这些扩展包可能包含路由、控制器、视图以及扩展包的相关配置，目的是为了增加 Laravel 的应用。接下来的说明主要涵盖了 Laravel 开发这些扩展包的重点。

所有 Laravel 扩展包都发布到 Packagist 以及 Composer，所以学习这些美好的 PHP 扩展包管理工具是必须的。

视图

您扩展包内部的架构全部由您自己规划。然而，原则上会有一个或更多的 服务提供者. 服务提供者包含着所有的 服务容器 绑定，也定义了所有您扩展包的相关配置、视图以及语言文件在什么地方。

视图

扩展包的视图基本上使用两个双冒号来指定:

```
return view('package::view.name');
```

所有您所要做的只有告诉 Laravel 您所配置扩展包名称视图的位置在哪里。如果您的扩展包取名为 “courier” 您可能需要添加如下到您的服务提供者的 `boot` 方法:

```
public function boot()
{
    $this->loadViewsFrom(__DIR__.'/path/to/views', 'courier');
}
```

现在您可以使用如下的语法来加载扩展包的视图:

```
return view('courier::view.name');
```

当您使用 `loadViewsFrom` 方法，Laravel 实际上为了您的视图注册了两个位置。一个是您应用程序的 `resources/views/vendor` 目录，一个是您指定的目录。所以使用我们的例子 `courier` 当要求一个扩展包的视图时，Laravel 会第一时间检查是否有一个开发者自行自定义在 `resources/views/vendor/courier` 的视图存在。然而如果还没有这个路径的视图被自定义。Laravel 会搜索您在扩展包 `loadViewsFrom` 方法里所指定的视图。这个方法让个别的用户可以方便的自定义且覆写您在扩展包里的视图。

视图的发布

发布扩展包的视图到 `resources/views/vendor` 目录，您必须在服务提供者里的 `boot` 方法里使用 `publishes` 方法:

```
public function boot()
{
    $this->loadViewsFrom(__DIR__.'/path/to/views', 'courier');

    $this->publishes([
        __DIR__.'/path/to/views' => base_path('resources/views/vendor/courier'),
    ]);
}
```

现在当您扩展包的用户使用 Laravel 的命令 `vendor:publish` 您的视图目录将会被复制到所特定的目录

如果您想要覆写已存在的文件，可以使用 `--force`:

```
php artisan vendor:publish --force
```

> 注意**:** 您可以使用 `publishes` 方法，发布任何您的文件到任何您想要的地方。

语言

扩展包的语言文件基本上使用两个双冒号来指定:

```
return trans('package::file.line');
```

所有您所要做的只有告诉 Laravel 您所配置扩展包名称的语言位置在哪里。如果您的扩展包取名为 "courier" 您可能需要添加如下的语法到您的服务提供者的 `boot` 方法:

```
public function boot()
{
    $this->loadTranslationsFrom(__DIR__.'/path/to/translations', 'courier');
}
```

注意在您的 `translations` 目录里，必须要有更下一层的目录，例如 `en` `es` `ru`。

现在您可以使用下方的语法来加载您扩展包的语言:

```
return trans('courier::file.line');
```

配置文件

基本上，您可能想要将您扩展包相关配置的文件发布到应用程序本身的配置目录 `config`。这将允许您扩展包的用户简单的覆写这些默认的配置文件。

发布扩展包的配置文件只需要在服务提供者里的 `boot` 方法里使用 `publishes` 方法:

```
$this->publishes([
    __DIR__.'/path/to/config/courier.php' => config_path('courier.php'),
]);
```

现在当扩展包的用户执行 `vendor:publish` 命令，您的文件将会被复制到特定的位置。当然只要配置文件已经被发布，就可以如其他配置文件一样被访问:

```
$value = config('courier.option');
```

您可能也选择想要合并您扩展包的配置文件和应用程序里的副本配置文件。这允许您的用户在已经被发布的副本配置文件里只包含任何他们想要覆写的配置选项。如果想要合并配置文件，可在服务提供者里的 `register` 方法里使用 `mergeConfigFrom` 方法

```
$this->mergeConfigFrom(
    __DIR__.'/path/to/config/courier.php', 'courier'
);
```

公共资源

您的扩展包也许会有一些资源文件比如 JavaScript，CSS，和图片。如果要发布资源，可以在您的服务提供商的 `boot` 方法中使用 `publishes` 方法。在这个例子中，我们同样增加了 “public” 资源的分区标记。

```
$this->publishes([
    __DIR__.'/path/to/assets' => public_path('vendor/courier'),
], 'public');
```

现在当扩展包的用户执行 `vendor:publish` 命令，您的文件将会被复制到特定的位置。由于每次扩展更新时都会覆盖这些资源，你可以使用 `--force` 标识：

```
php artisan vendor:publish --tag=public --force
```

如果你想确保你的公共资源始终是最新的，你可以把这条命名添加到 `composer.json` 文件中的 `post-update-cmd` 列表中。

发布分类文件

您可能想要分别的发布一些分类的文件。举例，您可能想要您的用户可以分别发布扩展包的配置文件与静态资源文件。您可以使用 `tagging` 来达成:

```
// Publish a config file
$this->publishes([
    __DIR__.'/../config/package.php' => config_path('package.php')
], 'config');

// Publish your migrations
$this->publishes([
    __DIR__.'/../database/migrations/' => base_path('/database/migrations')
], 'migrations');
```

您可以使用这些 `tag` ，来分别发布这些扩展包里的文件。

```
php artisan vendor:publish --provider="Vendor\Providers\PackageServiceProvider" --tag="co
```

路由

在扩展包里加载一个路由文件，只需要在服务提供者里的 `boot` 方法里使用 `include` :

根据服务提供者来包含一个路由文件

```
public function boot()
{
    include __DIR__.'/../../routes.php';
}
```

> 注意: 如果您的扩展包里使用了控制器，您必须要确认您在 `composer.json` 文件里的 auto-load 区块里，是否适当的配置这些控制器。

分页

配置

在其他的框架中，实现分页是令人感到苦恼的事，但是 Laravel 令它实现起来变得轻松。 Laravel 可以产生基于当前页面的智能「范围」链接，所产生的 HTML 兼容 Bootstrap CSS 框架.

使用

有几种方法来分页对象。最简单的是在搜索构建器使用 `paginate` 方法或 Eloquent 模型。

对数据库结果分页

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

> 注意： 目前 Laravel 使用 `groupBy` 来做分页操作无法有效率的执行，如果您需要使用 `groupBy` 来分页数据集，建议您手动查找数据库，并使用 `Paginator::make` 。

对 **Eloquent** 模型分页

您也可以对 Eloquent 模型分页:

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

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

发送给 `paginate` 方法的参数是您希望每页要显示的对象选项数目，只要您取得查找结果后，您可以在视图中显示，并使用 `render` 方法去建立分页链接：

```
<div class="container">
    <?php foreach ($users as $user): ?>
        <?php echo $user->name; ?>
    <?php endforeach; ?>
</div>

<?php echo $users->render(); ?>
```

这就是所有建立分页系统的步骤了！您会注意到我们还没有告知 Laravel 我们目前的页面是哪一页，这个信息 Laravel 会自动帮您做好。

您也可以透过以下方法获得额外的分页信息：

- `currentPage`
- `lastPage`
- `perPage`
- `hasMorePages`
- `url`
- `nextPageUrl`
- `total`
- `count`

「简单分页」

如果您只是要在您的分页视图显示「上一页」和「下一页」链接，您有个选项 `simplePaginate` 方法来执行更高效率的搜索。当您不需要精准的显示页码在视图上时，这个方法在较大的数据集非常有用：

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

手动建立分页

有的时候您可能会想要从数组中对象手动建立分页实体， 您可以根据您的需要通过 `Illuminate\Pagination\Paginator` 或 `Illuminate\Pagination\LengthAwarePaginator` 实体来建立。

自定义分页 URL

您还可以透过 `setPath` 方法自定义使用的 URL：

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

$users->setPath('custom/url');
```

上面的例子将建立 URL，类似以下内容： http://example.com/custom/url?page=2

追加分页链接

您可以使用 `appends` 方法增加搜索字串到分页链接中：

```
<?php echo $users->appends(['sort' => 'votes'])->render(); ?>
```

这样会产生类似下列的链接：

```
http://example.com/something?page=2&sort=votes
```

如果您想要将「哈希片段」加到分页的 URL，您可以使用 `fragment` 方法：

```
<?php echo $users->fragment('foo')->render(); ?>
```

此方法调用后将产生 URL，看起来像这样：

```
http://example.com/something?page=2#foo
```

转换至 JSON

`Paginator` 类实现 `Illuminate\Contracts\Support\JsonableInterface` 接口的 `toJson` 方法。由路由返回的值，您可能将 `Paginator` 实体传换成 JSON。JSON 表单的实体会包含一些「元」信息，例如 `total`、`current_page`、`last_page`。该实体数据将可通过在 JSON 数组中 `data` 的键取得。

队列

设置

Laravel 队列组件提供一个统一的 API 集成了许多不同的队列服务，队列允许你延后执行一个耗时的任务，例如延后至指定的时间才发送邮件，进而大幅的加快了应用程序处理请求的速度。

队列的配置文件在 `config/queue.php`，在这个文件你将可以找到框架中每种不同的队列服务的连接设置，其中包含了 Beanstalkd、IronMQ、Amazon SQS、Redis、`null`，以及同步 (本地端使用) 驱动设置。驱动 `null` 只是简单的舍弃队列工作，因此那些工作永远不会执行。

队列数据表

为了能够使用 `database` 驱动，你需要建立一个数据表来保存工作。要使用一个迁移建立这个数据表，可以执行 `queue:table` Artisan 命令：

```
php artisan queue:table
```

其他队列依赖

下面的依赖是使用对应的队列驱动所需的扩展包：

- Amazon SQS: `aws/aws-sdk-php`
- Beanstalkd: `pda/pheanstalk ~3.0`
- IronMQ: `iron-io/iron_mq`
- Redis: `predis/predis ~1.0`

基本用法

推送一个工作至队列

应用程序中能够放进队列的工作都存放在 `App\Commands` 目录下，你可以借由下面 Artisan 命令产生一个可使用队列的命令：

```
php artisan make:command SendEmail --queued
```

要推送一个新的工作至队列，请使用 `Queue::push` 方法：

```
Queue::push(new SendEmail($message));
```

> 注意: 在这个例子当中，我们直接使用 `Queue` Facade，然而，常见的作法是借由 Command Bus 去分派队列命令。我们将会在这篇文章中继续使用 `Queue` Facade，不过，也要熟悉使用 command bus，因为它能够同时分派你的网站应用程序中队列与同步的命令。

默认情况下，`make:command` Artisan 命令会产生一个 "self-handling" 的命令，意味着命令里会包含一个 `handle` 方法。这个方法将会在队列执行时被调用。你可以在 `handle` 方法使用时提示传入任何你需要的依赖，而 服务容器 会自动注入他们：

```
public function handle(UserRepository $users)
{
    //
}
```

如果你希望你的命令有独立的处理类别，你可以在使用 `make:command` 命令时加上 `--handler` 标识。

```
php artisan make:command SendEmail --queued --handler
```

这个被产生出来的处理类别将会放在 `App\Handlers\Commands` 目录下面，并且服务容器会自动解析。

指定队列使用特定连接

你也可指定队列工作送至指定的连接：

```
Queue::pushOn('emails', new SendEmail($message));
```

发送相同的数据去多个队列工作

如果你需要发送一样的数据去几个不同的队列工作，你可以使用 `Queue::bulk` 方法：

```
Queue::bulk(array(new SendEmail($message), new AnotherCommand));
```

延迟执行一个工作

有时候你可能想要延迟执行一个队列工作，举例来说你希望一个队列工作在客户注册 15 分钟后才寄送 e-mail，你可以使用 `Queue::later` 方法来完成这件事情：

```
$date = Carbon::now()->addMinutes(15);

Queue::later($date, new SendEmail($message));
```

在这个例子中，我们使用 Carbon 日期类库来指定我们希望队列工作希望延迟的时间，另外你也可发送一个整数来设置你希望延迟的秒数。

> 注意: 在 Amazon SQS 服务中，有最大 900 秒（ 15 分钟 ）的限制。

将 **Eloquent** 模型放进队列

如果你队列工作的构造器接收一个 Eloquent 模型，只有这个模型的标记（ identifier ）会被序列化后放到队列中。当工作真正开始被处理的时候，队列系统会自动从数据库中重新取得完整的模型实例。这个对你的网站应用程序来说是完全透明的，并且预防一些在序列化完整 Eloquent 模型实例时可能遇到的问题。

删除一个处理中的工作

一旦一个工作被处理过后，这个工作必须从队列中删除。假如在工作执行后没有发生错误，这个将会自动完成。

如果你希望能够手动删除或着释放工作，在 `Illuminate\Queue\InteractsWithQueue` trait 中提供 `release` 以及 `delete` 方法的接口。其中 `release` 方法接受单一一个值：你想要等待工作再次能够执行的秒数。

```
public function handle(SendEmail $command)
{
    if (true)
    {
        $this->release(30);
    }
}
```

释放一个工作回到队列中

假如在工作执行后发生错误，这个工作将会自动被释放回到队列之中，如此一来便能够再次尝试执行工作。工作会一直被释放回队列直到到达应用程序的尝试上限。这个上限数值可以在使用 `queue:listen` 或 `queue:work` Artisan 命令时候借由 `--tries` 开关来设置。

检查工作执行次数

当一个工作执行后发生错误，这个工作将会自动的释放回队列当中，你可以透过 `attempts` 方法来检查这个工作已经被执行的次数：

```
if ($this->attempts() > 3)
{
    //
}
```

> 注意: 你的命令处理类别必须使用 `Illuminate\Queue\InteractsWithQueue` 这个 trait 才能够使用这个方法。

队列闭包

你也可以推送一个闭包去队列，这个方法非常的方便及快速的来处理需要使用队列的简单的任务：

推送一个闭包至队列

```
Queue::push(function($job) use ($id)
{
```

```
    Account::delete($id);

    $job->delete();
});
```

> 注意: 要让一个组件变量可以在队列闭包中可以使用我们会通过 `use` 命令，试着发送主键及重复使用的相关模块在你的队列工作中，这可以避免其他的序列化行为。

当使用 Iron.io push queues 时,你应该在队列闭包中采取一些其他的预防措施，我们应该在执行工作收到队列数据时检查token是否真来自 Iron.io，举例来说你推送一个队列工作到 `https://yourapp.com/queue/receive?token=SecretToken`，接下来在你的工作收到队列的请求时，你就可以检查token的值是否正确。

执行一个队列监听

Laravel 内含一个 Artisan 命令，它将推送到队列的工作拉来下执行，你可以使用 `queue:listen` 命令，来执行这件常驻任务：

开始队列监听

```
php artisan queue:listen
```

你也可以指定特定队列连接让监听器使用：

```
php artisan queue:listen connection
```

注意当这个任务开始时，这将会一直持续执行到他被手动停止，你也可以使用一个处理监控如 Supervisor 来确保这个队列监听不会停止执行。

你也可以在 `listen` 命令中使用逗号分隔的队列连接，来设置不同队列连接的优先层级：

```
php artisan queue:listen --queue=high,low
```

在这个范列中，总是会优先处理 `high-connection` 中的工作，然后才处理 `low-connection`。

指定工作超时参数

你也可以设置给每个工作允许执行的秒数：

```
php artisan queue:listen --timeout=60
```

指定队列休息时间

此外，你也可以指定让监听器在拉取新工作时要等待几秒：

```
php artisan queue:listen --sleep=5
```

注意队列只会在工作时休息，假如有许多可执行的工作，队列会持续的处理工作而不会休息

处理队列上的第一个工作

当你只想处理队列上的一个工作你可以使用 `queue:work` Artisan 命令：

```
php artisan queue:work
```

常驻队列处理器

在 `queue:work` 中也包含了一个 `--daemon` 选项，强迫队列处理器持续处理工作，而不会每次都重新启动框架，这个作法比起 `queue:listen` 可有效减少 CPU 使用量，但是却增加了布署时，对于处理中队列任务的复杂性。

要启动一个常驻的队列处理器，使用 `--daemon`：

```
php artisan queue:work connection --daemon

php artisan queue:work connection --daemon --sleep=3

php artisan queue:work connection --daemon --sleep=3 --tries=3
```

如你所见 `queue:work` 命令支持 `queue:listen` 大多相同的选项参数，你也可使用 `php artisan help queue:work` 命令来观看全部可用的选项参数。

布署常驻队列处理器

最简单布署一个应用程序使用常驻队列处理器的方式，就是将应用程序在开始布署时转成维护模式，你可以使用 `php artisan down` 命令来完成这件事情，当这个应用程序在维护模式，Laravel 将不会允许任何来自队列上的新工作，但会持续的处理已存在的工作。

要重新启动 `queue` 也是非常容易，请将底下命令加到部署命令：

```
php artisan queue:restart
```

上述命令会在执行完目前的工作后，重新启动队列。

> 注意: 这个命令依赖缓存系统来排定重新启动任务。默认 APCu 无法在命令提示字符中工作。如果你正在使用 APCu 请将 `apc.enable_cli=1` 加到你的 APCu 设置当中。

撰写常驻队列处理器

常驻队列处理器不会在处理每一个工作之前都重新启动框架。因此，你应该注意并小心地在工作处理完成之前释放占用的资源。例如，如果你正在使用 GD 函式库操作图片，当你完成工作的时候，你应该使用 `imagedestroy` 方法来释放占用的内存。

同样地，数据库连接可能在长时间执行的队列处理器中断线，你可以使用 `DB::reconnect` 方法来确保你每次都有一个全新的连接。

推送队列

你可以利用强大的 Laravel 5 队列架构来进行推送队列工作，不需要执行任何的常驻或背景监听，目前只支持 Iron.io 驱动，在你开始前建立一个 Iron.io 帐号及添加你的 Iron 凭证到 `config/queue.php` 配置文件。

注册一个推送队列订阅

接下来，你可以使用 `queue:subscribe` Artisan 命令注册一个 URL，这将会接收新的推送队列工作：

```
php artisan queue:subscribe queue_name http://foo.com/queue/receive
```

现在当你登录你的 Iron 管理后台，你将会看到你新的推送队列，以及订阅的 URL，你可以订阅许多的 URLs 给你希望的队列，接下来建立一个 route 给你的 `queue/receive` 及从 `Queue::marshal` 方法回传回应：

```
Route::post('queue/receive', function()
{
    return Queue::marshal();
});
```

这里的 `marshal` 方法会将触发正确的处理类别，而发送工作到队列中只要使用一样的 `Queue::push` 方法。

已失败的工作

事情往往不会如你预期的一样，有时候你推送工作到队列会失败，别担心，Laravel 包含一个简单的方法去指定一个工作最多可以被执行几次，在工作被执行到一定的次数时，他将会添加至 `failed_jobs` 数据表里，然后失败工作的数据表名称可以在 `config/queue.php` 里进行设置：

要产生一个迁移来建立 `failed_jobs` 数据表，你可以使用 `queue:failed-table` Artisan 命令：

```
php artisan queue:failed-table
```

你可以指定一个最大值来限制一个工作应该最多被执行几次，在你执行 `queue:listen` 时加上 `--tries`：

```
php artisan queue:listen connection-name --tries=3
```

假如你会想注册一个事件，这个事件会将会在队列失败时被调用，你可以使用 `Queue::failing` 方法，这个事件是一个很好的机会让你可以通知你的团队通过 e-mail 或 HipChat。

```
Queue::failing(function($connection, $job, $data)
{
    //
});
```

你可能够直接在队列工作类别中定义一个 `failed` 方法，这让你能够在工作失败时候，执行一些特定的动作：

```
public function failed()
{
    // 当工作失败的时候会被调用……
}
```

重新尝试失败的工作

要看到所有失败的工作，你可以使用 `queue:failed` 命令：

```
php artisan queue:failed
```

这个 `queue:failed` 命令将会列出工作 ID、连接、队列名称及失败的时间，可以使用工作 ID 重新执行一个失败的工作，例如一个已经失败的工作的 ID 是 5，我们可以使用下面的命令：

```
php artisan queue:retry 5
```

假如你想删除一个失败的工作，可以使用 `queue:forget` 命令：

```
php artisan queue:forget 5
```

要删除全部失败的工作，可以使用 `queue:flush` 命令：

```
php artisan queue:flush
```

会话

配置

由于 HTTP 协定是无状态（Stateless）的，所以 session 提供一种保存用户数据的方法。Laravel 支持了多种 session 后端驱动，并通过清楚、统一的 API 提供使用。也内置支持如 Memcached、Redis 和数据库的后端驱动。

session 的配置文件配置在 `config/session.php` 中，请务必看一下 session 配置文件中可用的选项配置及注释。Laravel 默认使用 `file` 的 session 驱动，它在大多的应用中可以良好运作。

如果你想在 Laravel 中使用 `Redis` sessions，你需要先通过 Composer 安装 `predis/predis` 扩展包 (~1.0)。

> 注意： 如果你需要加密所有的 session 数据，就将选项 `encrypt` 配置为 `true` 。

保留键值

Laravel 框架在内部有使用 `flash` 作为 session 的键值，所以应该避免 session 使用此名称。

使用 Session

获取 session 有很多种方式，可以通过 HTTP request 类的 `session` 方法，`Session` facade 或者 `session` 辅助函数。如果在调用 `session` 辅助函数时没有传入参数，会返回整个 session 对象。比如：

```
session()->regenerate();
```

保存对象到 Session 中

```
Session::put('key', 'value');

session(['key' => 'value']);
```

保存对象进 Session 数组值中

```
Session::push('user.teams', 'developers');
```

从 Session 取回对象

```
$value = Session::get('key');

$value = session('key');
```

从 **Session** 取回对象，若无则返回默认值

```
$value = Session::get('key', 'default');

$value = Session::get('key', function() { return 'default'; });
```

从 **Session** 取回对象，并删除

```
$value = Session::pull('key', 'default');
```

从 **Session** 取出所有对象

```
$data = Session::all();
```

判断对象在 **Session** 中是否存在

```
if (Session::has('users'))
{
    //
}
```

从 **Session** 中移除对象

```
Session::forget('key');
```

清空所有 **Session**

```
Session::flush();
```

重新产生 **Session ID**

```
Session::regenerate();
```

暂存数据（**Flash Data**）

有时你可能希望暂存一些数据，并只在下次请求有效。你可以使用 `Session::flash` 方法来达成目的：

```
Session::flash('key', 'value');
```

刷新当前暂存数据，延长到下次请求

```
Session::reflash();
```

只刷新指定快闪数据

```
Session::keep(array('username', 'email'));
```

数据库 Sessions

当使用 `database` session 驱动时，你必需建置一张保存 session 的数据表。下方例子使用 `Schema` 来建表：

```
Schema::create('sessions', function($table)
{
    $table->string('id')->unique();
    $table->text('payload');
    $table->integer('last_activity');
});
```

当然你也可以使用 Artisan 命令 `session:table` 来建 migration 表：

```
php artisan session:table

composer dump-autoload

php artisan migrate
```

Session 驱动

session 配置文件中的「driver」定义了 session 数据将以哪种方式被保存。Laravel 提供了许多良好的驱动：

- `file` - sessions 将保存在 `storage/framework/sessions`。
- `cookie` - sessions 将安全保存在加密的 cookies 中。
- `database` - sessions 将保存在你的应用程序数据库中。
- `memcached` / `redis` - sessions 将保存在一个高速缓存的系统中。
- `array` - sessions 将单纯的以 PHP 数组保存，只存活在当次请求。

> 注意： array 驱动典型应用在 unit tests 环境下，所以不会留下任何 session 数据。

模板

Blade 模板

Blade 是 Laravel 所提供的一个简单却又非常强大的模板引擎。不像控制器页面布局，Blade 是使用 模板继承(template inheritance) 和 区块(sections)。所有的 Blade 模板后缀名都要命名为 `.blade.php`。

定义一个 **Blade** 页面布局

```
<!-- Stored in resources/views/layouts/master.blade.php -->

<html>
    <body>
        @section('sidebar')
            This is the master sidebar.
        @show

        <div class="container">
            @yield('content')
        </div>
    </body>
</html>
```

在视图模板中使用 **Blade** 页面布局

```
@extends('layouts.master')

@section('sidebar')
    @@parent

    <p>This is appended to the master sidebar.</p>
@stop

@section('content')
    <p>This is my body content.</p>
@stop
```

请注意 如果视图 `继承(extend)` 了一个 Blade 页面布局会将页面布局中定义的区块用视图的所定义的区块重写。如果想要将页面布局中的区块内容也能在继承此布局的视图中呈现，那就要在区块中使用 `@@parent` 语法指令，通过这种方式可以把内容附加到页面布局中，我们会在侧边栏区块或者页脚区块看到类似的使用。

有时候，如您不确定这个区块内容有没有被定义，您可能会想要传一个默认的值给 @yield。您可以传入第二个参数作为默认值给 @yield：

```
@yield('section', 'Default Content')
```

其他 Blade 控制语法结构

在 Blade 视图中打印（Echoing）数据

```
Hello, {{ $name }}.

The current UNIX timestamp is {{ time() }}.
```

检查数据是否存在后再打印数据

有时候您想要打印一个变量，但您不确定这个变量是否存在，通常情况下，您会想要这样写：:

```
{{ isset($name) ? $name : 'Default' }}

然而，除了写这种三元运算符语法之外，Blade 让您可以使用下面这种更简便的语法：

{{ $name or 'Default' }}
```

使用花括号显示文字

如果您需要显示的一个字符串刚好被花括号包起来，您可以在花括号之前加上 @ 符号前缀来跳出 Blade 引擎的解析：

```
@{{ This will not be processed by Blade }}
```

如果您不想数据被转义, 也可以使用如下语法：

```
Hello, {!! $name !!}.
```

> 特别注意: 在您的应用程序打印用户所提供的内容时要非常小心。请记得永远使用双重花括号来转义内容中的 HTML 实体字符串。

If 声明

```
@if (count($records) === 1)
    I have one record!
@elseif (count($records) > 1)
    I have multiple records!
@else
    I don't have any records!
@endif

@unless (Auth::check())
    You are not signed in.
@endunless
```

循环

```
@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
```

加载子视图

```
@include('view.name')
```

您也可以通过传入数组的形式将数据传递给加载的子视图：

```
@include('view.name', ['some' => 'data'])
```

重写区块

如果想要重写掉前面区块中的内容，您可以使用 `overwrite` 声明：

```
@extends('list.item.container')

@section('list.item.content')
    <p>This is an item of type {{ $item->type }}</p>
@overwrite
```

显示语言行

```
@lang('language.line')

@choice('language.line', 1)
```

注释

```
{{-- This comment will not be in the rendered HTML --}}
```

扩展 Blade

Blade 甚至允许你定义自己的控制语法结构。 当一个 Blade 文件被编译时， 每一个自定义的扩展语法会与视图内容一起被调用， 您可以做任何的操作, 简单如 `str_replace` 以及更为复杂的正则表达式。

Blade 的编译器带有一些辅助方法 `createMatcher` 及 `createPlainMatcher` ，这些辅助方法可以产生您需要的表达式来帮助您构建自己的自定义扩展语法。

其中 `createPlainMatcher` 方法是用在没有参数的语法指令如 `@endif` 及 `@stop` 等， 而 `createMatcher` 方法是用在带参数的语法指令中。

下面的例子创建了一个 `@datetime($var)` 语法命令, 这个命令只是简单的对 `$var` 调用 `->format()` 方法：

```
Blade::extend(function($view, $compiler)
{
    $pattern = $compiler->createOpenMatcher('datetime');

    return preg_replace($pattern, '$1<?php echo $2->format(\'m/d/Y H:i\')); ?>', $view);
});
```

测试

介绍

Laravel 在建立时就有考虑到单元测试。事实上，它支持立即使用被引入的 PHPUnit 做测试，而且已经为你的应用程序建立了 `phpunit.xml` 文件。

在 `tests` 文件夹有提供一个测试例子。在安装新 Laravel 应用程序之后，只要在命令行上执行 `phpunit` 来进行测试流程。

定义并执行测试

要建立一个测试案例，只要在 `tests` 文件夹建立新的测试文件。测试类必须继承自 `TestCase`，接着你可以如你平常使用 PHPUnit 一般去定义测试方法。

测试类例子

```
class FooTest extends TestCase {

    public function testSomethingIsTrue()
    {
        $this->assertTrue(true);
    }

}
```

你可以从终端机执行 `phpunit` 命令来执行应用程序的所有测试。

> 注意: 如果你定义自己的 `setUp` 方法， 请记得调用 `parent::setUp`。

测试环境

当执行单元测试的时候，Laravel 会自动将环境配置成 `testing`。另外 Laravel 会在测试环境导入 `session` 和 `cache` 的配置文件。当在测试环境里这两个驱动会被配置为 `array` (空数组)，代表在测试的时候没有 session 或 cache 数据将会被保留。视情况你可以任意的建立你需要的测试环境配置。

`testing` 环境的变量可以在 `phpunit.xml` 文件中配置。

从测试调用路由

从单一测试中调用路由

你可以使用 `call` 方法，轻易地调用你的任何一个路由来测试：

```
$response = $this->call('GET', 'user/profile');

$response = $this->call($method, $uri, $parameters,  $cookies, $files, $server, $content)
```

接着你可以检查 `Illuminate\Http\Response` 对象：

```
$this->assertEquals('Hello World', $response->getContent());
```

从测试调用控制器

你也可以从测试调用控制器：

```
$response = $this->action('GET', 'HomeController@index');

$response = $this->action('GET', 'UserController@profile', array('user' => 1));
```

> 注意: 当使用 `action` 方法的时候，你不需要指定完整的控制器命名空间。只需要指定 `App\Http\Controllers` 命名空间后面的类名称部分。

`getContent` 方法会返回求值后的字串内容响应。如果你的路由返回一个 `View`，你可以通过 `original` 属性访问它：

```
$view = $response->original;

$this->assertEquals('John', $view['name']);
```

你可以使用 `callSecure` 方法去调用 HTTPS 路由：

```
$response = $this->callSecure('GET', 'foo/bar');
```

模拟 **Facades**

当测试的时候，你或许常会想要模拟调用 Laravel 静态 facade。举个例子，思考下面的控制器行为：

```
public function getIndex()
{
    Event::fire('foo', ['name' => 'Dayle']);

    return 'All done!';
}
```

我们可以在 facade 上使用 `shouldReceive` 方法，来模拟调用 `Event` 类，它将会返回一个 Mockery mock 对象实例。

模拟 Facade

```
public function testGetIndex()
{
    Event::shouldReceive('fire')->once()->with('foo', ['name' => 'Dayle']);

    $this->call('GET', '/');
}
```

> 注意: 你不应该模拟 `Request` facade。取而代之，当执行你的测试，传递想要的输入数据进去 `call` 方法。

框架 Assertions

Laravel 附带几个 `assert` 方法，让测试更简单一点：

Assert 响应为 OK

```
public function testMethod()
{
    $this->call('GET', '/');

    $this->assertResponseOk();
}
```

Assert 响应的状态码

```
$this->assertResponseStatus(403);
```

Assert 响应为重定向

```
$this->assertRedirectedTo('foo');

$this->assertRedirectedToRoute('route.name');

$this->assertRedirectedToAction('Controller@method');
```

Assert 响应的视图包含一些数据

```
public function testMethod()
{
    $this->call('GET', '/');

    $this->assertViewHas('name');
    $this->assertViewHas('age', $value);
}
```

Assert Session 包含一些数据

```
public function testMethod()
{
    $this->call('GET', '/');

    $this->assertSessionHas('name');
    $this->assertSessionHas('age', $value);
}
```

Assert Session 有错误信息

```
public function testMethod()
{
    $this->call('GET', '/');

    $this->assertSessionHasErrors();

    // Asserting the session has errors for a given key...
    $this->assertSessionHasErrors('name');

    // Asserting the session has errors for several keys...
    $this->assertSessionHasErrors(array('name', 'age'));
}
```

Assert 旧输入内容有一些数据

```
public function testMethod()
{
    $this->call('GET', '/');

    $this->assertHasOldInput();
}
```

辅助方法

`TestCase` 类包含几个辅助方法让应用程序的测试更为简单。

从测试里配置和刷新 Sessions

```
$this->session(['foo' => 'bar']);

$this->flushSession();
```

配置目前为通过身份验证的用户

你可以使用 `be` 方法配置目前为通过身份验证的用户：

```
$user = new User(array('name' => 'John'));
```

```
$this->be($user);
```

你可以从测试中使用 `seed` 方法重新填充你的数据库：

在测试中重新填充数据库

```
$this->seed();

$this->seed($connection);
```

更多建立填充数据的信息可以在文档的 迁移与数据填充 部分找到。

重置应用程序

你可能已经知道，你可以通过 `$this->app` 在任何测试方法中访问你的应用程序服务容器。这个服务容器会在每个测试类被重置。如果你希望在给定的方法手动强制重置应用程序，你可以从你的测试方法使用 `refreshApplication` 方法。这将会重置任何额外的绑定，例如那些从测试案例执行开始被放到 IoC 容器的 mocks。

表单验证

基本用法

Laravel 通过 `Validation` 类让您可以简单、方便的验证数据正确性及查看相应的验证错误信息。

基本验证例子

```
$validator = Validator::make(
    array('name' => 'Dayle'),
    array('name' => 'required|min:5')
);
```

上文中传递给 `make` 这个方法的第一个参数用来设定所需要被验证的数据名称，第二个参数设定该数据可被接受的规则。

使用数组来定义规则

多个验证规则可以使用"|"符号分隔，或是单一数组作为单独的元素分隔。

```
$validator = Validator::make(
    array('name' => 'Dayle'),
    array('name' => array('required', 'min:5'))
);
```

验证多个字段

```
$validator = Validator::make(
    array(
        'name' => 'Dayle',
        'password' => 'lamepassword',
        'email' => 'email@example.com'
    ),
    array(
        'name' => 'required',
        'password' => 'required|min:8',
        'email' => 'required|email|unique:users'
    )
);
```

当一个 `Validator` 实例被建立后，`fails`（或 `passes`）这两个方法就可以在验证时使用，如下：

```
if ($validator->fails())
{
    // The given data did not pass validation
}
```

假如验证失败，您可以从验证器中接收错误信息。

```
$messages = $validator->messages();
```

您可能不需要错误信息，只想取得无法通过验证的规则，您可以使用 `failed` 方法：

```
$failed = $validator->failed();
```

验证文件

`Validator` 类提供了一些规则用来验证文件，例如 `size`，`mimes` 等等。当需要验证文件时，您仅需将它们和您其他的数据一同送给验证器即可。

验证后钩子

验证器也允许你在完成验证后增加回调函数。这也允许你可以进行更进一步的验证，甚至在消息集合中增加更多的错误信息。我们在验证器实例中使用 `after` 方法来作为开始：

```
$validator = Validator::make(...);

$validator->after(function($validator)
{
    if ($this->somethingElseIsInvalid())
    {
        $validator->errors()->add('field', 'Something is wrong with this field!');
    }
});

if ($validator->fails())
{
    //
}
```

您可以根据需要为验证器增加任意的 `after` 回调函数。

控制器验证

当然，如果每一次需要验证的时候都手动的建立并且验证 `Validator` 实例会非常的麻烦。不用担心，你有其他的选择！Laravel自带的 `App\Http\Controllers\Controller` 基类使用了一个 `ValidatesRequests` 的 trait。这个 trait 提供了一个单一的、便捷的方法来验证 HTTP 请求。代码如下：

```
/**
 * Store the incoming blog post.
 *
 * @param  Request  $request
 * @return Response
 */
public function store(Request $request)
{
    $this->validate($request, [
        'title' => 'required|unique|max:255',
        'body' => 'required',
    ]);

    //
}
```

如果验证通过了，你的代码会正常继续执行。如果验证失败，那么会抛出一个 `Illuminate\Contracts\Validation\ValidationException` 异常。这个异常会被自动捕获，然后重定向至用户上一个页面。而错误信息甚至已经存储至 session 中！

如果收到的是一个 AJAX 请求，那么不会生成一个重定向。相反的，一个带有 422 状态码的 HTTP 响应会被返回给浏览器，包含了一个含有错误信息的 JSON 对象。

比如，如下是手动创建验证的等效写法：

```
/**
 * Store the incoming blog post.
 *
 * @param  Request  $request
 * @return Response
 */
public function store(Request $request)
{
    $v = Validator::make($request->all(), [
        'title' => 'required|unique|max:255',
        'body' => 'required',
    ]);

    if ($v->fails())
    {
        return redirect()->back()->withErrors($v->errors());
    }

    //
}
```

自定义闪存后的错误格式

如果你想要自定义验证失败后已经闪存至 session 的错误消息格式，可以通过覆盖基类控制器的 `formatValidationErrors` 。不要忘记在文件顶部引入 `Illuminate\Validation\Validator` 类。

```
/**
 * {@inheritdoc}
 */
protected function formatValidationErrors(Validator $validator)
```

```
{
    return $validator->errors()->all();
}
```

表单请求验证

如果是更复杂的验证场景，你可能需要创建一个"表单请求"。表单请求是一个自定义的请求类包含了一些验证的逻辑。你可以通过 Artisan 的命令行 `make:request` 来创建一个表单请求类。

```
php artisan make:request StoreBlogPostRequest
```

生成的类会放置在 `app/Http/Requests` 目录中。 我们在 `rules` 方法中增加一些验证规则：

```
/**
 * Get the validation rules that apply to the request.
 *
 * @return array
 */
public function rules()
{
    return [
        'title' => 'required|unique|max:255',
        'body' => 'required',
    ];
}
```

那么，我们的验证规则是怎么执行的呢？你所要做的只是在控制器方法中加上请求的类型提示：

```
/**
 * Store the incoming blog post.
 *
 * @param  StoreBlogPostRequest  $request
 * @return Response
 */
public function store(StoreBlogPostRequest $request)
{
    // The incoming request is valid...
}
```

当控制器的方法被调用前，表单请求已经验证了，意味着你不需要在控制器里写任何的验证逻辑。它已经验证完了！

如果验证失败，用户会收到一个重定向请求至上一个页面。而错误信息也已经存储至 session 中方便视图展示。如果收到的是一个 AJAX 请求，一个带有 422 状态码的 HTTP 响应会被返回给浏览器，包含了一个含有错误信息的 JSON 对象。

授权表单请求

表单请求类同样也包含了一个 `authorize` 方法。通过这个方法，你可以检查认证后的用户是否有权限去更新一个已有的资源。比如，如果一个用户尝试去更新一篇博客的评论，他是否真的发布过这个评论？举个例子：

```
/**
 * Determine if the user is authorized to make this request.
 *
 * @return bool
 */
public function authorize()
{
    $commentId = $this->route('comment');

    return Comment::where('id', $commentId)
                ->where('user_id', Auth::id())->exists();
}
```

注意上面例子中调用的 `route` 方法。这个方法允许你获取调用路由中定义的 URI 参数，比如下面例子中的 `{comment}` 参数：

```
Route::post('comment/{comment}');
```

如果 `authorize` 方法返回 `false` ， 一个带有 403 状态码的 HTTP 响应会被返回给浏览器，你控制器的方法也不会被执行。

如果你打算在应用的其他地方做一些权限的逻辑，在 `authorize` 方法中返回 `true` 即可：

```
/**
 * Determine if the user is authorized to make this request.
 *
 * @return bool
 */
public function authorize()
{
    return true;
}
```

自定义闪存后的错误信息格式

如果你想要自定义验证失败后已经闪存至 session 的错误消息格式，可以通过覆盖基类请求类(`App\Http\Requests\Request`)的 `formatValidationErrors` 。不要忘记在文件顶部引入 `Illuminate\Validation\Validator` 类。

```
/**
 * {@inheritdoc}
 */
protected function formatErrors(Validator $validator)
{
    return $validator->errors()->all();
}
```

使用错误信息

当您调用一个 `Validator` 实例的 `messages` 方法后，您会得到一个命名为 `MessageBag` 的实例，该实例

里有许多方便的方法能让您取得相关的错误信息。

查看一个字段的第一个错误信息

```
echo $messages->first('email');
```

查看一个字段的所有错误信息

```
foreach ($messages->get('email') as $message)
{
    //
}
```

查看所有字段的所有错误信息

```
foreach ($messages->all() as $message)
{
    //
}
```

判断一个字段是否有错误信息

```
if ($messages->has('email'))
{
    //
}
```

错误信息格式化输出

```
echo $messages->first('email', '<p>:message</p>');
```

> 注意： 默认错误信息以 Bootstrap 兼容语法输出。

查看所有错误信息并以格式化输出

```
foreach ($messages->all('<li>:message</li>') as $message)
{
    //
}
```

错误信息 & 视图

当您开始进行验证数据时，您会需要一个简易的方法去取得错误信息并返回到您的视图中，在 Laravel 中您可以很方便的处理这些操作，您可以通过下面的路由例子来了解：

```
Route::get('register', function()
```

```
{
    return View::make('user.register');
});

Route::post('register', function()
{
    $rules = array(...);

    $validator = Validator::make(Input::all(), $rules);

    if ($validator->fails())
    {
        return redirect('register')->withErrors($validator);
    }
});
```

需要记住的是，当验证失败后，我们会使用 `withErrors` 方法来将 `Validator` 实例进行重定向。这个方法会将错误信息存入 session 中，这样才能在下个请求中被使用。

然而，我们并不需要特别去将错误信息绑定在我们 GET 路由的视图中。因为 Laravel 会确认在 Session 数据中检查是否有错误信息，并且自动将它们绑定至视图中。所以请注意，**`$errors`** 变量存在于所有的视图中，所有的请求里，让您可以直接假设 `$errors` 变量已被定义且可以安全地使用。 `$errors` 变量是 `MessageBag` 类的一个实例。

所以，在重定向之后，您可以自然的在视图中使用 `$errors` 变量：

```
<?php echo $errors->first('email'); ?>
```

命名错误清单

假如您在一个页面中有许多的表单，您可能希望为错误命名一个 `MessageBag`。 这样能方便您针对特定的表单查看其错误信息, 我们只要简单的在 `withErrors` 的第二个参数设定名称即可：

```
return redirect('register')->withErrors($validator, 'login');
```

接着您可以从一个 `$errors` 变量中取得已命名的 `MessageBag` 实例：

```
<?php echo $errors->login->first('email'); ?>
```

可用验证规则

以下是现有可用的验证规则清单与他们的函数名称：

- Accepted
- Active URL
- After (Date)
- Alpha
- Alpha Dash
- Alpha Numeric

- Array
- Before (Date)
- Between
- Boolean
- Confirmed
- Date
- Date Format
- Different
- Digits
- Digits Between
- E-Mail
- Exists (Database)
- Image (File)
- In
- Integer
- IP Address
- Max
- MIME Types
- Min
- Not In
- Numeric
- Regular Expression
- Required
- Required If
- Required With
- Required With All
- Required Without
- Required Without All
- Same
- Size
- String
- Timezone
- Unique (Database)
- URL

accepted

字段值为 *yes*, *on*, 或是 *1* 时，验证才会通过。这在确认"服务条款"是否同意时很有用。

active_url

字段值通过 PHP 函数 `checkdnsrr` 来验证是否为一个有效的网址。

after:*date*

验证字段是否是在指定日期之后。这个日期将会使用 PHP `strtotime` 函数验证。

alpha

字段仅全数为字母字串时通过验证。

alpha_dash

字段值仅允许字母、数字、破折号（-）以及底线（_）

alpha_num

字段值仅允许字母、数字

array

字段值仅允许为数组

before:*date*

验证字段是否是在指定日期之前。这个日期将会使用 PHP `strtotime` 函数验证。

between:*min*,*max*

字段值需介于指定的 *min* 和 *max* 值之间。字串、数值或是文件都是用同样的方式来进行验证。

confirmed

字段值需与对应的字段值 `foo_confirmation` 相同。例如，如果验证的字段是 `password` ，那对应的字段 `password_confirmation` 就必须存在且与 `password` 字段相符。

date

字段值通过 PHP `strtotime` 函数验证是否为一个合法的日期。

date*format:_format*

字段值通过 PHP `date_parse_from_format` 函数验证符合 *format* 制定格式的日期是否为合法日期。

different:*field*

字段值需与指定的字段 *field* 值不同。

digits:*value*

字段值需为数字且长度需为 *value*。

digits*between:_min*,*max*

字段值需为数字，且长度需介于 *min* 与 *max* 之间。

boolean

字段必须可以转换成布尔值，可接受的值为 `true`，`false`，`1`，`0`，`"1"`，`"0"`。

email

字段值需符合 email 格式。

exists:*table*,*column*

字段值需与存在于数据库 *table* 中的 *column* 字段值其一相同。

Exists 规则的基本使用方法

```
'state' => 'exists:states'
```

指定一个自定义的字段名称

```
'state' => 'exists:states,abbreviation'
```

您可以指定更多条件且那些条件将会被新增至 "where" 查询里：

```
'email' => 'exists:staff,email,account_id,1'
/* 这个验证规则为 email 需存在于 staff 这个数据库表中 email 字段中且 account_id=1 */
```

通过 `NULL` 搭配"where"的缩写写法去检查数据库的是否为 `NULL`

```
'email' => 'exists:staff,email,deleted_at,NULL'
```

image

文件必需为图片(jpeg, png, bmp, gif 或 svg)

in:*foo*,*bar*,...

字段值需符合事先给予的清单的其中一个值

integer

字段值需为一个整数值

ip

字段值需符合 IP 位址格式。

max:*value*

字段值需小于等于 *value*。字串、数字和文件则是判断 `size` 大小。

mimes:*foo*,*bar*,...

文件的 MIME 类需在给定清单中的列表中才能通过验证。

MIME规则基本用法

```
'photo' => 'mimes:jpeg,bmp,png'
```

min:*value*

字段值需大于等于 *value*。字串、数字和文件则是判断 `size` 大小。

not*in:_foo*,*bar*,...

字段值不得为给定清单中其一。

numeric

字段值需为数字。

regex:*pattern*

字段值需符合给定的正规表示式。

注意: 当使用 `regex` 模式时，您必须使用数组来取代"|"作为分隔，尤其是当正规表示式中含有"|"字串。

required

字段值为必填。

required_if:*field*,*value*

字段值在 *field* 字段值为 *value* 时为必填。

required*with:_foo*,*bar*,...

字段值 仅在 任一指定字段有值情况下为必填。

required*with_all:_foo*,*bar*,...

字段值 仅在 所有指定字段皆有值情况下为必填。

required*without:_foo*,*bar*,...

字段值 仅在 任一指定字段没有值情况下为必填。

required*without_all:_foo*,*bar*,...

字段值 仅在 所有指定字段皆没有值情况下为必填。

same:*field*

字段值需与指定字段 *field* 等值。

size:*value*

字段值的尺寸需符合给定 *value* 值。对于字串来说，*value* 为需符合的字串长度。对于数字来说，*value* 为需符合的整数值。对于文件来说，*value* 为需符合的文件大小（单位 kb)。

timezone

字段值通过 PHP `timezone_identifiers_list` 函数来验证是否为有效的时区。

unique:*table*,*column*,*except*,*idColumn*

字段值在给定的数据库中需为唯一值。如果 `column（字段）` 选项没有指定，将会使用字段名称。

唯一(Unique)规则的基本用法

```
'email' => 'unique:users'
```

指定一个自定义的字段名称

```
'email' => 'unique:users,email_address'
```

强制唯一规则忽略指定的 ID

```
'email' => 'unique:users,email_address,10'
```

增加额外的 Where 条件

您也可以指定更多的条件式到 "where" 查询语句中：

```
'email' => 'unique:users,email_address,NULL,id,account_id,1'
```

上述规则为只有 `account_id` 为 `1` 的数据列会做唯一规则的验证。

url

字段值需符合 URL 的格式。

> 注意: 此函数会使用 PHP `filter_var` 方法验证。

添加条件验证规则

某些情况下，您可能 只想 当字段有值时，才进行验证。这时只要增加 `sometimes` 条件进条件列表中，就可以快速达成：

```
$v = Validator::make($data, array(
    'email' => 'sometimes|required|email',
));
```

在上述例子中，`email` 字段只会在当其在 `$data` 数组中有值的情况下才会被验证。

复杂的条件式验证

有时，您可以希望给指定字段在其他字段长度有超过 100 时才验证是否为必填。或者您希望有两个字段，当其中一字段有值时，另一字段将会有一个默认值。增加这样的验证条件并不复杂。首先，利用您尚未更动的 静态规则 创建一个 `Validator` 实例：

```
$v = Validator::make($data, array(
```

```
    'email' => 'required|email',
    'games' => 'required|numeric',
));
```

假设我们的网页应用程序是专为游戏收藏家所设计。如果游戏收藏家收藏超过一百款游戏，我们希望他们说明为什么他们拥有这么多游戏。如，可能他们经营一家二手游戏商店，或是他们可能只是享受收集的乐趣。有条件的加入此需求，我们可以在 `Validator` 实例中使用 `sometimes` 方法。

```
$v->sometimes('reason', 'required|max:500', function($input)
{
    return $input->games >= 100;
});
```

传递至 `sometimes` 方法的第一个参数是我们要条件式认证的字段名称。第二个参数是我们想加入验证规则。 闭包（Closure）作为第三个参数传入，如果返回值为 true 那该规则就会被加入。这个方法可以轻而易举的建立复杂的条件式验证。您也可以一次对多个字段增加条件式验证：

```
$v->sometimes(array('reason', 'cost'), 'required', function($input)
{
    return $input->games >= 100;
});
```

注意:

> 注意： 传递至您的 `Closure` 的 `$input` 参数为 `Illuminate\Support\Fluent` 的实例且用来作为获取您的输入及文件的对象。

自定义错误信息

如果有需要，您可以设置自定义的错误信息取代默认错误信息。这里有几种方式可以设定自定义消息。

传递自定义消息进验证器

```
$messages = array(
    'required' => 'The :attribute field is required.',
);

$validator = Validator::make($input, $rules, $messages);
```

> 注意： 在验证中，`:attribute` 占位符会被字段的实际名称给取代。您也可以在验证信息中使用其他的占位符。

其他的验证占位符

```
$messages = array(
    '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',
);
```

为特定属性赋予一个自定义信息

有时您只想为一个特定字段指定一个自定义错误信息：

```
$messages = array(
    'email.required' => 'We need to know your e-mail address!',
);
```

在语言包文件中指定自定义消息

某些状况下，您可能希望在语言包文件中设定您的自定义消息，而非直接将他们传递给 `Validator`。要达到这个目的，将您的信息增加至 `resources/lang/xx/validation.php` 文件的 `custom` 数组中。

```
'custom' => array(
    'email' => array(
        'required' => 'We need to know your e-mail address!',
    ),
),
```

自定义验证规则

注册自定义验证规则

Laravel 提供了各种有用的验证规则，但是，您可能希望可以设定自定义验证规则。注册生成自定义的验证规则的方法之一就是使用 `Validator::extend` 方法：

```
Validator::extend('foo', function($attribute, $value, $parameters)
{
    return $value == 'foo';
});
```

自定义验证器闭包接收三个参数：要被验证的 `$attribute(属性)` 的名称，属性的值 `$value`，传递至验证规则的 `$parameters` 数组。

您同样可以传递一个类和方法到 `extend` 方法中，取代原本的闭包：

```
Validator::extend('foo', 'FooValidator@validate');
```

注意，您同时需要为您的自定义规则制订一个错误信息。您可以使用行内自定义信息数组或是在认证语言文件里新增。

扩展 Validator 类

除了使用闭包回调来扩展 Validator 外，您一样可以直接扩展 Validator 类。您可以写一个扩展自 `Illuminate\Validation\Validator` 的验证器类。您也可以增加验证方法到以 `validate` 为开头的类中：

```php
<?php

class CustomValidator extends Illuminate\Validation\Validator {

    public function validateFoo($attribute, $value, $parameters)
    {
        return $value == 'foo';
    }

}
```

拓展自定义验证器解析器

接下来，您需要注册您自定义验证器扩展：

```php
Validator::resolver(function($translator, $data, $rules, $messages)
{
    return new CustomValidator($translator, $data, $rules, $messages);
});
```

当创建自定义验证规则时，您可能有时需要为错误信息定义自定义的占位符。您可以如上所述创建一个自定义的验证器，然后增加 `replaceXXX` 函数进验证器中。

```php
protected function replaceFoo($message, $attribute, $rule, $parameters)
{
    return str_replace(':foo', $parameters[0], $message);
}
```

如果您想要增加一个自定义信息 "replacer" 但不扩展 Validator 类，您可以使用 `Validator::replacer` 方法：

```php
Validator::replacer('rule', function($message, $attribute, $rule, $parameters)
{
    //
});
```

数据库使用基础

配置

Laravel 让连接数据库和执行查找变得相当容易。数据库相关配置文件都在 `config/database.php` 。 在这个文件你可以定义所有的数据库连接，以及指定默认的数据库连接。默认文件中已经有所有支持的数据库系统例子了。

目前 Laravel 支持四种数据库系统： MySQL、Postgres、SQLite、以及 SQL Server。

读取/写入连接

有时候你可能希望使用特定数据库连接进行 SELECT 操作，同时使用另外的连接进行 INSERT 、UPDATE 、以及 DELETE 操作。 Laravel 让这些变得轻松简单，并确保你不论在使用原始查找、查找构建器、或者是 Eloquent ORM 使用的都是正确的连接。

来看看如何配置读取/写入连接，让我们来看以下的例子：

```
'mysql' => [
    'read' => [
        'host' => '192.168.1.1',
    ],
    'write' => [
        'host' => '196.168.1.2'
    ],
    'driver'    => 'mysql',
    'database'  => 'database',
    'username'  => 'root',
    'password'  => '',
    'charset'   => 'utf8',
    'collation' => 'utf8_unicode_ci',
    'prefix'    => '',
],
```

注意我们加了两个键值到配置文件数组中： `read` 及 `write` 。 两个键值都包含了单一键值的数组： `host` 。 `read` 及 `write` 的其余数据库配置会从 `mysql` 数组中合并。 所以，如果我们想要覆写配置值，只要将选项放入 `read` 和 `write` 数组即可。 所以在上面的例子里， `192.168.1.1` 将被用作「读取」连接，而 `192.168.1.2` 将被用作「写入」连接。数据库凭证、 前缀、 字符编码配置、以及其他所有的配置会共用 `mysql` 数组里的配置。

执行查找

如果配置好数据库连接，就可以通过 `DB` facade 执行查找。

执行 **Select** 查找

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

`select` 方法会返回一个 `array` 结果。

执行 **Insert** 语法

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

执行 **Update** 语法

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

执行 **Delete** 语法

```
DB::delete('delete from users');
```

> 注意： `update` 和 `delete` 语法会返回在操作中所影响的数据笔数。

执行一般语法

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

监听查找事件

你可以使用 `DB::listen` 方法，去监听查找的事件：

```
DB::listen(function($sql, $bindings, $time)
{
    //
});
```

数据库事务处理

你可以使用 `transaction` 方法，去执行一组数据库事务处理的操作：

```
DB::transaction(function()
{
    DB::table('users')->update(['votes' => 1]);

    DB::table('posts')->delete();
});
```

> 注意： 在 `transaction` 闭包若抛出任何异常会导致事务自动回滚。

有时候你可能需要自己开始一个事务：

```
DB::beginTransaction();
```

你可以通过 `rollback` 的方法回滚事务：

```
DB::rollback();
```

最后，你可以通过 `commit` 的方法提交事务：

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

获取连接

若要使用多个连接，可以通过 `DB::connection` 方法取用：

```
$users = DB::connection('foo')->select(...);
```

你也可以取用原始底层的 PDO 实例：

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

有时候你可能需要重新连接到特定的数据库：

```
DB::reconnect('foo');
```

如果你因为超过了底层 PDO 实例的 `max_connections` 的限制，需要关闭特定的数据库连接，可以通过 `disconnect` 方法:

```
DB::disconnect('foo');
```

查找日志记录

Laravel 可以在内存里访问这次请求中所有的查找语句。然而在有些例子下要注意，比如一次添加 大量的数据，可能会导致应用程序耗损过多内存。 如果要启用日志，可以使用 `enableQueryLog` 方法：

```
DB::connection()->enableQueryLog();
```

要得到执行过的查找纪录数组，你可以使用 `getQueryLog` 方法：

```
    $queries = DB::getQueryLog();
```

查询构造器

介绍

数据库查询构造器 (query builder) 提供方便、流畅的接口，用来建立及执行数据库查找语法。在你的应用程序里面，它可以被使用在大部分的数据库操作，而且它在所有支持的数据库系统上都可以执行。

> 注意: Laravel 查询构造器使用 PDO 参数绑定，以保护应用程序免于 SQL 注入，因此传入的参数不需额外转义特殊字符。

Selects

从数据表中取得所有的数据列

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

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

从数据表中分块查找数据列

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

通过在 `闭包` 中返回 `false` 来停止处理接下来的数据列：

```
DB::table('users')->chunk(100, function($users)
{
    //
```

```
    return false;
});
```

从数据表中取得单一数据列

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

var_dump($user->name);
```

从数据表中取得单一数据列的单一字段

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

取得单一字段值的列表

```
$roles = DB::table('roles')->lists('title');
```

这个方法将会返回数据表 role 的 title 字段值的数组。你也可以通过下面的方法，为返回的数组指定自定义键值。

```
$roles = DB::table('roles')->lists('title', 'name');
```

指定查询子句 **(Select Clause)**

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

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

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

增加查询子句到现有的查询中

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

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

使用 **where** 及运算符

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

「**or**」语法

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

```
                    ->orWhere('name', 'John')
                    ->get();
```

使用 **Where Between**

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

使用 **Where Not Between**

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

使用 **Where In** 与数组

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

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

使用 **Where Null** 找有未配置的值的数据

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

排序**(Order By)**、分群**(Group By)** 及 **Having**

```
$users = DB::table('users')
                    ->orderBy('name', 'desc')
                    ->groupBy('count')
                    ->having('count', '>', 100)
                    ->get();
```

偏移**(Offset)** 及 限制**(Limit)**

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

Joins

查询构造器也可以使用 join 语法，看看下面的例子：

基本的 **Join** 语法

```
DB::table('users')
            ->join('contacts', 'users.id', '=', 'contacts.user_id')
```

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

Left Join 语法

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

你也可以指定更高级的 join 子句：

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

如果你想在你的 join 中使用 where 型式的子句，你可以在 join 子句里使用 `where` 或 `orWhere` 方法。下面的方法将会比较 contacts 数据表中的 user_id 的数值，而不是比较两个字段。

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

高级 Wheres

群组化参数

有些时候你需要更高级的 where 子句，如「where exists」或嵌套的群组化参数。Laravel 的查询构造器也可以处理这样的情况：

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

上面的查找语法会产生下方的 SQL：

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

Exists 语法

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

上面的查找语法会产生下方的 SQL：

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

聚合

查找产生器也提供各式各样的聚合方法，如 `count`、 `max`、 `min`、 `avg` 及 `sum`。

使用聚合方法

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

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

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

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

$total = DB::table('users')->sum('votes');
```

原生表达式

有些时候你需要使用原生表达式在查找语句里，这样的表达式会成为字串插入至查找，因此要小心勿建立任何 SQL 注入点。要建立原生表达式，你可以使用 `DB::raw` 方法：

使用原生表达式

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

添加

添加数据进数据表

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

添加自动递增 (Auto-Incrementing) ID 的数据至数据表

如果数据表有自动递增的ID，可以使用 `insertGetId` 添加数据并返回该 ID：

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

> 注意: 当使用 PostgreSQL 时，insertGetId 方法会预期自动增加的字段是以「id」为命名。

添加多个数据进数据表

```
DB::table('users')->insert(array(
    array('email' => 'taylor@example.com', 'votes' => 0),
    array('email' => 'dayle@example.com', 'votes' => 0),
));
```

更新

更新数据表中的数据

```
DB::table('users')
            ->where('id', 1)
            ->update(array('votes' => 1));
```

自增或自减一个字段的值

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

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

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

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

也能够同时指定其他要更新的字段：

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

删除

删除数据表中的数据

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

删除数据表中的所有数据

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

清空数据表

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

Unions

查询构造器也提供一个快速的方法去「合并 (union)」两个查找的结果：

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

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

`unionAll` 方法也可以使用，它与 `union` 方法的使用方式一样。

悲观锁定 (Pessimistic Locking)

查询构造器提供了少数函数协助你在 SELECT 语句中做到「悲观锁定」。

想要在 SELECT 语句中加上「Shard lock」，只要在查找语句中使用 `sharedLock` 函数：

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

要在 select 语法中使用「锁住更新(lock for update)」时，你可以使用 `lockForUpdate` 方法：

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

Eloquent ORM

介绍

Laravel 的 Eloquent ORM 提供了漂亮、简洁的 ActiveRecord 实现来和数据库的互动。 每个数据库表会和一个对应的「模型」互动。

在开始之前，记得把 `config/database.php` 里的数据库连接配置好。

基本用法

我们先从建立一个 Eloquent 模型开始。模型通常放在 `app` 目录下，但是您可以将它们放在任何地方，只要能通过 composer.json 自动载入。所有的 Eloquent 模型都继承于 `Illuminate\Database\Eloquent\Model` 。

定义一个 **Eloquent** 模型

```
class User extends Model {}
```

你也可以通过 `make:model` 命令自动生成 Eloquent 模型：

```
php artisan make:model User
```

注意我们并没有告诉 Eloquent User 模型会使用哪个数据库表。若没有特别指定，系统会默认自动对应名称为「类名称的小写复数形态」的数据库表。所以，在上面的例子中， Eloquent 会假设 `User` 模型将把数据存在 `users` 数据库表。您也可以在类中定义 `table` 属性自定义要对应的数据库表。

```
class User extends Model {

    protected $table = 'my_users';

}
```

> 注意： Eloquent 也会假设每个数据库表都有一个字段名称为 `id` 的主键。您可以在类里定义 `primaryKey` 属性来重写。同样的，您也可以定义 `connection` 属性，指定模型连接到指定的数据库连接。

定义好模型之后，您就可以从数据库表新增及获取数据了。注意在默认情况下，在数据库表里需要有 `updated_at` 和 `created_at` 两个字段。如果您不想设定或自动更新这两个字段，则将类里的 `$timestamps` 属性设为 false即可。

取出所有模型数据

```
$users = User::all();
```

根据主键取出一条数据

```
$user = User::find(1);

var_dump($user->name);
```

> 提示： 所有查询构造器里的方法，查询 Eloquent 模型时也可以使用。

根据主键取出一条数据或抛出异常

有时, 您可能想要在找不到模型数据时抛出异常，通过 `App::error` 捕捉异常处理并显示 404 页面。

```
$model = User::findOrFail(1);

$model = User::where('votes', '>', 100)->firstOrFail();
```

要注册错误处理，可以监听 `ModelNotFoundException`

```
use Illuminate\Database\Eloquent\ModelNotFoundException;

App::error(function(ModelNotFoundException $e)
{
    return Response::make('Not Found', 404);
});
```

Eloquent 模型结合查询语法

```
$users = User::where('votes', '>', 100)->take(10)->get();

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

Eloquent 聚合查询

当然，您也可以使用查询构造器的聚合查询方法。

```
$count = User::where('votes', '>', 100)->count();
```

如果没办法使用流畅接口产生出查询语句，也可以使用 `whereRaw` 方法：

```
$users = User::whereRaw('age > ? and votes = 100', array(25))->get();
```

拆分查询

如果您要处理非常多（数千条）Eloquent 查询结果，使用 `chunk` 方法可以让您顺利工作而不会消耗大量内存：

```
User::chunk(200, function($users)
{
    foreach ($users as $user)
    {
        //
    }
});
```

传到方法里的第一个参数表示每次「拆分」要取出的数据数量。第二个参数的闭合函数会在每次取出数据时被调用。

指定查询时连接数据库

您也可以指定在执行 Eloquent 查询时要使用哪个数据库连接。只要使用 `on` 方法：

```
$user = User::on('connection-name')->find(1);
```

如果您在使用 读取 / 写入连接, 您可以通过如下命令来强制查询使用 `写入` 连接：

```
$user = User::onWriteConnection()->find(1);
```

批量赋值

在建立一个新的模型时，您把属性以数组的方式传入模型的构造方法，这些属性值会经由批量赋值存成模

型数据。这一点非常方便，然而，若盲目地将用户输入存到模型时，可能会造成严重的安全隐患。如果盲目的存入用户输入，用户可以随意的修改任何以及所有模型的属性。基于这个理由，所有的 Eloquent 模型默认会阻止批量赋值 。

我们以在模型里设定 `fillable` 或 `guarded` 属性作为开始。

定义模型 `Fillable` 属性

`fillable` 属性指定了哪些字段支持批量赋值 。可以设定在类的属性里或是实例化后设定。

```
class User extends Model {

    protected $fillable = array('first_name', 'last_name', 'email');

}
```

在上面的例子里，只有三个属性允许批量赋值。

定义模型 `Guarded` 属性

`guarded` 与 `fillable` 相反，是作为「黑名单」而不是「白名单」：

```
class User extends Model {

    protected $guarded = array('id', 'password');

}
```

> 注意： 使用 `guarded` 时， `Input::get()` 或任何用户可以控制的未过滤数据，永远不应该传入 `save` 或 `update` 方法，因为没有在「黑名单」内的字段可能被更新。

阻挡所有属性被批量赋值

上面的例子中， `id` 和 `password` 属性不会被批量赋值，而所有其他的属性则允许批量赋值。您也可以使用 guard 属性阻止所有属性被批量赋值：

```
protected $guarded = array('*');
```

新增，更新，删除

要从模型新增一条数据到数据库，只要建立一个模型实例并调用 `save` 方法即可。

储存新的模型数据

```
$user = new User;

$user->name = 'John';

$user->save();
```

> 注意： 通常 Eloquent 模型主键值会自动递增。但是您若想自定义主键，将 `incrementing` 属性设成 false 。

也可以使用 `create` 方法存入新的模型数据，新增完后会返回新增的模型实例。但是在新增前，需要先在模型类里设定好 `fillable` 或 `guarded` 属性，因为 Eloquent 默认会防止批量赋值。

在新模型数据被储存或新增后，若模型有自动递增主键，可以从对象取得 `id` 属性值：

```
$insertedId = $user->id;
```

在模型里设定 **Guarded** 属性

```
class User extends Model {

    protected $guarded = array('id', 'account_id');

}
```

使用模型的 **Create** 方法

```
// 在数据库中建立一个新的用户...
$user = User::create(array('name' => 'John'));

// 以属性找用户，若没有则新增并取得新的实例...
$user = User::firstOrCreate(array('name' => 'John'));

// 以属性找用户，若没有则建立新的实例...
$user = User::firstOrNew(array('name' => 'John'));
```

更新取出的模型

要更新模型，可以取出它，更改属性值，然后使用 `save` 方法：

```
$user = User::find(1);

$user->email = 'john@foo.com';

$user->save();
```

储存模型和关联数据

有时您可能不只想要储存模型本身，也想要储存关联的数据。您可以使用 `push` 方法达到目的：

```
$user->push();
```

您可以结合查询语句，批次更新模型：

```
$affectedRows = User::where('votes', '>', 100)->update(array('status' => 2));
```

> 注意： 若使用 Eloquent 查询构造器批次更新模型，则不会触发模型事件。

删除模型

要删除模型，只要使用实例调用 `delete` 方法：

```
$user = User::find(1);

$user->delete();
```

按主键值删除模型

```
User::destroy(1);

User::destroy(array(1, 2, 3));

User::destroy(1, 2, 3);
```

当然，您也可以结合查询语句批次删除模型：

```
$affectedRows = User::where('votes', '>', 100)->delete();
```

只更新模型的时间戳

如果您只想要更新模型的时间戳，您可以使用 `touch` 方法：

```
$user->touch();
```

软删除

通过软删除方式删除了一个模型后，模型中的数据并不是真的从数据库被移除。而是会设定 `deleted_at` 时间戳。要让模型使用软删除功能，只要在模型类里加入 `SoftDeletingTrait` 即可：

```
use Illuminate\Database\Eloquent\SoftDeletes;

class User extends Model {

    use SoftDeletes;

    protected $dates = ['deleted_at'];

}
```

要加入 `deleted_at` 字段到数据库表，可以在迁移文件里使用 `S oftDeletes` 方法：

```
$table->softDeletes();
```

现在当您使用模型调用 `delete` 方法时， `deleted_at` 字段会被更新成现在的时间戳。在查询使用软删除功能的模型时，被「删除」的模型数据不会出现在查询结果里。

强制查询软删除数据

要强制让已被软删除的模型数据出现在查询结果里，在查询时使用 `withTrashed` 方法：

```
$users = User::withTrashed()->where('account_id', 1)->get();
```

`withTrashed` 也可以用在关联查询：

```
$user->posts()->withTrashed()->get();
```

如果您只想查询被软删除的模型数据，可以使用 `onlyTrashed` 方法：

```
$users = User::onlyTrashed()->where('account_id', 1)->get();
```

要把被软删除的模型数据恢复，使用 `restore` 方法：

```
$user->restore();
```

您也可以结合查询语句使用 `restore` ：

```
User::withTrashed()->where('account_id', 1)->restore();
```

如同 `withTrashed` ， `restore` 方法也可以用在关联对象：

```
$user->posts()->restore();
```

如果想要真的从模型数据库删除，使用 `forceDelete` 方法：

```
$user->forceDelete();
```

`forceDelete` 方法也可以用在关联对象：

```
$user->posts()->forceDelete();
```

要确认模型是否被软删除了，可以使用 `trashed` 方法：

```
if ($user->trashed())
{
    //
}
```

时间戳

默认 Eloquent 会自动维护数据库表的 `created_at` 和 `updated_at` 字段。只要把这两个「时间戳」字段加到数据库表， Eloquent 就会处理剩下的工作。如果不想让 Eloquent 自动维护这些字段，把下面的属性加到模型类里：

关闭自动更新时间戳

```
class User extends Model {

    protected $table = 'users';

    public $timestamps = false;

}
```

自定义时间戳格式

如果想要自定义时间戳格式，可以在模型类里重写 `getDateFormat` 方法：

```
class User extends Model {

    protected function getDateFormat()
    {
        return 'U';
    }

}
```

范围查询

定义范围查询

范围查询可以让您轻松的重复利用模型的查询逻辑。要设定范围查询，只要定义有 `scope` 前缀的模型方法：

```
class User extends Model {

    public function scopePopular($query)
    {
        return $query->where('votes', '>', 100);
    }

    public function scopeWomen($query)
    {
        return $query->whereGender('W');
    }

}
```

使用范围查询

```
$users = User::popular()->women()->orderBy('created_at')->get();
```

动态范围查询

有时您可能想要定义可接受参数的范围查询方法。只要把参数加到方法里：

```
class User extends Model {

    public function scopeOfType($query, $type)
    {
        return $query->whereType($type);
    }

}
```

然后把参数值传到范围查询方法调用里：

```
$users = User::ofType('member')->get();
```

Global Scopes

有时您可能希望定义一个 scope 可以用于模型的所有查询中。本质上，这也是 Eloquent 的"软删除"功能的实现原理。Global scopes 是通过 PHP traits 的组合以及实现 `Illuminate\Database\Eloquent\ScopeInterface` 接口来定义的。

首先，我们需要定义一个 trait。 这里我们用 Laravel 的 `SoftDeletes` 举例：

```
trait SoftDeletes {

    /**
     * Boot the soft deleting trait for a model.
     *
     * @return void
     */
    public static function bootSoftDeletes()
    {
        static::addGlobalScope(new SoftDeletingScope);
    }

}
```

如果一个 Eloquent 模型引入了一个 trait ，而这个 trait 中带有符合 `bootNameOfTrait` 惯例命名的方法 ,那么这个方法会在 Eloquent 模型启动的时候调用， 您可以在此时注册 global scope ，或者做一些其他您想要的操作。定义的 scope 必须实现 `ScopeInterface` 接口，这个接口提供了两个方法： `apply` 和 `remove`。

`apply` 方法接受一个 `Illuminate\Database\Eloquent\Builder` 查询构造器对象以及它所应用的 `Model` ，用来添加这个 scope 所需的额外的 `where` 子句。而 `remove` 方法同样接受一个 `Builder` 对象

以及 `Model` ，用来反向的执行 `apply` 操作。也就是说， `remove` 方法应该移除已经添加的 `where` 子句(或者其他查询子句)。因此，我们的 `SoftDeletingScope` 的方法应该如下：

```
/**
 * Apply the scope to a given Eloquent query builder.
 *
 * @param  \Illuminate\Database\Eloquent\Builder  $builder
 * @return void
 */
public function apply(Builder $builder, Model $model)
{
    $model = $builder->getModel();

    $builder->whereNull($model->getQualifiedDeletedAtColumn());
}

/**
 * Remove the scope from the given Eloquent query builder.
 *
 * @param  \Illuminate\Database\Eloquent\Builder  $builder
 * @return void
 */
public function remove(Builder $builder, Model $model)
{
    $column = $model->getQualifiedDeletedAtColumn();

    $query = $builder->getQuery();

    foreach ((array) $query->wheres as $key => $where)
    {
        // If the where clause is a soft delete date constraint, we will remove it from
        // the query and reset the keys on the wheres. This allows this developer to
        // include deleted model in a relationship result set that is lazy loaded.
        if ($this->isSoftDeleteConstraint($where, $column))
        {
            unset($query->wheres[$key]);

            $query->wheres = array_values($query->wheres);
        }
    }
}
```

关联

当然，您的数据库表很可能跟另一张表相关联。例如，一篇 blog 文章可能有很多评论，或是一张订单跟下单客户相关联。 Eloquent 让管理和处理这些关联变得很容易。 Laravel 有很多种关联类型：

- 一对一
- 一对多
- 多对多
- 远层一对多关联
- 多态关联
- 多态的多对多关联

一对一

定义一对一关联

一对一关联是很基本的关联。例如一个 `User` 模型会对应到一个 `Phone` 。 在 Eloquent 里可以像下面这样定义关联：

```
class User extends Model {

    public function phone()
    {
        return $this->hasOne('App\Phone');
    }

}
```

传到 `hasOne` 方法里的第一个参数是关联模型的类名称。定义好关联之后，就可以使用 Eloquent 的动态属性取得关联对象：

```
$phone = User::find(1)->phone;
```

SQL 会执行如下语句：

```
select * from users where id = 1

select * from phones where user_id = 1
```

注意， Eloquent 假设对应的关联模型数据库表里，外键名称是基于模型名称。在这个例子里，默认 `Phone` 模型数据库表会以 `user_id` 作为外键。如果想要更改这个默认，可以传入第二个参数到 `hasOne` 方法里。更进一步，您可以传入第三个参数，指定关联的外键要对应到本身的哪个字段：

```
return $this->hasOne('App\Phone', 'foreign_key');

return $this->hasOne('App\Phone', 'foreign_key', 'local_key');
```

定义相对的关联

要在 `Phone` 模型里定义相对的关联，可以使用 `belongsTo` 方法：

```
class Phone extends Model {

    public function user()
    {
        return $this->belongsTo('App\User');
    }

}
```

在上面的例子里， Eloquent 默认会使用 `phones` 数据库表的 `user_id` 字段查询关联。如果想要自己指定外键字段，可以在 `belongsTo` 方法里传入第二个参数：

```
class Phone extends Model {

    public function user()
    {
        return $this->belongsTo('App\User', 'local_key');
    }

}
```

除此之外，也可以传入第三个参数指定要参照上层数据库表的哪个字段：

```
class Phone extends Model {

    public function user()
    {
        return $this->belongsTo('App\User', 'local_key', 'parent_key');
    }

}
```

一对多

一对多关联的例子如，一篇 Blog 文章可能「有很多」评论。可以像这样定义关联：

```
class Post extends Model {

    public function comments()
    {
        return $this->hasMany('App\Comment');
    }

}
```

现在可以经由动态属性取得文章的评论：

```
$comments = Post::find(1)->comments;
```

如果需要增加更多条件限制，可以在调用 `comments` 方法后面通过链式查询条件方法：

```
$comments = Post::find(1)->comments()->where('title', '=', 'foo')->first();
```

同样的，您可以传入第二个参数到 `hasMany` 方法更改默认的外键名称。以及，如同 `hasOne` 关联，可以指定本身的对应字段：

```
return $this->hasMany('App\Comment', 'foreign_key');

return $this->hasMany('App\Comment', 'foreign_key', 'local_key');
```

定义相对的关联

要在 `Comment` 模型定义相对应的关联，可使用 `belongsTo` 方法：

```
class Comment extends Model {

    public function post()
    {
        return $this->belongsTo('App\Post');
    }

}
```

多对多

多对多关联更为复杂。这种关联的例子如，一个用户（ user ）可能用有很多身份（ role ），而一种身份可能很多用户都有。例如很多用户都是「管理者」。多对多关联需要用到三个数据库表： `users` ， `roles` ，和 `role_user` 。 `role_user` 枢纽表命名是以相关联的两个模型数据库表，依照字母顺序命名，枢纽表里面应该要有 `user_id` 和 `role_id` 字段。

可以使用 `belongsToMany` 方法定义多对多关系：

```
class User extends Model {

    public function roles()
    {
        return $this->belongsToMany('App\Role');
    }

}
```

现在我们可以从 `User` 模型取得 roles：

```
$roles = User::find(1)->roles;
```

如果不想使用默认的枢纽数据库表命名方式，可以传递数据库表名称作为 `belongsToMany` 方法的第二个参数：

```
return $this->belongsToMany('App\Role', 'user_roles');
```

也可以更改默认的关联字段名称：

```
return $this->belongsToMany('App\Role', 'user_roles', 'user_id', 'foo_id');
```

当然，也可以在 `Role` 模型定义相对的关联：

```
class Role extends Model {

    public function users()
    {
```

```
        return $this->belongsToMany('App\User');
    }

}
```

Has Many Through 远层一对多关联

「远层一对多关联」提供了方便简短的方法，可以经由多层间的关联取得远层的关联。例如，一个 `Country` 模型可能通过 `Users` 关联到很多 `Posts` 模型。 数据库表间的关系可能看起来如下：

```
countries
    id - integer
    name - string

users
    id - integer
    country_id - integer
    name - string

posts
    id - integer
    user_id - integer
    title - string
```

虽然 `posts` 数据库表本身没有 `country_id` 字段，但 `hasManyThrough` 方法让我们可以使用 `$country->posts` 取得 country 的 posts。我们可以定义以下关联：

```
class Country extends Model {

    public function posts()
    {
        return $this->hasManyThrough('App\Post', 'User');
    }

}
```

如果想要手动指定关联的字段名称，可以传入第三和第四个参数到方法里：

```
class Country extends Model {

    public function posts()
    {
        return $this->hasManyThrough('App\Post', 'User', 'country_id', 'user_id');
    }

}
```

多态关联

多态关联可以用一个简单的关联方法，就让一个模型同时关联多个模型。例如，您可能想让 photo 模型同时和一个 staff 或 order 模型关联。可以定义关联如下：

```
class Photo extends Model {

    public function imageable()
    {
        return $this->morphTo();
    }

}

class Staff extends Model {

    public function photos()
    {
        return $this->morphMany('App\Photo', 'imageable');
    }

}

class Order extends Model {

    public function photos()
    {
        return $this->morphMany('App\Photo', 'imageable');
    }

}
```

取得多态关联对象

现在我们可以从 staff 或 order 模型取得多态关联对象：

```
$staff = Staff::find(1);

foreach ($staff->photos as $photo)
{
    //
}
```

取得多态关联对象的拥有者

然而，多态关联真正神奇的地方，在于要从 `Photo` 模型取得 staff 或 order 对象时：

```
$photo = Photo::find(1);

$imageable = $photo->imageable;
```

Photo 模型里的 `imageable` 关联会返回 `Staff` 或 `Order` 实例，取决于这是哪一种模型拥有的照片。

多态关联的数据库表结构

为了理解多态关联的运作机制，来看看它们的数据库表结构：

```
staff
    id - integer
```

```
    name - string

orders
    id - integer
    price - integer

photos
    id - integer
    path - string
    imageable_id - integer
    imageable_type - string
```

要注意的重点是 `photos` 数据库表的 `imageable_id` 和 `imageable_type`。在上面的例子里， ID 字段会包含 staff 或 order 的 ID，而 type 是拥有者的模型类名称。这就是让 ORM 在取得 `imageable` 关联对象时，决定要哪一种模型对象的机制。

多态的多对多关联

Polymorphic Many To Many Relation Table Structure 多态的多对多关联数据库表结构

除了一般的多态关联，也可以使用多对多的多态关联。例如，Blog 的 `Post` 和 `Video` 模型可以共用多态的 `Tag` 关联模型。首先，来看看数据库表结构：

```
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
```

现在，我们准备好设定模型关联了。 `Post` 和 `Video` 模型都可以经由 `tags` 方法建立 `morphToMany` 关联：

```
class Post extends Model {

    public function tags()
    {
        return $this->morphToMany('App\Tag', 'taggable');
    }

}
```

在 `Tag` 模型里针对每一种关联建立一个方法：

```
class Tag extends Model {

    public function posts()
    {
        return $this->morphedByMany('App\Post', 'taggable');
    }

    public function videos()
    {
        return $this->morphedByMany('App\Video', 'taggable');
    }

}
```

关联查询

根据关联条件查询

在取得模型数据时，您可能想要以关联模型作为查询限制。例如，您可能想要取得所有「至少有一篇评论」的Blog 文章。可以使用 `has` 方法达成目的：

```
$posts = Post::has('comments')->get();
```

也可以指定运算符和数量：

```
$posts = Post::has('comments', '>=', 3)->get();
```

也可以使用"点号"的形式来获取嵌套的 `has` 声明：

```
$posts = Post::has('comments.votes')->get();
```

如果想要更进阶的用法，可以使用 `whereHas` 和 `orWhereHas` 方法，在 `has` 查询里设置 "where" 条件：

```
$posts = Post::whereHas('comments', function($q)
{
    $q->where('content', 'like', 'foo%');

})->get();
```

动态属性

Eloquent 可以经由动态属性取得关联对象。 Eloquent 会自动进行关联查询，而且会很聪明的知道应该要使用 `get`（用在一对多关联）或是 `first`（用在一对一关联）方法。可以经由和「关联方法名称相同」的动态属性取得对象。例如，如下面的模型对象 `$phone`：

```
class Phone extends Model {
```

```
    public function user()
    {
        return $this->belongsTo('App\User');
    }

}

$phone = Phone::find(1);
```

您可以不用像下面这样打印用户的 email ：

```
echo $phone->user()->first()->email;
```

而可以简写如下：

```
echo $phone->user->email;
```

> 注意： 若取得的是许多关联对象，会返回 `Illuminate\Database\Eloquent\Collection` 对象。

预载入

预载入是用来减少 N + 1 查询问题。例如，一个 `Book` 模型数据会关联到一个 `Author` 。关联会像下面这样定义：

```
class Book extends Model {

    public function author()
    {
        return $this->belongsTo('App\Author');
    }

}
```

现在考虑下面的代码：

```
foreach (Book::all() as $book)
{
    echo $book->author->name;
}
```

上面的循环会执行一次查询取回所有数据库表上的书籍，然而每本书籍都会执行一次查询取得作者。所以若我们有 25 本书，就会进行 26次查询。

很幸运地，我们可以使用预载入大量减少查询次数。使用 `with` 方法指定想要预载入的关联对象：

```
foreach (Book::with('author')->get() as $book)
{
    echo $book->author->name;
}
```

现在，上面的循环总共只会执行两次查询：

```
select * from books

select * from authors where id in (1, 2, 3, 4, 5, ...)
```

使用预载入可以大大提高程序的性能。

当然，也可以同时载入多种关联：

```
$books = Book::with('author', 'publisher')->get();
```

甚至可以预载入巢状关联：

```
$books = Book::with('author.contacts')->get();
```

上面的例子中， `author` 关联会被预载入， author 的 `contacts` 关联也会被预载入。

预载入条件限制

有时您可能想要预载入关联，同时也想要指定载入时的查询限制。下面有一个例子：

```
$users = User::with(array('posts' => function($query)
{
    $query->where('title', 'like', '%first%');

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

上面的例子里，我们预载入了 user 的 posts 关联，并限制条件为 post 的 title 字段需包含 "first" 。

当然，预载入的闭合函数里不一定只能加上条件限制，也可以加上排序：

```
$users = User::with(array('posts' => function($query)
{
    $query->orderBy('created_at', 'desc');

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

延迟预载入

也可以直接从模型的 collection 预载入关联对象。这对于需要根据情况决定是否载入关联对象时，或是跟缓存一起使用时很有用。

```
$books = Book::all();

$books->load('author', 'publisher');
```

你可以传入一个闭包来对查询构建器进行条件限制：

```
$books->load(['author' => function($query)
{
    $query->orderBy('published_date', 'asc');
}]);
```

新增关联模型

附加一个关联模型

您常常会需要加入新的关联模型。例如新增一个 comment 到 post 。除了手动设定模型的 `post_id` 外键，也可以从上层的 `Post` 模型新增关联的 comment ：

```
$comment = new Comment(array('message' => 'A new comment.'));

$post = Post::find(1);

$comment = $post->comments()->save($comment);
```

上面的例子里，新增的 comment 模型中 `post_id` 字段会被自动设定。

如果想要同时新增很多关联模型：

```
$comments = array(
    new Comment(array('message' => 'A new comment.')),
    new Comment(array('message' => 'Another comment.')),
    new Comment(array('message' => 'The latest comment.'))
);

$post = Post::find(1);

$post->comments()->saveMany($comments);
```

从属关联模型 (Belongs To)

要更新 `belongsTo` 关联时，可以使用 `associate` 方法。这个方法会设定子模型的外键：

```
$account = Account::find(10);

$user->account()->associate($account);

$user->save();
```

新增多对多关联模型 (Many To Many)

您也可以新增多对多的关联模型。让我们继续使用 `User` 和 `Role` 模型作为例子。我们可以使用 `attach` 方法简单地把 roles 附加给一个 user：

附加多对多模型

```
$user = User::find(1);

$user->roles()->attach(1);
```

也可以传入要存在枢纽表中的属性数组：

```
$user->roles()->attach(1, array('expires' => $expires));
```

当然，有 `attach` 方法就会有相反的 `detach` 方法：

```
$user->roles()->detach(1);
```

`attach` 和 `detach` 都可以接受ID数组作为参数：

```
$user = User::find(1);

$user->roles()->detach([1, 2, 3]);

$user->roles()->attach([1 => ['attribute1' => 'value1'], 2, 3]);
```

使用 **Sync** 方法同时附加一个以上多对多关联

您也可以使用 `sync` 方法附加关联模型。 `sync` 方法会把根据 ID 数组把关联存到枢纽表。附加完关联后，枢纽表里的模型只会关联到 ID 数组里的 id ：

```
$user->roles()->sync(array(1, 2, 3));
```

Sync 时在枢纽表加入额外数据

也可以在把每个 ID 加入枢纽表时，加入其他字段的数据：

```
$user->roles()->sync(array(1 => array('expires' => true)));
```

有时您可能想要使用一个命令，在建立新模型数据的同时附加关联。可以使用 `save` 方法达成目的：

```
$role = new Role(array('name' => 'Editor'));

User::find(1)->roles()->save($role);
```

上面的例子里，新的 `Role` 模型对象会在储存的同时关联到 `user` 模型。也可以传入属性数组把数据加到关联数据库表：

```
User::find(1)->roles()->save($role, array('expires' => $expires));
```

更新上层时间戳

当模型 `belongsTo` 另一个模型时，比方说一个 `Comment` 属于一个 `Post` ，如果能在子模型被更新时，更新上层的时间戳，这将会很有用。例如，当 `Comment` 模型更新时，您可能想要能够同时自动更新 `Post` 的 `updated_at` 时间戳。 Eloquent 让事情变得很简单。只要在子关联的类里，把关联方法名称加入 `touches` 属性即可：

```
class Comment extends Model {

    protected $touches = array('post');

    public function post()
    {
        return $this->belongsTo('App\Post');
    }

}
```

现在，当您更新 `Comment` 时，对应的 `Post` 会自动更新 `updated_at` 字段：

```
$comment = Comment::find(1);

$comment->text = 'Edit to this comment!';

$comment->save();
```

使用枢纽表

如您所知，要操作多对多关联需要一个中间的数据库表。 Eloquent 提供了一些有用的方法可以和这张表互动。例如，假设 `User` 对象关联到很多 `Role` 对象。取出这些关联对象时，我们可以在关联模型上取得 `pivot` 数据库表的数据：

```
$user = User::find(1);

foreach ($user->roles as $role)
{
    echo $role->pivot->created_at;
}
```

注意我们取出的每个 `Role` 模型对象会自动给一个 `pivot` 属性。这属性包含了枢纽表的模型数据，可以像一般的 Eloquent 模型一样使用。

默认 `pivot` 对象只会有关联键的属性。如果您想让 pivot 可以包含其他枢纽表的字段，可以在定义关联方法时指定那些字段：

```
return $this->belongsToMany('App\Role')->withPivot('foo', 'bar');
```

现在可以在 `Role` 模型的 `pivot` 对象上取得 `foo` 和 `bar` 属性了。

如果您想要可以自动维护枢纽表的 `created_at` 和 `updated_at` 时间戳，在定义关联方法时加上 `withTimestamps` 方法：

```
return $this->belongsToMany('App\Role')->withTimestamps();
```

删除枢纽表的关联数据

要删除模型在枢纽表的所有关联数据，可以使用 `detach` 方法：

```
User::find(1)->roles()->detach();
```

注意，如上的操作不会移除 `roles` 数据库表里面的数据，只会移除枢纽表里的关联数据。

更新枢纽表的数据

有时您只想更新枢纽表的数据，而没有要移除关联。如果您想更新枢纽表，可以像下面的例子使用 `updateExistingPivot` 方法：

```
User::find(1)->roles()->updateExistingPivot($roleId, $attributes);
```

自定义枢纽模型

Laravel 允许您自定义枢纽模型。要自定义模型，首先要建立一个继承 Eloquent 的「基本」模型类。在其他的 Eloquent 模型继承这个自定义的基本类，而不是默认的 Eloquent 。在基本模型类里，加入下面的方法返回自定义的枢纽模型实例：

```
public function newPivot(Model $parent, array $attributes, $table, $exists)
{
    return new YourCustomPivot($parent, $attributes, $table, $exists);
}
```

集合

所有 Eloquent 查询返回的数据，如果结果多于一条，不管是经由 `get` 方法或是 `relationship` ，都会转换成集合对象返回。这个对象实现了 `IteratorAggregate` PHP 接口，所以可以像数组一般进行遍历。而集合对象本身还拥有很多有用的方法可以操作模型数据。

确认集合中里是否包含特定键值

例如，我们可以使用 `contains` 方法，确认结果数据中，是否包含主键为特定值的对象。

```
$roles = User::find(1)->roles;

if ($roles->contains(2))
{
```

```
    //
}
```

集合也可以转换成数组或 JSON：

```
$roles = User::find(1)->roles->toArray();

$roles = User::find(1)->roles->toJson();
```

如果集合被转换成字符串类型，会返回 JSON 格式：

```
$roles = (string) User::find(1)->roles;
```

集合遍历

Eloquent 集合里包含了一些有用的方法可以进行循环或是进行过滤：

```
$roles = $user->roles->each(function($role)
{
    //
});
```

集合过滤

过滤集合时，回调函数的使用方式和 array_filter) 里一样。

```
$users = $users->filter(function($user)
{
    return $user->isAdmin();
});
```

> 注意： 如果要在过滤集合之后转成 JSON，转换之前先调用 `values` 方法重设数组的键值。

遍历传入集合里的每个对象到回调函数

```
$roles = User::find(1)->roles;

$roles->each(function($role)
{
    //
});
```

依照属性值排序

```
$roles = $roles->sortBy(function($role)
{
    return $role->created_at;
});
```

依照属性值排序

```
$roles = $roles->sortBy('created_at');
```

返回自定义的集合对象

有时您可能想要返回自定义的集合对象，让您可以在集合类里加入想要的方法。可以在 Eloquent 模型类里重写 `newCollection` 方法：

```
class User extends Model {

    public function newCollection(array $models = array())
    {
        return new CustomCollection($models);
    }

}
```

获取器和修改器

定义获取器

Eloquent 提供了一种便利的方法，可以在获取或设定属性时进行转换。要定义获取器，只要在模型里加入类似 `getFooAttribute` 的方法。注意方法名称应该使用驼峰式大小写命名，而对应的 database 字段名称是下划线分隔小写命名：

```
class User extends Model {

    public function getFirstNameAttribute($value)
    {
        return ucfirst($value);
    }

}
```

上面的例子中， first_name 字段设定了一个获取器。注意传入方法的参数是原本的字段数据。

定义修改器

修改器的定义方式也是类似的：

```
class User extends Model {

    public function setFirstNameAttribute($value)
    {
        $this->attributes['first_name'] = strtolower($value);
    }

}
```

日期转换器

默认 Eloquent 会把 `created_at` 和 `updated_at` 字段属性转换成 Carbon 实例，它提供了很多有用的方法，并继承了 PHP 原生的 `DateTime` 类。

您可以通过重写模型的 `getDates` 方法，自定义哪个字段可以被自动转换，或甚至完全关闭这个转换：

```
public function getDates()
{
    return array('created_at');
}
```

当字段是表示日期的时候，可以将值设为 UNIX timestamp 、日期字符串（ Y-m-d ）、 日期时间（ date-time ）字符串，当然还有 `DateTime` 或 `Carbon` 实例。

要完全关闭日期转换功能，只要从 `getDates` 方法返回空数组即可：

```
public function getDates()
{
    return array();
}
```

属性类型转换

如果您想要某些属性始终转换成另一个数据类型, 您可以在模型中增加 `casts` 属性。否则，您需要为每个属性定义修改器，这样会增加更多的时间开销。这里有一个使用 `casts` 属性的例子：

```
/**
 * 需要被转换成基本类型的属性值。
 *
 * @var array
 */
protected $casts = [
    'is_admin' => 'boolean',
];
```

现在当你获取 `is_admin` 属性时始终会是布尔类型，甚至在数据库中存储的这个值是一个整型也会被转换。其他支持的类型转换值有： `integer`, `real`, `float`, `double`, `string`, `boolean`, `object` 和 `array` 。

如果您存储的值是一个序列化的 JSON 时，那么 `array` 类型转换将会非常有用。比如，您的数据表里有一个 TEXT 类型的字段存储着序列化后的 JSON 数据， 通过增加 `array` 类型转换, 当获取这个属性的时候会自动反序列化成 PHP 的数组：

```
/**
 * 需要被转换成基本类型的属性值。
 *
 * @var array
 */
```

```
protected $casts = [
    'options' => 'array',
];
```

现在，当你使用 Eloquent 模型时：

```
$user = User::find(1);

// $options 是一个数组...
$options = $user->options;

// options 会自动序列化成 JSON...
$user->options = ['foo' => 'bar'];
```

模型事件

Eloquent 模型有很多事件可以触发，让您可以在模型操作的生命周期的不同时间点，使用下列方法绑定事件： `creating`，`created`，`updating`，`updated`，`saving`，`saved`，`deleting`，`deleted`，`restoring`，`restored`。

当一个对象初次被储存到数据库， `creating` 和 `created` 事件会被触发。如果不是新对象而调用了 `save` 方法， `updating` / `updated` 事件会被触发。而两者的 `saving` / `saved` 事件都会被触发。

使用事件取消数据库操作

如果 `creating` 、 `updating` 、 `saving` 、 `deleting` 事件返回 false 的话，就会取消数据库操作

```
User::creating(function($user)
{
    if ( ! $user->isValid()) return false;
});
```

注册事件监听者的方式

您可以在 `EventServiceProvider` 中注册您的模型事件绑定。比如：

```
/**
 * Register any other events for your application.
 *
 * @param  \Illuminate\Contracts\Events\Dispatcher  $events
 * @return void
 */
public function boot(DispatcherContract $events)
{
    parent::boot($events);

    User::creating(function($user)
    {
        //
    });
}
```

模型观察者

要整合模型的事件处理，可以注册一个模型观察者。观察者类里要设定对应模型事件的方法。例如，观察者类里可能有 `creating`、 `updating` 、 `saving` 方法，还有其他对应模型事件名称的方法：

例如，一个模型观察者类可能看起来如下：

```
class UserObserver {

    public function saving($model)
    {
        //
    }

    public function saved($model)
    {
        //
    }

}
```

可以使用 `observe` 方法注册一个观察者实例：

```
User::observe(new UserObserver);
```

模型 URL 生成

当你把一个模型实例传递给 `route` 或者 `action` 方法时，模型的主键会被插入到生成的 URI 中。比如：

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

action('UserController@show', [$user]);
```

在这个例子中 `$user->id` 属性会被插入到生成的 URL 的 `{user}` 这个占位符中。不过，如果你想使用其他的属性而不是 ID 的话，你可以覆盖模型的 `getRouteKey` 方法：

```
public function getRouteKey()
{
    return $this->slug;
}
```

转换成数组 / JSON

将模型数据转成数组

当构建 JSON API 时，您可能常常需要把模型和关联对象转换成数组或JSON。所以Eloquent里已经包含了这些方法。要把模型和已载入的关联对象转成数组，可以使用 `toArray` 方法：

```
$user = User::with('roles')->first();

return $user->toArray();
```

注意也可以把整个的模型集合转换成数组：

```
return User::all()->toArray();
```

将模型转换成 JSON

要把模型转换成 JSON，可以使用 `toJson` 方法：

```
return User::find(1)->toJson();
```

从路由中返回模型

注意当模型或集合被转换成字符串类型时会自动转换成 JSON 格式，这意味着您可以直接从路由返回 Eloquent 对象！

```
Route::get('users', function()
{
    return User::all();
});
```

转换成数组或 JSON 时隐藏属性

有时您可能想要限制能出现在数组或 JSON 格式的属性数据，比如密码字段。只要在模型里增加 `hidden` 属性即可

```
class User extends Model {

    protected $hidden = array('password');

}
```

> 注意： 要隐藏关联数据，要使用关联的方法名称，而不是动态获取的属性名称。

此外，可以使用 `visible` 属性定义白名单：

```
protected $visible = array('first_name', 'last_name');
```

有时候您可能想要增加不存在数据库字段的属性数据。这时候只要定义一个获取器即可：

```
public function getIsAdminAttribute()
{
    return $this->attributes['admin'] == 'yes';
}
```

定义好获取器之后，再把对应的属性名称加到模型里的 `appends` 属性：

```
protected $appends = array('is_admin');
```

把属性加到 `appends` 数组之后，在模型数据转换成数组或 JSON 格式时就会有对应的值。在 `appends` 数组中定义的值同样遵循模型中 `visible` 和 `hidden` 的设定。

结构生成器

介绍

Laravel 的结构生成器 (`Schema`) 提供一个与数据库无关的数据表产生方法，它可以很好的处理 Laravel 支持的各种数据库类型，并且在不同系统间提供一致性的 API 操作。

建立与删除数据表

要建立一个新的数据表，可使用 `Schema::create` 方法：

```
Schema::create('users', function($table)
{
    $table->increments('id');
});
```

传入 `create` 方法的第一个参数是数据表名称，第二个参数是 `Closure` 并接收 `Blueprint` 对象被用来定义新的数据表。

要修改数据表名称，可使用 `rename` 方法：

```
Schema::rename($from, $to);
```

要指定特定连接来操作，可使用 `Schema::connection` 方法：

```
Schema::connection('foo')->create('users', function($table)
{
    $table->increments('id');
});
```

要移除数据表，可使用 `Schema::drop` 方法：

```
Schema::drop('users');

Schema::dropIfExists('users');
```

加入字段

更新现有的数据表，可使用 `Schema::table` 方法：

```
Schema::table('users', function($table)
{
    $table->string('email');
});
```

数据表产生器提供多种字段类型可使用，在您建立数据表时也许会用到：

命令	功能描述
`$table->bigIncrements('id');`	ID 自动增量，使用相当于「big integer」类型
`$table->bigInteger('votes');`	相当于 BIGINT 类型
`$table->binary('data');`	相当于 BLOB 类型
`$table->boolean('confirmed');`	相当于 BOOLEAN 类型
`$table->char('name', 4);`	相当于 CHAR 类型，并带有长度
`$table->date('created_at');`	相当于 DATE 类型
`$table->dateTime('created_at');`	相当于 DATETIME 类型
`$table->decimal('amount', 5, 2);`	相当于 DECIMAL 类型，并带有精度与基数
`$table->double('column', 15, 8);`	相当于 DOUBLE 类型，总共有 15 位数，在小数点后面有 8 位数
`$table->enum('choices', array('foo', 'bar'));`	相当于 ENUM 类型
`$table->float('amount');`	相当于 FLOAT 类型
`$table->increments('id');`	相当于 Incrementing 类型 (数据表主键)
`$table->integer('votes');`	相当于 INTEGER 类型
`$table->json('options');`	相当于 JSON 类型
`$table->longText('description');`	相当于 LONGTEXT 类型
`$table->mediumInteger('numbers');`	相当于 MEDIUMINT 类型
`$table->mediumText('description');`	相当于 MEDIUMTEXT 类型
`$table->morphs('taggable');`	加入整数 `taggable_id` 与字串 `taggable_type`
`$table->nullableTimestamps();`	与 `timestamps()` 相同，但允许 NULL
`$table->smallInteger('votes');`	相当于 SMALLINT 类型
`$table->tinyInteger('numbers');`	相当于 TINYINT 类型
`$table->softDeletes();`	加入 **deleted_at** 字段于软删除使用
`$table->string('email');`	相当于 VARCHAR 类型
`$table->string('name', 100);`	相当于 VARCHAR 类型，并指定长度

`$table->text('description');`	相当于 TEXT 类型
`$table->time('sunrise');`	相当于 TIME 类型
`$table->timestamp('added_on');`	相当于 TIMESTAMP 类型
`$table->timestamps();`	加入 **created_at** 和 **updated_at** 字段
`$table->rememberToken();`	加入 `remember_token` 使用 VARCHAR(100) NULL
`->nullable()`	标示此字段允许 NULL
`->default($value)`	声明此字段的默认值
`->unsigned()`	配置整数是无分正负

在 **MySQL** 使用 **After** 方法

若您使用 MySQL 数据库，您可以使用 `after` 方法来指定字段的顺序：

```
$table->string('name')->after('email');
```

修改字段

有时候您需要修改一个存在的字段，例如：您可能想增加保存文本字段的长度。通过 `change` 方法让这件事情变得非常容易！假设我们想要将字段 `name` 的长度从 25 增加到 50 的时候：

```
Schema::table('users', function($table)
{
    $table->string('name', 50)->change();
});
```

另外也能将某个字段修改为允许 NULL：

```
Schema::table('users', function($table)
{
    $table->string('name', 50)->nullable()->change();
});
```

修改字段名称

要修改字段名称，可在结构生成器内使用 `renameColumn` 方法，请确认在修改前 `composer.json` 文件内已经加入 `doctrine/dbal`。

```
Schema::table('users', function($table)
{
    $table->renameColumn('from', 'to');
});
```

> 注意: 目前暂时还不支持修改表中的结构为 `enum` 字段类型。

移除字段

要移除字段，可在结构生成器内使用 `dropColumn` 方法，请确认在移除前 `composer.json` 文件内已经加入 `doctrine/dbal`。

移除数据表字段

```
Schema::table('users', function($table)
{
    $table->dropColumn('votes');
});
```

移除数据表多个字段

```
Schema::table('users', function($table)
{
    $table->dropColumn(array('votes', 'avatar', 'location'));
});
```

检查是否存在

检查数据表是否存在

您可以轻松的检查数据表或字段是否存在，使用 `hasTable` 和 `hasColumn` 方法：

```
if (Schema::hasTable('users'))
{
    //
}
```

检查字段是否存在

```
if (Schema::hasColumn('users', 'email'))
{
    //
}
```

加入索引

结构生成器支持多种索引类型，有两种方法可以加入，方法一，您可以在定义字段时顺便附加上去，或者是分开另外加入：

```
$table->string('email')->unique();
```

或者，您可以独立一行来加入索引，以下是支持的索引类型：

命令	功能描述
`$table->primary('id');`	加入主键 (primary key)

`$table->primary(array('first', 'last'));`	加入复合键 (composite keys)
`$table->unique('email');`	加入唯一索引 (unique index)
`$table->index('state');`	加入基本索引 (index)

外键

Laravel 也支持数据表的外键约束：

```
$table->integer('user_id')->unsigned();
$table->foreign('user_id')->references('id')->on('users');
```

例子中，我们关注字段 `user_id` 参照到 `users` 数据表的 `id` 字段。请先确认已经建立外键！

您也可以指定选择在「on delete」和「on update」进行约束动作：

```
$table->foreign('user_id')
      ->references('id')->on('users')
      ->onDelete('cascade');
```

要移除外键，可使用 `dropForeign` 方法。外键的命名约定如同其他索引：

```
$table->dropForeign('posts_user_id_foreign');
```

> 注意: 当外键有参照到自动增量时，记得配置外键为 `unsigned` 类型。

移除索引

要移除索引您必须指定索引名称，Laravel 默认有脉络可循的索引名称。简单地链接这些数据表与索引的字段名称和类型。举例如下：

命令	功能描述
`$table->dropPrimary('users_id_primary');`	从「users」数据表移除主键
`$table->dropUnique('users_email_unique');`	从「users」数据表移除唯一索引
`$table->dropIndex('geo_state_index');`	从「geo」数据表移除基本索引

移除时间戳记和软删除

要移除 `timestamps`、`nullableTimestamps` 或 `softDeletes` 字段类型，您可以使用以下方法：

命令	功能描述
`$table->dropTimestamps();`	移除 **created_at** 和 **updated_at** 字段
`$table->dropSoftDeletes();`	移除 **deleted_at** 字段

保存引擎

要配置数据表的保存引擎，可在结构生成器配置 `engine` 属性：

```
Schema::create('users', function($table)
{
    $table->engine = 'InnoDB';

    $table->string('email');
});
```

迁移和数据填充

介绍

迁移是一种数据库的版本控制。可以让团队在修改数据库结构的同时，保持彼此的进度一致。迁移通常会和 结构生成器 一起使用，可以简单的管理数据库结构。

建立迁移文件

使用 Artisan CLI 的 `make:migrate` 命令建立迁移文件：

```
php artisan make:migration create_users_table
```

迁移文件会建立在 `database/migrations` 目录下，文件名会包含时间戳记，在执行迁移时用来决定顺序。

你也可以在建立迁移命令加上 `--path` 参数。路径要相对于应用程序所在的根目录。

```
php artisan make:migration foo --path=app/migrations
```

`--table` 和 `--create` 参数可以用来指定数据表名称，以及迁移文件是否要建立新的数据表。

```
php artisan make:migration add_votes_to_users_table --table=users

php artisan make:migration create_users_table --create=users
```

执行迁移

执行所有未执行的迁移

```
php artisan migrate
```

> 注意: 如果在执行迁移时发生「class not found」错误，试着先执行 `composer dump-autoload` 命令后再进行一次。

在线上环境 (Production) 中强制执行迁移

有些迁移操作是具有破坏性的，意味着可能让你遗失原本保存的数据。为了防止你在上线环境执行到这些迁移命令，你会被提示要在执行迁移前进行确认。加上 `--force` 参数执行强制迁移：

```
php artisan migrate --force
```

回滚迁移

回滚上一次的迁移

```
php artisan migrate:rollback
```

回滚所有迁移

```
php artisan migrate:reset
```

回滚所有迁移并且再执行一次

```
php artisan migrate:refresh

php artisan migrate:refresh --seed
```

数据填充

Laravel 可以简单的使用 seed 类，填充测试数据到数据库。所有的 seed 类放在 `database/seeds` 目录下。可以使用任何你想要的类名称，但是应该遵守某些大小写规范，如 `UserTableSeeder` 之类。默认已经有一个 `DatabaseSeeder` 类。在这个类里，使用 `call` 方法执行其他的 seed 类，让你控制填充的顺序。

Seed 类例子

```
class DatabaseSeeder extends Seeder {

    public function run()
    {
        $this->call('UserTableSeeder');

        $this->command->info('User table seeded!');
    }

}

class UserTableSeeder extends Seeder {

    public function run()
    {
        DB::table('users')->delete();

        User::create(array('email' => 'foo@bar.com'));
```

```
    }

}
```

要执行数据填充，可以使用 Artisan CLI 的 `db:seed` 命令：

```
php artisan db:seed
```

默认 `db:seed` 命令会执行 `DatabaseSeeder` ，可以使用它来调用其他 seed 类，不过，也可以使用 `--class` 参数指定要单独执行的类：

```
php artisan db:seed --class=UserTableSeeder
```

你也可以使用 `migrate:refresh` 命令填充数据，它会回滚并且再次执行所有迁移：

```
php artisan migrate:refresh --seed
```

Redis

介绍

Redis 是开源，先进的键值对保存库。由于它可用的键包含了 字串、哈希、列表、集合 和 有序集合，因此常被称作数据结构服务器。

在使用 Redis 之前，你需要经由 Composer 将 `predis/predis` 扩展包装在 Laravel 中。

> 提醒： 如果你用 PECL 安装了 Redis PHP extension，则需要重命名 `config/app.php` 里的 Redis 别名。

配置文件

应用程序的 Redis 配置文件在 `config/database.php`。在这个文件里，你会看到 **redis** 数组，里面有应用程序使用的 Redis 服务器数据：

```
'redis' => array(

    'cluster' => true,

    'default' => array('host' => '127.0.0.1', 'port' => 6379),

),
```

默认的服务器配置对于开发应该是足够的。然而，你可以根据使用环境自由修改数组数据。只要给每个 Redis 一个名称，并且配置服务器的 host 和 port。

`cluster` 选项会让 Laravel 的 Redis 客户端在所有 Redis 节点间执行客户端分片（ client-side sharding ），让你建立节点池，并因此拥有大量的 RAM 可用。然而，客户端分片的节点不能执行容错转移；因此，这主要适合用可以从另一台主要数据保存库取得的缓存数据。

如果你的 Redis 服务器需要认证，你可以在 Redis 服务器配置文件里加入 `password` 为键值的参数配置。

使用方式

你可以经由 `Redis::connection` 方法得到 Redis 实例：

```
$redis = Redis::connection();
```

你会得到一个使用 Redis 默认服务器的实例。如果你没有使用服务器集群，你可以在 `connection` 方法传入定义在 Redis 配置文件的服务器名称，以连到特定服务器：

```
$redis = Redis::connection('other');
```

一旦你有了 Redis 客户端实例，就可以使用实例发出任何 Redis 命令。Laravel 使用魔术方法传递命令到服务器：

```
$redis->set('name', 'Taylor');

$name = $redis->get('name');

$values = $redis->lrange('names', 5, 10);
```

注意，传入命令的参数仅只是传递到魔术方法里。当然，你不一定要使用魔术方法，你也可以使用 `command` 方法传递命令到服务器：

```
$values = $redis->command('lrange', array(5, 10));
```

若你只想对默认服务器下命令，可以使用 `Redis` 类的静态魔术方法：

```
Redis::set('name', 'Taylor');

$name = Redis::get('name');

$values = Redis::lrange('names', 5, 10);
```

> 提示： 也可以使用 Redis 作为 Laravel 的 缓存 和 会话 驱动。

管道

当你想要一次发送很多命令到服务器时可以使用管道。使用 `pipeline` 方法：

发送多个命令到服务器

```
Redis::pipeline(function($pipe)
{
    for ($i = 0; $i < 1000; $i++)
    {
        $pipe->set("key:$i", $i);
    }
});
```

Artisan 命令行接口

介绍

Artisan 是 Laravel 内置的命令行接口。它提供了一些有用的命令协助您开发，它是由强大的 Symfony Console 组件所驱动。

用法

列出所有可用的命令

要查看所有可以使用的 Artisan 命令，你可以使用 `list` 命令：

```
php artisan list
```

浏览命令的帮助画面

每个命令都包含一个显示并描述这个命令能够接受哪些参数和选项的「帮助画面」。要浏览帮助画面，只需要在命令名称前面加上 `help` 即可：

```
php artisan help migrate
```

指定环境配置

您可以指定要使用的环境配置，只要在执行命令时加上 `--env` 即可切换：

```
php artisan migrate --env=local
```

显示目前的 **Laravel** 版本

你也可以使用 `--version` 选项，查看目前安装的 Laravel 版本：

```
php artisan --version
```

在命令行接口以外的地方调用命令

有时你会希望在命令行接口以外的地方执行 Artisan 命令。例如，你可能会希望从 HTTP 路由调用 Artisan 命令。只要使用 `Artisan` facade 即可：

```
Route::get('/foo', function()
{
    $exitCode = Artisan::call('command:name', ['--option' => 'foo']);

    //
});
```

你甚至可以把 Artisan 命令放到队列，他们会通过 队列工作者 在后台执行：

```
Route::get('/foo', function()
{
    Artisan::queue('command:name', ['--option' => 'foo']);

    //
});
```

定时调用 **Artisan** 命令

过去，开发者会对每个他们想要调用的命令行指令建立 Cron 对象。然而，这很令人头痛。你的命令行指令调用不再包含在版本控制里面，并且你必须 SSH 进入你的服务器以添加 Cron 对象。让我们来让生活变得更轻松。Laravel 命令调用器允许你顺畅地且语义化地定义命令调用在 Laravel 里面，而且你的服务器只需要一个 Cron 对象。

你的命令调用保存在 `app/Console/Kernel.php` 文件。你会在这个类里看到一个 `schedule` 方法。为了帮助您开始，方法里面包含一个简单的例子。你可以依照你需要的自由地添加任何数量的预定工作到 `Schedule` 对象。你只需要添加这个 Cron 对象到服务器：

```
* * * * * php /path/to/artisan schedule:run 1>> /dev/null 2>&1
```

这个 Cron 将会每分钟调用 Laravel 命令调用器。接着，Laravel 评估你的预定工作并在时间到时执行工作。这不能再更简单了！

更多调用的例子

让我们来多看几个调用的例子：

调用闭包

```
$schedule->call(function()
{
    // 执行一些任务...

})->hourly();
```

调用终端机命令

```
$schedule->exec('composer self-update')->daily();
```

自己配置 Cron 表达式

```
$schedule->command('foo')->cron('* * * * *');
```

频繁的工作

```
$schedule->command('foo')->everyFiveMinutes();

$schedule->command('foo')->everyTenMinutes();

$schedule->command('foo')->everyThirtyMinutes();
```

每天一次的工作

```
$schedule->command('foo')->daily();
```

每天一次在特定时间 (24 小时制) 的工作

```
$schedule->command('foo')->dailyAt('15:00');
```

每天两次的工作

```
$schedule->command('foo')->twiceDaily();
```

每个工作日执行的工作

```
$schedule->command('foo')->weekdays();
```

每周一次的工作

```
$schedule->command('foo')->weekly();

// 调用每周一次在特定的日子 (0-6) 和时间的工作...
$schedule->command('foo')->weeklyOn(1, '8:00');
```

每月一次的工作

```
$schedule->command('foo')->monthly();
```

特定日期的工作

```
$schedule->command('foo')->mondays();
$schedule->command('foo')->tuesdays();
$schedule->command('foo')->wednesdays();
```

```
$schedule->command('foo')->thursdays();
$schedule->command('foo')->fridays();
$schedule->command('foo')->saturdays();
$schedule->command('foo')->sundays();
```

限制应该执行工作的环境

```
$schedule->command('foo')->monthly()->environments('production');
```

指定工作在当应用程序处于维护模式也应该执行

```
$schedule->command('foo')->monthly()->evenInMaintenanceMode();
```

只允许工作在闭包返回 **true** 的时候执行

```
$schedule->command('foo')->monthly()->when(function()
{
    return true;
});
```

将预定工作的输出发送到指定的 **E-mail**

```
$schedule->command('foo')->sendOutputTo($filePath)->emailOutputTo('foo@example.com');
```

> 注意**:** 你必须先把输出存到文件中才可以发送 email。

将预定工作的输出发送到指定的路径

```
$schedule->command('foo')->sendOutputTo($filePath);
```

在预定工作执行之后 **Ping** 一个给定的 **URL**

```
$schedule->command('foo')->thenPing($url);
```

Artisan 开发

简介

除了 Artisan 本身提供的命令之外，您也可以为您的应用程序建立属于你自己的命令。你可以将自定义命令存放在 `app/Console/commands` 目录底下；然而，您也可以任意选择存放位置，只要您的命令能够被 `composer.json` 自动加载。

建立自定义命令

自动创建类（Class）

要创建一个新的自定义命令，您可以使用 `make:console` 这个 Artisan 命令，这将会自动产生一个 Command stub 协助您开始创建您的自定义命令：

自动创建一个新的命令类

```
php artisan make:console FooCommand
```

上面的命令将会协助你自动创建一个类，并保存为文件 `app/Console/FooCommand.php`。

在创建自定义命令时，加上 `--command` 这个选项，将可以指定之后在终端机使用此自定义命令时，所要输入的自定义命令名称：

```
php artisan make:console AssignUsers --command=users:assign
```

撰写自定义命令

一旦你的自定义命令被创建后，你需要填写自定义命令的 `名称（name）` 与 `描述（description）`，您所填写的内容将会被显示在 Artisan 的 `list` 画面中。

当您的自定义命令被执行时，将会调用 `fire` 方法，您可以在此为自定义命令加入任何的逻辑判断。

参数与选项

你可以通过 `getArguments` 与 `getOptions` 为自定义命令自行定义任何需要的参数与选项。这两个方法都会返回一组命令数组，并由选项数组的清单所组成。

当定义 `arguments` 时，该数组值的定义分别如下：

```
array($name, $mode, $description, $defaultValue)
```

参数 `mode` 可以是下列其中一项： `InputArgument::REQUIRED` 或 `InputArgument::OPTIONAL`。

当定义 `options` 时，该数组值的定义分别如下：

```
array($name, $shortcut, $mode, $description, $defaultValue)
```

对选项而言，参数 `mode` 可以是下列其中一项： `InputOption::VALUE_REQUIRED`, `InputOption::VALUE_OPTIONAL`, `InputOption::VALUE_IS_ARRAY`, `InputOption::VALUE_NONE`。

模式为 `VALUE_IS_ARRAY` 表示调用命令时可以多次使用此选项来传入多个值：

```
php artisan foo --option=bar --option=baz
```

模式为 `VALUE_NONE` 则表示将此选项纯粹作为一种有或无的「开关」使用：

```
php artisan foo --option
```

取得输入值（参数与选项）

当您的自定义命令执行时，您需要让您的应用程序可以访问到这些参数和选项的值，要做到这一点，您可以使用 `argument` 和 `option` 方法：

取得自定义命令被输入的参数

```
$value = $this->argument('name');
```

取得自定义命令被输入的所有参数

```
$arguments = $this->argument();
```

取得自定义命令被输入的选项

```
$value = $this->option('name');
```

取得自定义命令被输入的所有选项

```
$options = $this->option();
```

产生输出

想要显示信息到终端屏幕上，您可以使用 `info`、 `comment`、 `question` 和 `error` 方法。每一种方法将会依据它所代表的目的，分别对应一种适当的 ANSI 颜色。

显示一般消息到终端屏幕

```
$this->info('Display this on the screen');
```

显示错误消息到终端屏幕

```
$this->error('Something went wrong!');
```

询问式输入

您也可以使用 `ask` 和 `confirm` 方法来提示用户进行输入：

提示用户进行输入

```
$name = $this->ask('What is your name?');
```

提示用户进行加密输入

```
$password = $this->secret('What is the password?');
```

提示用户进行确认

```
if ($this->confirm('Do you wish to continue? [yes|no]'))
{
    //
}
```

您也可以指定一个默认值给 `confirm` 方法，可以是 `true` 或 `false`：

```
$this->confirm($question, true);
```

调用其它命令

有时候您可能希望在您的命令内部调用其它命令，此时您可以使用 `call` 方法：

```
$this->call('command:name', ['argument' => 'foo', '--option' => 'bar']);
```

注册自定义命令

注册一个 **Artisan** 命令

一旦你的自定义命令撰写完成后，你需要将它注册于 Artisan 它才能被使用。这通常位于 `app/Console/Kernel.php` 这个文件中。在此文件的 `commands` 属性，你会找到一份命令的清单。若要注

册你的自定义命令，很简单的你只要将它加入清单中。当 Artisan 启动时，被列于此属性中的所有命令都将被 service container 解析，并且被注册于 Artisan 。