一、序言

发行版本说明

1、支持政策

对于 LTS 版本，比如 Laravel 5.1，我们将会提供为期两年的 bug 修复和三年的安全修复支持。LTS 版本将会提供最长时间的支持和维护。
对于其他通用版本，只提供六个月的 bug 修复和一年的安全修复支持，比如 Laravel 5.2。

2、Laravel 5.2

Laravel 5.2 在 5.1 基础上继续改进和优化，添加了许多新的功能特性：多认证驱动支持、隐式模型绑定、简化 Eloquent 全局作用域、可选择的认证脚手架、中间件组、访问频率限制、数组输入验证优化等等。

多认证驱动

在之前的 Laravel 版本中，框架只支持默认的、基于 session 的认证驱动，且在单个应用中只能拥有一个认证模型类（对应单张表），这为我们实现某型功能，比如前后端分离登录带来麻烦。
我们对此进行了改进，在 Laravel 5.2 中，你可以定义多个认证驱动，还有多个认证模型以及用户表，并且可以独立控制其认证处理（登录、注册、密码重置）。例如，如果你的应用包含一个后台管理员用户表和一个前台学生用户表，现在你可以使用 `Auth` 门面来实现后台用户和学生用户的独立登录而不相互影响。

认证脚手架

通过多认证驱动，Laravel 可以轻松处理后台用户认证；此外，Laravel 5.2 还提供了便捷的方式来创建前台认证视图，只需在终端执行如下 Artisan 命令即可：

```
php artisan make:auth
```

该命令会生成纯文本的、兼容 Bootstrap 样式的视图用于登录、注册和密码重置。该命令还会使用相应路由更新路由文件。
注意：该功能特性只能在新应用中使用，不能再应用升级过程中使用。

隐式模型绑定

隐式模型绑定使得在路由或控制器中直接注入相应模型实例更加便捷。假设你有一个路由定义如下：

```
use App\User;


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

在 Laravel 5.1 中，你需要通过 `Route::model` 方法告诉 Laravel 注入 `App\User` 实例以匹配路由定义中的 `{user}` 参数。
现在，在 Laravel 5.2 中，框架将会基于相应 URI 片段自动注入模型，从而允许你快速访问需要的模型实例。

如果路由参数片段 `{user}` 匹配路由闭包或控制器方法中相应变量 `$user`，并且被类型声明为一个 Eloquent 模型类的话，Laravel 将会自动注入该模型。
更多隐式模型绑定详情请查看 Laravel 5.2 文档 HTTP 路由模型绑定部分。

中间件组

中间件组允许你通过单个方便的键来对相关路由中间件进行分组，从而为某个路由一次指定多个中间件。例如，在同一个应用中构建 Web UI 或 API 时这一特性很有用，你可以将 session 和 csrf 路由分组到一个 `web` 组，或者将访问频率限制分组到 `api` 中。
实际上，默认的 Laravel 5.2 应用结构采用的正是这个方法。例如，在默认的 `App\Http\Kernel.php` 文件中你会看到如下内容：

```
/**
 * The application's route middleware groups.
 *
 * @var array
 */
protected $middlewareGroups = [
    'web' => [
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \App\Http\Middleware\VerifyCsrfToken::class,
    ],
```

```
    'api' => [
        'throttle:60,1',
    ],
];
```

然后，`web` 组像这样分配给路由：

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

访问频率限制

一个新的访问频率限制中间件已经被内置到框架中，从而允许你轻松限制给定 IP 地址在指定时间内对某个路由发起请求的数目。例如，要限制某个 IP 地址每分钟只能访问某个路由 60 次，你可以这么做：

```
Route::get('/api/users', ['middleware' => 'throttle:60,1', function () {
    //
}]);
```

数组输入验证

在 Laravel 5.2 可轻松实现表单字段的数组输入验证。例如，要验证给定数组输入字段中的每一个 `email` 是唯一的，可以这么实现：

```
$validator = Validator::make($request->all(), [
    'person.*.email' => 'email|unique:users'
]);
```

同样，你可以在语言文件中使用 `*` 来指定验证数组字段：

```
'custom' => [
    'person.*.email' => [
        'unique' => 'Each person must have a unique e-mail address',
    ]
],
```

Eloquent 全局作用域优化

在之前的 Laravel 版本，Eloquent 全局作用域的实现是复杂且容易出错的，但在 Laravel 5.2 中，全局查询作用域只需实现一个简单的方法 apply 即可。

关于全局作用域详情请查看全局作用域文档。

升级指南

从 5.1 升级到 5.2.0

更新依赖

更新 `composer.json` 文件指向 `laravel/framework 5.2.*`。
注：如果你安装的是 Laravel 5.2 的 beta 版本，还要添加 `"minimum-stability": "beta"` 到 `composer.json` 文件。
添加 `symfony/dom-crawler ~3.0` 和 `symfony/css-selector ~3.0` 到 `composer.json` 的 `require-dev` 部分。

认证

配置文件

更新 `config/auth.php` 文件内容如下：
https://github.com/laravel/laravel/blob/develop/config/auth.php
更新完成后，基于原来的配置设置认证选项，如果不做改动，认证服务将基于 Laravel 5.1。

在新的 `auth.php` 配置文件中，要特别注意 `passwords.users.email` 配置项，由于在 Laravel 5.2 对 `email` 视图路径有所改动，因此要确保该视图路径与应用实际的路径相匹配，如果不匹配的话要更新该配置值。

Contracts

如果你实现了 `Illuminate\Contracts\Auth\Authenticatable` 契约但没有使用 `Authenticatable` trait，那么需要添加一个新的 `getAuthIdentifierName` 方法到该契约实现类。通常，该方法返回认证实体的主键字段名，如：`id`。
这对你的应用没有什么影响，除非你手动实现了 `Illuminate\Contracts\Auth\Authenticatable`。

自定义驱动

如果你使用了 `Auth::extend` 方法自定义获取用户的方法，现在需要使用 `Auth::provider` 来自定义用户提供者。一旦你自定义了提供者，就要在新的 `auth.php` 配置文件中的 `providers` 数组中配置该提供者。
更多自定义认证提供者详情，请查看其对应文档。

授权

`Illuminate\Auth\Access\UnauthorizedException` 被重命名为 `Illuminate\Auth\Access\AuthorizationException`。如果你没有手动捕获该异常那么这一改变对之前代码没有什么影响。

集合

Eloquent 集合基类

调用 Eloquent 集合实例的 `pluck`, `keys`, `zip`, `collapse`, `flatten`, `flip` 方法现在会返回集合基类。

保留键名

`slice`、`chunk` 和 `reverse` 方法现在会保留集合的键名，如果你不想这些方法保留键名，使用集合实例的 `values` 方法即可。

Composer 类

`Illuminate\Foundation\Support\Composer` 类现在被移动到 `Illuminate\Support\Composer`，如果你没有在代码中使用该类那么这一改变对程序没有影响。

命令和处理器

自处理命令

在创建任务/命令时你不再需要实现 `SelfHandling` 契约，所有任务现在默认都是自处理的，因此你可以在自己的类中移除该接口。

独立的命令&处理器

Laravel 5.2 命令现在只支持自处理命令，不再支持独立的命令和处理器。

如果你想要继续使用独立的命令和处理器，可以安装提供向后兼容支持的 Laravel Collective 包：https://github.com/LaravelCollective/bus

配置

开发环境

添加一个 `env` 配置项到配置文件 `app.php` 中：

```
'env' => env('APP_ENV', 'production'),
```

缓存和环境

如果你在开发过程中使用 `config:cache` 命令，必须保证只是在配置文件中调用了 `env` 函数，而不是在应用程序的其它地方。

如果你在应用程序中调用了 `env` 函数，强烈建议添加适当的配置值到配置文件，然后在该位置调用 `env`，从而允许你将 `env` 调用改为 `config` 调用。

CSRF 验证

在单元测试中不再支持自动进行 CSRF 验证，当然这一改变对你的应用程序代码没什么影响。

Elixir

PHP 的 `elixir` 方法现在返回一个完整 URL 而不是相对 URL，这对应用程序没有什么影响，除非你曾经手动将这些 URL 转化成完整 URL。

Eloquent

日期转化

当调用模型或模型集合的 `toArray` 方法时，任何添加到 `$casts` 的属性，
如 `date` 或 `datetime`，现在都会被转化为字符串。这使得在 `$dates` 数组中制定的日期转化变得简单方便。

全局作用域

我们重写了全局作用域的实现以便于使用，全局作用域不再需要 `remove` 方法，因此可以在所有你使用到该方法的地方将其移除。
如果你曾经在 Eloquent 查询构建器上调用过了 `getQuery` 方法以获取底层查询构建器实例，现在应该改为调用 `toBase` 方法。
如果你因为某种原因直接调用了 `remove` 方法，需要将其改成 `$eloquentBuilder->withoutGlobalScope($scope)` 这种方式来调用。
在 Eloquent 查询构建器中新增了 `withoutGlobalScope` 和 `withoutGlobalScopes` 方法，任何调用 `$model->removeGlobalScopes($builder)` 的地方现在都要改成 `$builder->withoutGlobalScopes()`。

事件

核心事件对象

Laravel 的一些核心事件触发现在使用事件对象取代之前的事件名称以及动态参数，下面是原来的事件名称与现在的事件对象对应关系：

Old	New
`auth.attempting`	`Illuminate\Auth\Events\Attempting`
`auth.login`	`Illuminate\Auth\Events\Login`
`auth.logout`	`Illuminate\Auth\Events\Logout`
`cache.missed`	`Illuminate\Cache\Events\CacheMissed`
`cache.hit`	`Illuminate\Cache\Events\CacheHit`
`cache.write`	`Illuminate\Cache\Events\KeyWritten`
`cache.delete`	`Illuminate\Cache\Events\KeyForgotten`
`connection.{name}.beginTransaction`	`Illuminate\Database\Events\TransactionBeginning`

Old	New
connection.{name}.committed	Illuminate\Database\Events\TransactionCommitted
connection.{name}.rollingBack	Illuminate\Database\Events\TransactionRolledBack
illuminate.query	Illuminate\Database\Events\QueryExecuted
illuminate.queue.after	Illuminate\Queue\Events\JobProcessed
illuminate.queue.failed	Illuminate\Queue\Events\JobFailed
illuminate.queue.stopping	Illuminate\Queue\Events\WorkerStopping
mailer.sending	Illuminate\Mail\Events\MessageSending
router.matched	Illuminate\Routing\Events\RouteMatched

这些事件对象传入参数和 Laravel 5.1 的事件处理器一样，例如，如果你在 Laravel 5.1 中使用了 DB:listen 事件，在 5.2 中更新代码如下：

```
DB::listen(function ($event) {

    dump($event->sql);

    dump($event->bindings);

});
```

你可以去检查每个事件对象类去查看它们的公有属性。

异常处理

App\Exceptions\Handler 类的 $dontReport 属性应该被更新为至少包含以下异常类型的其中一个：

```
use Illuminate\Auth\Access\AuthorizationException;

use Illuminate\Database\Eloquent\ModelNotFoundException;

use Symfony\Component\HttpKernel\Exception\HttpException;

use Illuminate\Foundation\Validation\ValidationException;


/**

 * A list of the exception types that should not be reported.
```

```
 *
 * @var array
 */
protected $dontReport = [
    AuthorizationException::class,
    HttpException::class,
    ModelNotFoundException::class,
    ValidationException::class,
];
```

隐式模型绑定

Laravel 5.2 支持“隐式模型绑定”，以便在路由和控制器中基于 URI 标识符自动注入模型实例。然而，这也改变了路由和控制器中类型提示模型实例这一行为。

如果你之前在路由或控制器中类型提示了模型实例，并且希望注入一个空的模型实例，那么现在应该移除这个类型提示然后在路由或控制器中直接创建一个新的模型实例；否则，Laravel 将会基于路由 URI 的标识符试图从数据库获取一个已存在的模型实例。

IronMQ

IronMQ 队列驱动被移动到自己的扩展包中，不再被框架核心支持：
http://github.com/LaravelCollective/iron-queue

任务/队列

`php artisan make:job` 命令现在默认会创建一个队列任务类，如果你想要创建一个同步任务（非队列），在使用该命令时加上 `--sync` 选项。

邮件

邮件配置中移除了 `pretend` 选项，取而代之的，使用 `log` 邮件驱动执行和 `pretend` 同样的功能，并且将邮件信息记录到日志中。

分页

为了与框架生成的其它 URL 保持一致，分页 URL 不再包含斜杠，这一改变对应用代码不产生任何影响。

服务提供者

`Illuminate\Foundation\Providers\ArtisanServiceProvider` 从配置文件 `app.php` 的服务提供者列表中移除。
`Illuminate\Routing\ControllerServiceProvider` 从配置文件 `app.php` 的服务提供者列表中移除。

Session

数据库 Session 驱动

我们为框架编写了新的 `database` Session 驱动，该驱动包含更多的用户信息，例如用户 ID、IP 地址以及用户代理，如果你想要继续使用之前的 `database` 驱动，需要在配置文件 `session.php` 中指定 `legacy-database` 驱动。

如果你想要使用新的驱动，还需要添加 `user_id (nullable integer)`、`ip_address (nullable string)` 以及 `user_agent (text)` 列到存放 Session 的数据表中。

Stringy

框架不再内置 Stringy 库，如果要在应用中使用，你需要通过 Composer 手动安装。

验证

`ValidatesRequests` trait 现在会抛出 `Illuminate\Foundation\Validation\ValidationException` 异常以取代之前的 `Illuminate\Http\Exception\HttpResponseException`。如果你没有手动捕获该异常，那么这对之前的代码没有影响。

废弃

下面这些功能在 Laravel 5.2 中被废弃，在 Laravel 5.3 中会被彻底移除：

- `Illuminate\Contracts\Bus\SelfHandling` 契约
- 集合的 `lists` 方法被重命名为 `pluck` 方法。
- 隐式控制器路由 `Route::controller` 被废弃。在路由文件中请使用明确的路由注册。
- Laravel 5.1 的 `database` Session 驱动被重命名为 `legacy-database`。
- `Str::randomBytes` 方法被废弃，直接使用 PHP 的 `random_bytes` 即可。
- `Str::equals` 方法被废弃，直接使用 PHP 的 `hash_equals` 方法即可。
- `Illuminate\View\Expression` 被废弃，使用 `Illuminate\Support\HtmlString` 即可。

贡献代码

1、缺陷报告

为了鼓励促进更加有效积极的合作，Laravel 强烈鼓励使用 GitHub 的 `pull requests`，而不是仅仅报告缺陷，“缺陷报告”也可以通过一个包含失败测试的 `pull requests` 的方式提交。

然而，如果你以文件的方式提交缺陷报告，你的问题应该包含一个标题和对该问题的明确说明，还要包含尽可能多的相关信息以及论证该问题的代码样板，缺陷报告的目的是为了让你自己和其他人更方便的重现缺陷并对其进行修复。

记住，缺陷报告被创建是为了其他人遇到同样问题的时候能够和你一起合作解决它，不要寄期望于缺陷会自动解决抑或有人跳出来修复它，创建缺陷报告是为了帮你自己和别人走上修复问题之路。

Laravel 源码通过 Github 进行管理，每一个 Laravel 项目都有其对应的代码库：

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

2、核心开发讨论

你可以在 LaraChat 的 Slack 小组的 #internals 频道讨论关于 Laravel 的 bugs、新特性、以及如何实现已有特性等。Taylor Otwell，Laravel 的主要维护者，通常在工作日的上午 8 点到下午 5 点（西六区或美国芝加哥时间）在线，其它时间也可能偶尔在线。

3、哪个分支？

所有的 bug 修复应该被提交到最新的稳定分支，永远不要把 bug 修复提交到 `master` 分支，除非它们能够修复下个发行版本中的特性。
当前版本中完全向后兼容的次要特性也可以提交到最新的稳定分支。

重要的新特性总是要被提交到 `master` 分支，包括下个发行版本。
如果你不确定是重要特性还是次要特性，请在 `#laravel-dev IRC` 频道问一下 Taylor Otwell

4、安全漏洞

如果你在 Laravel 中发现安全漏洞，请发送邮件到 taylor@laravel.com，所有的安全漏洞将会被及时解决。

5、编码风格

Laravel 遵循 PSR-2 编码标准和 PSR-4 自动载入标准。

DocBlocks

下面是注释示例：

```
/**
 * Register a binding with the container.
 *
 * @param string|array $abstract
```

```
 * @param \Closure|string|null $concrete
 * @param bool $shared
 * @return void
 */
public function bind($abstract, $concrete = null, $shared = false)
{
    //
}
```

Code Style Fixer

可以使用 PHP-CS-FIXER 在代码提交前修复代码风格。
在此之前，需要安装全局工具，然后通过在项目根目录下运行如下命令检查代码风格：

```
php-cs-fixer fix
```

二、开始

安装

1、服务器要求

Laravel 框架有对服务器有少量要求，当然，Laravel Homestead 已经满足所有这些要求，所以我们强烈推荐使用 Homestead 作为 Laravel 本地开发环境。
如果你没有使用 Homestead，那么需要保证开发环境满足以下要求：

- PHP 版本 >= 5.5.9
- PHP 扩展：OpenSSL
- PHP 扩展：PDO
- PHP 扩展：Mbstring
- PHP 扩展：Tokenizer

2、安装 Laravel

Laravel 使用 Composer 管理依赖，因此，使用 Laravel 之前，确保机器上已经安装了 Composer。

通过 Laravel 安装器

首先，通过 Composer 安装 Laravel 安装器：

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

确保 `~/.composer/vendor/bin` 在系统路径中，否则不能在任意路径调用 `laravel` 命令。安装完成后，通过简单的 `laravel new` 命令即可在当前目录下创建一个新的 Laravel 应用，例如，`laravel new blog` 将会创建一个名为 `blog` 的新应用，且包含所有 Laravel 依赖。该安装方法比通过 Composer 安装要快很多：

```
laravel new blog
```

通过 Composer Create-Project

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

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

3、基本配置

Laravel 框架的所有配置文件都存放在 `config` 目录下，并且每一个配置项都有注释，所以你可以随意浏览任意配置文件去熟悉这些配置项。

目录权限

安装完 Laravel 后，需要配置一些目录的读写权限：`storage` 和 `bootstrap/cache` 目录应该是可写的，如果你使用 Homestead 虚拟机做为开发环境，这些权限已经设置好了。

应用 Key

接下来要做的事情就是将应用的 `key`（APP_KEY）设置为一个随机字符串，如果你是通过 Composer 或者 Laravel 安装器安装的话，该 `key` 的值已经通过 `key:generate` 命令生成好了。通常，该字符串应该是 32 位长，通过 `.env` 文件中的 APP_KEY 进行配置，如果你还没有将 `.env.example` 文件重命名为 `.env`，现在立即这样做。如果应用 `key` 没有被设置，用户 Session 和其它加密数据将会有安全隐患。
如果你想要手动生成该 `key` 的值，使用如下 Artisan 命令即可：

```
php artisan key:generate
```

更多配置

Laravel 几乎不再需要其它任何配置就可以正常使用了，但是，你最好再看看 `config/app.php` 文件，其中包含了一些基于应用可能需要进行改变的配置，比如 `timezone` 和 `locale`（分别用于配置时区和本地化）。
你可能还想要配置 Laravel 的一些其它组件，比如缓存、数据库、Session 等，关于这些我们将会在后续文档一一探讨。

配置

1、介绍

Laravel 的所有配置文件都存放在 `config` 目录下，每个配置项都有注释，以保证浏览任意配置文件的配置项都能直观了解该配置项的作用及用法。

2、访问配置值

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

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

如果要在运行时设置配置值，传递数组参数到 `config` 方法即可：

```
config(['app.timezone' => 'America/Chicago']);
```

3、环境配置

基于应用运行的环境不同设置不同的配置值能够给我们开发带来极大的方便，比如，我们通常在本地和线上环境配置不同的缓存驱动，这一机制在 Laravel 中很容易实现。

Laravel 使用 Vance Lucas 开发的 PHP 库 DotEnv 来实现这一机制，在新安装的 Laravel 中，根目录下有一个 `.env.example` 文件，如果 Laravel 是通过 Composer 安装的，那么该文件已经被重命名为 `.env`，否则的话你要自己手动重命名该文件。

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

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

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

不要把 `.env` 文件提交到源码控制（svn 或 git 等）中，因为每个使用你的应用的开发者/服务器可能要求不同的环境配置。

如果你是在一个团队中进行开发，你需要将 `.env.example` 文件随你的应用一起提交到源码控制中：将一些配置值以占位符的方式放置在 `.env.example` 文件中，这样其他开发者就会很清楚运行你的应用需要配置哪些环境变量。

访问当前应用环境

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

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

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

```
if (App::environment('local')) {

    // The environment is local

}


if (App::environment('local', 'staging')) {

    // The environment is either local OR staging...

}
```

应用实例也可以通过辅助函数 `app` 来访问：

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

4、配置缓存

为了给应用加速，你可以使用 Artisan 命令 `config:cache` 将所有配置文件的配置缓存到单个文件里，这将会将所有配置选项合并到单个文件从而可以被框架快速加载。
应用一旦上线，就要运行一次 `php artisan config:cache`，但是在本地开发时，没必要经常运行该命令，因为配置值经常需要改变。

5、维护模式

当你的应用处于维护模式时，所有对应用的请求都会返回同一个自定义视图。这一机制在对应用进行升级或者维护时，使得“关闭”站点变得轻而易举。对维护模式的判断代码位于应用默认的中间件栈中，如果应用处于维护模式，则状态码为 `503` 的 `HttpException` 将会被抛出。
要开启维护模式，只需执行 Artisan 命令 `down` 即可：

```
php artisan down
```

要关闭维护模式，对应的 Artisan 命令是 `up`：

```
php artisan up
```

维护模式响应模板

默认的维护模式响应模板是 `resources/views/errors/503.blade.php`

维护模式 & 队列

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

维护模式的替代方案

由于维护模式命令的执行需要几秒时间，你可以考虑使用 Envoyer 实现 0 秒下线作为替代方案。

Laravel Homestead

1、简介

Laravel 致力于让整个 PHP 开发过程变得让人愉悦，包括本地开发环境，为此官方为我们提供了一整套本地开发环境 —— Laravel Homestead。

Laravel Homestead 是一个打包好各种 Laravel 开发所需要的工具及环境的 Vagrant 盒子（Vagrant 提供了一个便捷的方式来管理和设置虚拟机），该盒子为我们提供了优秀的开发环境，有了它，我们不再需要在本地环境安装 PHP、HHVM、Web 服务器以及其它工具软件，我们也完全不用再担心误操作搞乱操作系统 —— 因为 Vagrant 盒子是一次性的，如果出现错误，可以在数分钟内销毁并重新创建该 Vagrant 盒子！

Homestead 可以运行在 Windows、Mac 以及 Linux 系统上，其中已经安装好了 Nginx、PHP7.0、MySQL、Postgres、Redis、Memcached、Node 以及很多其它开发 Laravel 应用所需要的东西。

注：如果你使用的是 Windows，需要开启系统的硬件虚拟化（VT-x），这通常可以通过 BIOS 来开启。

预装软件

- Ubuntu 14.04
- Git
- PHP 7.0
- HHVM
- Xdebug
- Nginx
- MySQL
- SQLite 3
- Postgres
- Composer
- Node（With PM2, Bower, Grunt, and Gulp）
- Redis
- Memcached
- Beanstalkd
- Blackfire Profiler

2、安装 & 设置

首次安装

在使用 Homestead 之前，需要先安装 Virtual Box/VMWare 和 Vagrant，所有这些软件包都为常用操作系统提供了一个便于使用的可视化安装器。

安装 Homestead Vagrant 盒子

VirtualBox/VMWare 和 Vagrant 安装好了之后，在终端中使用能如下命令将 `laravel/homesterad` 添加到 Vagrant 中。下载该盒子将会花费一些时间，时间长短主要取决于你的网络连接速度：

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

如果上述命令执行失败，可以使用 Vagrant 老版本的方式，这需要输入完整的 URL：

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

通过 GitHub 安装 Homestead

你还可以通过简单克隆仓库代码来实现 Homestead 安装。将仓库克隆到用户目录下的 Homestead 目录，这样 Homestead 盒子就可以会作为所有其他 Laravel 项目的主机：

```
cd ~

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

克隆完成后，在 Homestead 目录下运行 `bash init.sh` 命令来创建 `Homestead.yaml` 配置文件，`Homestead.yaml` 配置文件文件位于 `~/.homestead` 目录：

```
bash init.sh
```

配置 Homestead

设置 Provider

`Homestead.yaml` 文件中的 `provider` 键表示使用哪个 Vagrant 提供者：`virtualbox`、`vmware_fushion` 或者 `vmware_workstation`，你可以将其设置为自己喜欢的提供者：

```
provider: virtualbox
```

配置共享文件夹

`Homestead.yaml` 文件中的 `folders` 属性列出了所有主机和 Homestead 虚拟机共享的文件夹，一旦这些目录中的文件有了修改，将会在本地和 Homestead 虚拟机之间保持同步，如果有需要的话，你可以配置多个共享文件夹（一般一个就够了）：

```
folders:
```

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

如果要开启 NFS，只需简单添加一个标识到同步文件夹配置：

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

配置 Nginx 站点

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

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

你还可以通过设置 `hhvm` 为 `true` 让所有的 Homestead 站点使用 HHVM：

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

默认情况下，每个站点都可以通过 HTTP（端口号：8000）和 HTTPS（端口号：44300）进行访问。

Hosts 文件

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

```
192.168.10.10 homestead.app
```

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

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

启动 Vagrant Box

配置好 `Homestead.yaml` 文件后，在 Homestead 目录下运行 `vagrant up` 命令，Vagrant 将会启动虚拟机并自动配置共享文件夹以及 Nginx 站点。
销毁该机器，可以使用 vagrant destroy –force

为指定项目安装 Homestead

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

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

这样就在项目中安装了 Homestead。Homestead 安装完成后，使用 `make` 命令生成 `Vagrantfile` 和 `Homestead.yaml` 文件，`make` 命令将会自动配置 `Homestead.yaml` 中的 `sites` 和 `folders` 属性。
Mac/Linux：

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

Windows:

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

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

3、日常使用

全局访问 Homestead

有时候你想要在文件系统的任意位置运行 `vagrant up` 启动 Homestead 虚拟机，要实现这一目的需要将 Homestead 安装目录添加到系统路径。这样你就可以在系统的任意位置运行 `homestead` 或 `homestead ssh` 来启动/登录虚拟机。

通过 SSH 连接

你可以在 Homestead 目录下通过运行 `vagrant ssh` 以 SSH 方式连接到虚拟机，但是如果你需要以更平滑的方式连接到 Homestead，可以为主机添加一个别名来快速连接到 Homestead 盒子，创建完别名后，可以使用 `vm` 命令从任何地方以 SSH 方式连接到 Homestead 虚拟机：

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

连接到数据库

默认已经在 Homestead 虚拟机中为 MySQL 和 Postgres 数据库做好了配置，更方便的是，Laravel 的 `.env` 还为连接 Homestead 数据库做好了配置。
想要通过本地的 Navicat 或 Sequel Pro 连接到 Homestead 上的 MySQL 或 Postgres 数据库，可以通过新建连接来实现，主机 IP 都是 `127.0.0.1`，对于 MySQL 而言，端口号是 `33060`，对 Postgres 而言，端口号是 `54320`，用户名/密码是 `homestead`/`secret`。
注意：只有从本地连接 Homestead 的数据库时才能使用这些非标准的端口，在 Homestead 虚拟机中还是应该使用默认的 3306 和 5432 端口进行数据库连接配置。

添加更多站点

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

配置 Cron 调度任务

Laravel 提供了很方便的方式来调度 Cron 任务：只需每分钟调度运行一次 Artisan 命令 `schedule:run` 即可。`schedule:run` 会检查定义在 `App\Console\Kernel` 类中定义的调度任务并判断运行哪些任务。
如果想要为某个 Homestead 站点运行 `schedule:run` 命令，需要在定义站点时设置 `schedule` 为 `true`：

```
sites:

    - map: homestead.app

      to: /home/vagrant/Code/Laravel/public

      schedule: true
```

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

端口转发配置

默认情况下，Homestead 端口转发配置如下：

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

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

```
ports:

    - send: 93000
```

```
      to: 9300

    - send: 7777

      to: 777

      protocol: udp
```

4、使用 Blackfire Profiler 进行性能分析

SensioLabs 开发的 Blackfire Profiler 能自动收集代码执行数据，比如内存、CPU 时间、硬盘 I/O 等，Homestead 使得在应用中使用该性能分析器变得轻而易举。
Blackfire Profiler 需要的软件包已经预安装到 Homestead 盒子，你只需要在 `Homestead.yaml` 文件中设置 Blackfire Server ID 和 token：

```
blackfire:

    - id: your-server-id

      token: your-server-token

      client-id: your-client-id

      client-token: your-client-token
```

配置好 Blackfire 的凭证之后，在 Homestead 目录下使用 `vagrant provision` 重新启动 Homestead。在此之前，确保你已经查看过 Blackfire 文档了解了如何在浏览器安装相应的 Blackfire 扩展。

三、基础

HTTP 路由

1、基本路由

所有应用路由都定义在 `App\Providers\RouteServiceProvider` 类载入的 `app/Http/routes.php` 文件中。
最基本的 Laravel 路由接收一个 URI 和一个闭包：

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

    return 'Hello World';

});
```

```
Route::post('foo', function () {

    //

});
```

默认情况下，`routes.php` 文件包含单个路由和一个路由群组，该路由群组包含的所有路由都使用了中间件组 `web`，而这个中间件组为路由提供了 Session 状态和 CSRF 保护功能。通常，我们会将所有路由定义在这个路由组中。

有效的路由方法

我们可以注册路由来响应任何 HTTP 请求：

```
Route::get($uri, $callback);

Route::post($uri, $callback);

Route::put($uri, $callback);

Route::patch($uri, $callback);

Route::delete($uri, $callback);

Route::options($uri, $callback);
```

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

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

    //

});


Route::any('foo', function () {

    //

});
```

2、路由参数

必选参数

有时我们需要在路由中捕获 URI 片段。比如，要从 URL 中捕获用户 ID，需要通过如下方式定义路由参数：

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

    return 'User '.$id;
```

```
});
```

可以按需要在路由中定义多个路由参数：

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

路由参数总是通过花括号进行包裹，这些参数在路由被执行时会被传递到路由的闭包。

注意：路由参数不能包含 `-` 字符，需要的话可以使用 `_` 替代。

可选参数

有时候可能需要指定可选的路由参数，这可以通过在参数名后加一个 `?` 标记来实现，这种情况下需要给相应的变量指定默认值：

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

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

3、命名路由

命名路由为生成 URL 或重定向提供了便利。实现也很简单，在定义路由时使用数组键 `as` 指定路由名称：

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

此外，还可以为控制器动作指定路由名称：

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

此外，除了在路由数组定义中指定路由名称外，还可以通过在路由定义之后使用 `name` 方法链的方式来实现：

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

路由群组 & 命名路由

如果你在使用路由群组，可以通过在路由群组的属性数组中指定 `as` 关键字来为群组中的路由设置一个共用的路由名前缀：

```
Route::group(['as' => 'admin::'], function () {
    Route::get('dashboard', ['as' => 'dashboard', function () {
        // 路由被命名为 "admin::dashboard"
    }]);
});
```

为命名路由生成 URL

如果你为给定路由进行了命名，就可以通过 `route` 函数为该命名路由生成对应 URL：

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

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

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

4、路由群组

路由群组允许我们在多个路由中共享路由属性，比如中间件和命名空间等，这样的话我们就不必为每一个路由单独定义属性。共享属性以数组的形式作为第一个参数被传递给 `Route::group` 方法。
下面我们通过几个简单的应用实例来演示路由群组。

中间件

要给路由群组中定义的所有路由分配中间件，可以在群组属性数组中使用 `middleware`。中间件将会按照数组中定义的顺序依次执行：

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

    Route::get('user/profile', function () {
        // 使用 Auth 中间件
    });
});
```

命名空间

另一个通用的例子是路由群组分配同一个 PHP 命名空间给其下的多个控制器，可以在分组属性数组中使用 `namespace` 来指定群组中所有控制器的公共命名空间：

```
Route::group(['namespace' => 'Admin'], function(){
    // 控制器在 "App\Http\Controllers\Admin" 命名空间下

    Route::group(['namespace' => 'User'], function(){
        // 控制器在 "App\Http\Controllers\Admin\User" 命名空间下
    });
});
```

默认情况下，`RouteServiceProvider` 引入 `routes.php` 并指定其下所有控制器类所在的默认命名空间 `App\Http\Controllers`，因此，我们在定义的时候只需要指定命名空间 `App\Http\Controllers` 之后的部分即可。

子域名路由

路由群组还可以被用于子域名路由通配符，子域名可以像 URI 一样被分配给路由参数，从而允许捕获子域名的部分用于路由或者控制器，子域名可以通过群组属性数组中的 `domain` 来指定：

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

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

路由前缀

群组属性 `prefix` 可以用来为群组中每个路由添加一个给定 URI 前缀，比如，你可以为所有路由 URI 添加 `admin` 前缀 ：

```
Route::group(['prefix' => 'admin'], function () {
    Route::get('users', function () {
        // 匹配 "/admin/users" URL
    });
});
```

你还可以使用 `prefix` 参数为路由群组指定公共路由参数：

```
Route::group(['prefix' => 'accounts/{account_id}'], function () {
    Route::get('detail', function ($account_id) {
        // 匹配 accounts/{account_id}/detail URL
    });
});
```

5、CSRF 攻击

简介

跨站请求伪造是一种通过伪装授权用户的请求来利用授信网站的恶意漏洞。Laravel 使得防止应用遭到跨站请求伪造攻击变得简单。

Laravel 自动为每一个被应用管理的有效用户会话生成一个 CSRF “令牌”，该令牌用于验证授权用户和发起请求者是否是同一个人。想要生成包含 CSRF 令牌的隐藏输入字段，可以使用帮助函数 `csrf_field` 来实现：

```
<?php echo csrf_field(); ?>
```

辅助函数 `csrf_field` 会生成如下 HTML：

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

当然还可以使用 Blade 模板引擎提供的方式：

```
{!! csrf_field() !!}
```

你不需要自己编写代码去验证 POST、PUT 或者 DELETE 请求的 CSRF 令牌，因为 Laravel 自带的 HTTP 中间件 `VerifyCsrfToken` 会为我们做这项工作：将请求中输入的 `token` 值和 Session 中的存储的 `token` 作对比来进行验证。

从 CSRF 保护中排除指定 URL

有时候我们需要从 CSRF 保护中排除一些 URL，比如，如果你使用了 Stripe 来处理支付并用到他们的 webhook 系统，这时候就需要从 Laravel 的 CSRF 保护中排除 webhook 处理器路由。

要实现这一目的，你需要在 `VerifyCsrfToken` 中间件中将要排除的 URL 添加到 `$except` 属性：

```
<?php

namespace App\Http\Middleware;

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

class VerifyCsrfToken extends BaseVerifier
{
    /**
     *从 CSRF 验证中排除的 URL
     *
     * @var array
     */
    protected $except = [
        'stripe/*',
    ];
}
```

X-CSRF-Token

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

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

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

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

X-XSRF-Token

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

6、路由模型绑定

Laravel 路由模型绑定为注入类实例到路由提供了方便，例如，你可以将匹配给定 ID 的整个 User 类实例注入到路由中，而不是直接注入用户 ID。

隐式绑定

Laravel 会自动解析定义在路由或控制器动作（变量名匹配路由片段）中的 Eloquent 模型类型声明，例如：

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

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

自定义键名

如果你想要隐式模型绑定使用数据表的其它字段，可以重写 Eloquent 模型类的 `getRouteKeyName` 方法：

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

显式绑定

要注册显式绑定，需要使用路由的 `model` 方法来为给定参数指定绑定类。应该在 `RouteServiceProvider::boot` 方法中定义模型绑定：

绑定参数到模型

```
public function boot(Router $router)
{
    parent::boot($router);
    $router->model('user', 'App\User');
}
```

接下来，定义一个包含 `{user}` 参数的路由：

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

由于我们已经绑定 `{user}` 参数到 `App\User` 模型，User 实例会被注入到该路由。因此，如果请求 URL 是 `profile/1`，就会注入一个用户 ID 为 1 的 User 实例。
如果匹配的模型实例在数据库不存在，会自动生成并返回 HTTP 404 响应。

自定义解析逻辑

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

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

自定义“Not Found”
如果你想要指定自己的“Not Found”行为，将封装该行为的闭包作为第三个参数传递给 `model` 方法：

```
$router->model('user', 'App\User', function() {
    throw new NotFoundHttpException;
});
```

7、表单方法伪造

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

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

还可以使用辅助函数 `method_field` 来实现这一目的：

```
<?php echo method_field('PUT'); ?>
```

当然，也支持 Blade 模板引擎：

```
{{ method_field('PUT') }}
```

HTTP 中间件

1、简介

HTTP 中间件提供了为过滤进入应用的 HTTP 请求提供了一套便利的机制。例如，Laravel 内置了一个中间件来验证用户是否经过授权，如果用户没有经过授权，中间件会将

用户重定向到登录页面，否则如果用户经过授权，中间件就会允许请求继续往前进入下一步操作。
当然，除了认证之外，中间件还可以被用来处理更多其它任务。比如：CORS 中间件可以用于为离开站点的响应添加合适的头（跨域）；日志中间件可以记录所有进入站点的请求。

Laravel 框架自带了一些中间件，包括维护模式、认证、CSRF 保护中间件等等。所有的中间件都位于 `app/Http/Middleware` 目录。

2、定义中间件

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

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

这个命令会在 `app/Http/Middleware` 目录下创建一个新的中间件类 `OldMiddleware`，在这个中间件中，我们只允许提供的 `age` 大于 `200` 的访问路由，否则，我们将用户重定向到主页：

```
<?php

namespace App\Http\Middleware;

use Closure;

class OldMiddleware
{
    /**
     * 返回请求过滤器
     *
     * @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);
    }

}
```

正如你所看到的，如果 `age<=200`，中间件会返回一个 HTTP 重定向到客户端；否则，请求会被传递下去。将请求往下传递可以通过调用回调函数 `$next` 并传入 `$request`。理解中间件的最好方式就是将中间件看做 HTTP 请求到达目标动作之前必须经过的“层”，每一层都会检查请求并且可以完全拒绝它。

中间件之前/之后

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

```
<?php

namespace App\Http\Middleware;

use Closure;

class BeforeMiddleware
{
    public function handle($request, Closure $next)
    {
        // 执行动作

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

然而，下面这个中间件则会在请求处理后执行其任务：

```
<?php


namespace App\Http\Middleware;


use Closure;


class AfterMiddleware

{

    public function handle($request, Closure $next)

    {

        $response = $next($request);


        // 执行动作


        return $response;

    }

}
```

3、注册中间件

全局中间件

如果你想要中间件在每一个 HTTP 请求期间被执行，只需要将相应的中间件类设置到 `app/Http/Kernel.php` 的数组属性 `$middleware` 中即可。

分配中间件到路由

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

```
// 在 App\Http\Kernel 里中

protected $routeMiddleware = [

    'auth' => \App\Http\Middleware\Authenticate::class,

    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::c
lass,
```

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

中间件在 HTTP Kernel 中被定义后，可以在路由选项数组中使用 `$middleware` 键来指定该中间件：

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

使用数组分配多个中间件到路由：

```
Route::get('/', ['middleware' => ['first', 'second'], function () {
    //
}]);
```

除了使用数组外，还可以使用 `middleware` 方法链的方式定义路由：

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

中间件组

有时候你可能想要通过指定一个键名的方式将相关中间件分到一个组里面，从而更方便将其分配到路由中，这可以通过使用 HTTP Kernel 的 `$middlewareGroups` 实现。Laravel 自带了开箱即用的 `web` 和 `api` 两个中间件组以包含可以应用到 Web UI 和 API 路由的通用中间件：

```
/**
 * 应用的路由中间件组
 *
 * @var array
 */
protected $middlewareGroups = [
    'web' => [
```

```
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \App\Http\Middleware\VerifyCsrfToken::class,
    ],

    'api' => [
        'throttle:60,1',
        'auth:api',
    ],
];
```

中间件组可以被分配给路由和控制器动作，使用和单个中间件分配同样的语法。再次申明，中间件组的目的只是让一次分配给路由多个中间件的实现更加简单：

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

4、中间件参数

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

```
<?php

namespace App\Http\Middleware;

use Closure;

class RoleMiddleware
{
```

```
    /**
     * 运行请求过滤器
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure $next
     * @param string $role
     * @return mixed
     * translator http://laravelacademy.org
     */
    public function handle($request, Closure $next, $role)
    {
        if (! $request->user()->hasRole($role)) {
            // Redirect...
        }

        return $next($request);
    }

}
```

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

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

5、可终止的中间件

有时候中间件可能需要在 HTTP 响应发送到浏览器之后做一些工作。比如，Laravel 内置的“session”中间件会在响应发送到浏览器之后将 Session 数据写到存储器中，为了实现这个，定义一个可终止的中间件并添加 `terminate` 方法到这个中间件：

```
<?php
```

```
namespace Illuminate\Session\Middleware;

use Closure;

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

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

`terminate` 方法将会接收请求和响应作为参数。一旦你定义了一个可终止的中间件，应该将其加入到 HTTP kernel 的全局中间件列表中。

当调用中间件上的 `terminate` 方法时，Laravel 将会从服务容器中取出该中间件的新的实例，如果你想要在调用 `handle` 和 `terminate` 方法时使用同一个中间件实例，则需要使用容器的 `singleton` 方法将该中间件注册到容器中。

HTTP 控制器

1、简介

将所有的请求处理逻辑都放在单个 `routes.php` 中显然是不合理的，你也许还希望使用控制器类组织管理这些行为。控制器可以将相关的 HTTP 请求封装到一个类中进行处理。通常控制器存放在 `app/Http/Controllers` 目录中。

2、基本控制器

下面是一个基本控制器类的例子。所有的 Laravel 控制器应该继承自 Laravel 自带的控制器基类 `Controller`：

```
<?php


namespace App\Http\Controllers;


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


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

我们可以像这样定义指向该控制器动作的路由：

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

现在，如果一个请求匹配上面的路由 URI，`UserController` 的 `showProfile` 方法就会被执行。当然，路由参数也会被传递给这个方法。

控制器 & 命名空间

你应该注意到我们在定义控制器路由的时候没有指定完整的控制器命名空间，而只是定义了 `App\Http\Controllers` 之后的部分。默认情况下，`RouteServiceProvider` 将会在一个路由群组中载入 `routes.php` 文件，并且该路由群组指定定了群组中路由控制器所在的命名空间。

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

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

命名控制器路由

和闭包路由一样，可以指定控制器路由的名称：

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

你还可以使用辅助函数 route 来为已命名的控制器路由生成对应的 URL：

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

3、控制器中间件

中间件可以像这样分配给控制器路由：

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

但是，将中间件放在控制器构造函数中更方便，在控制器的构造函数中使用 middleware 方法你可以很轻松的分配中间件给该控制器。你甚至可以限定该中间件应用到该控制器类的指定方法：

```
class UserController extends Controller
{
    /**
     * 实例化一个新的 UserController 实例
     *
     * @return void
     */
    public function __construct()
    {
        $this->middleware('auth');
```

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

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

    }

}
```

4、RESTful 资源控制器

Laravel 的资源控制器使得构建围绕资源的 RESTful 控制器变得毫无痛苦，例如，你可能想要在应用中创建一个控制器，用于处理关于图片存储的 HTTP 请求，使用 Artisan 命令 `make:controller`，我们可以快速创建这样的控制器：

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

该 Artisan 命令将会生成一个控制器文件 `app/Http/Controllers/PhotoController.php`，这个控制器包含了每一个资源操作对应的方法。
接下来，可以为该控制器注册一个资源路由：

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

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

资源控制器处理的动作

方法	路径	动作	路由名称
GET	`/photo`	index	photo.index
GET	`/photo/create`	create	photo.create
POST	`/photo`	store	photo.store
GET	`/photo/{photo}`	show	photo.show
GET	`/photo/{photo}/edit`	edit	photo.edit
PUT/PATCH	`/photo/{photo}`	update	photo.update
DELETE	`/photo/{photo}`	destroy	photo.destroy

只定义部分资源路由

声明资源路由时可以指定该路由处理的动作子集：

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


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

命名资源路由

默认情况下，所有资源控制器动作都有一个路由名称，然而，我们可以通过传入 `names` 数组来覆盖这些默认的名字：

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

补充资源控制器

如果有必要在默认资源路由之外添加额外的路由到资源控制器，应该在调用 `Route::resource` 之前定义这些路由；否则，通过 `resource` 方法定义的路由可能无意中优先于补充的额外路由：

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

5、依赖注入 & 控制器

构造函数注入

Laravel 使用服务容器解析所有的 Laravel 控制器，因此，可以在控制器的构造函数中类型声明任何依赖，这些依赖会被自动解析并注入到控制器实例中：

```
<?php


namespace App\Http\Controllers;


use Illuminate\Routing\Controller;
```

```
use App\Repositories\UserRepository;


class UserController extends Controller
{
    /**
     * The user repository instance.
     */
    protected $users;


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

当然，你还可以类型提示任何 Laravel 契约，如果容器可以解析，就可以进行类型提示。

方法注入

除了构造函数注入之外，还可以在控制器的动作方法中进行依赖的类型提示，例如，我们可以在某个方法中类型提示 `Illuminate\Http\Request` 实例：

```
<?php


namespace App\Http\Controllers;


use Illuminate\Http\Request;
```

```
use Illuminate\Routing\Controller;


class UserController extends Controller
{
    /**
     * 存储新用户
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        $name = $request->input('name');


        //
    }
}
```

如果控制器方法期望输入路由参数，只需要将路由参数放到其他依赖之后，例如，如果你的路由定义如下：

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

你需要通过定义控制器方法如下所示来类型提示 `Illuminate\Http\Request` 并访问路由参数 `id`：

```
<?php


namespace App\Http\Controllers;


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

```
class UserController extends Controller
{
    /**
     * 更新指定用户
     *
     * @param Request $request
     * @param int $id
     * @return Response
     * @translator http://laravelacademy.org
     */
    public function update(Request $request, $id)
    {
        //
    }
}
```

6、路由缓存

注意：路由缓存不会作用于基于闭包的路由。要使用路由缓存，必须将闭包路由转化为控制器路由。

如果你的应用完全基于控制器路由，可以使用 Laravel 的路由缓存，使用路由缓存将会极大减少注册所有应用路由所花费的时间开销，在某些案例中，路由注册速度甚至能提高 100 倍！想要生成路由缓存，只需执行 Artisan 命令 `route:cache`:

```
php artisan route:cache
```

就这么简单！你的缓存路由文件现在取代 `app/Http/routes.php` 文件被使用，记住，如果你添加新的路由需要重新生成路由缓存。因此，只有在项目部署阶段才需要运行 `route:cache` 命令。

想要移除缓存路由文件，使用 `route:clear` 命令即可：

```
php artisan route:clear
```

HTTP 请求

1、访问请求实例

通过依赖注入获取当前 HTTP 请求实例，应该在控制器的构造函数或方法中对 Illuminate\Http\Request 类进行类型提示，当前请求实例会被服务容器自动注入：

```
<?php


namespace App\Http\Controllers;


use Illuminate\Http\Request;

use Illuminate\Routing\Controller;


class UserController extends Controller

{

    /**

     * 存储新用户

     *

     * @param Request $request

     * @return Response

     */

    public function store(Request $request)

    {

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


        //

    }

}
```

如果你的控制器方法还期望获取路由参数输入，只需要将路由参数置于其它依赖之后即可，例如，如果你的路由定义如下：

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

你仍然可以对 `Illuminate\Http\Request` 进行类型提示并通过如下方式定义控制器方法来访问路由参数：

```
<?php


namespace App\Http\Controllers;


use Illuminate\Http\Request;

use Illuminate\Routing\Controller;


classUser Controller extends Controller

{

    /**

     * 更新指定用户

     *

     * @param Request $request

     * @param int $id

     * @return Response

     */

    public function update(Request $request,$id)

    {

        //

    }

}
```

基本请求信息

`Illuminate\Http\Request` 实例提供了多个方法来检测应用的 HTTP 请求，Laravel 的 `Illuminate\Http\Request` 继承自 `Symfony\Component\HttpFoundation\Request` 类，下面演示了一些该类中的有用方法：

获取请求 URI

path 方法将会返回请求的 URI，因此，如果进入的请求路径是 http://domain.com/foo/bar，则 path 方法将会返回 foo/bar：

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

is 方法允许你验证进入的请求是否与给定模式匹配。使用该方法时可以使用 * 通配符：

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

//

}
```

想要获取完整的 URL，而不仅仅是路径信息，可以使用请求实例中的 url 或 fullUrl 方法：

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

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

获取请求方法

method 方法将会返回请求的 HTTP 请求方式。你还可以使用 isMethod 方法来验证 HTTP 请求方式是否匹配给定字符串：

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

if($request->isMethod('post')){

    //

}
```

PSR-7 请求

PSR-7 标准指定了 HTTP 消息接口，包括请求和响应。如果你想要获取 PSR-7 请求实例，首先需要安装一些库，Laravel 使用 Symfony HTTP Message Bridge 组件将典型的 Laravel 请求和响应转化为 PSR-7 兼容的实现：

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

composer require zendframework/zend-diactoros
```

安装完这些库之后，你只需要在路由或控制器中通过对请求类型进行类型提示就可以获取 PSR-7 请求：

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

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

//

});
```

如果从路由或控制器返回的是 PSR-7 响应实例，则其将会自动转化为 Laravel 响应实例并显示出来。

2、获取请求输入

获取输入值

使用一些简单的方法，就可以从 `Illuminate\Http\Request` 实例中访问用户输入。你不需要关心请求所使用的 HTTP 请求方法，因为对所有请求方式的输入访问接口都是一致的：

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

此外，你也可以通过使用 `Illuminate\Http\Request` 实例上的动态属性来访问用户输入。例如，如果你的应用表单包含 `name` 字段，那么可以像这样访问提交的值：

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

你还可以传递一个默认值作为第二个参数给 `input` 方法，如果请求输入值在当前请求未出现时该值将会被返回：

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

处理表单数组输入时，可以使用”.”来访问数组输入：

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

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

判断输入值是否出现

判断值是否在请求中出现，可以使用 `has` 方法，如果值出现过了且不为空，`has` 方法返回 `true`：

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

    //

}
```

获取所有输入数据

你还可以通过 `all` 方法获取所有输入数据：

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

获取输入的部分数据

如果你需要取出输入数据的子集，可以使用 `only` 或 `except` 方法，这两个方法都接收一个数组或动态列表作为唯一参数：

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

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


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

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

上一次请求输入

Laravel 允许你在两次请求之间保存输入数据，这个特性在检测校验数据失败后需要重新填充表单数据时很有用，但如果你使用的是 Laravel 内置的验证服务，则不需要手动使用这些方法，因为一些 Laravel 内置的校验设置会自动调用它们。

将输入存储到一次性 Session

`Illuminate\Http\Request` 实例的 `flash` 方法会将当前输入存放到一次性 Session（所谓的一次性指的是从 Session 中取出数据中，对应数据会从 Session 中销毁）中，这样在下一次请求时数据依然有效：

```


$request->flash();
```

你还可以使用 `flashOnly` 和 `flashExcept` 方法将输入数据子集存放到 Session 中：

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

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

将输入存储到一次性 Session 然后重定向

如果你经常需要一次性存储输入并重定向到前一页，可以简单使用 `withInput` 方法来将输入数据链接到 `redirect` 后面：

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

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

取出上次请求数据

要从 Session 中取出上次请求的输入数据，可以使用 Request 实例的 `old` 方法。`old` 方法提供了便利的方式从 Session 中取出一次性数据：

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

Laravel 还提供了一个全局的辅助函数 old，如果你是在 Blade 模板中显示上次输入数据，使用辅助函数 old 更方便，如果给定参数没有对应输入，返回 null：

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

Cookies

从请求中取出 Cookies

Laravel 框架创建的所有 cookies 都经过加密并使用一个认证码进行签名，这意味着如果客户端修改了它们则需要对其进行有效性验证。我们使用 Illuminate\Http\Request 实例的 cookie 方法从请求中获取 cookie 的值：

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

新增 Cookie 到响应

Laravel 提供了一个全局的辅助函数 cookie 作为一个简单工厂来生成新的 Symfony\Component\HttpFoundation\Cookie 实例，新增的 cookie 通过 withCookie 方法被附加到 Illuminate\Http\Response 实例：

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

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

return $response;
```

想要创建一个长期有效的 cookie，可以使用 cookie 工厂的 forever 方法：

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

文件上传

获取上传的文件

可以使用 Illuminate\Http\Request 实例的 file 方法来访问上传文件，该方法返回的对象是 Symfony\Component\HttpFoundation\File\UploadedFile 类的一个实例，该类继承自 PHP 标准库中提供与文件交互方法的 SplFileInfo 类：

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

验证文件是否存在使用 hasFile 方法判断文件在请求中是否存在：

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

    //
```

```
}
```

验证文件是否上传成功

使用 `isValid` 方法判断文件在上传过程中是否出错：

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

    //

}
```

保存上传的文件

使用 `move` 方法将上传文件保存到新的路径，该方法将上传文件从临时目录（在 PHP 配置文件中配置）移动到指定新目录：

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

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

其它文件方法

UploadedFile 实例中很有很多其它方法，查看该类的 API 了解更多相关方法。

HTTP 响应

1、基本响应

所有路由和控制器都会返回某种被发送到用户浏览器的响应，Laravel 提供了多种不同的方式来返回响应，最基本的响应就是从路由或控制器返回一个简单的字符串：

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

    return 'Hello World';

});
```

给定的字符串会被框架自动转化为 HTTP 响应。

Response 对象

然而，大多数路由和控制器动作都会返回一个完整的 `Illuminate\Http\Response` 实例或视图，返回一个完整的 Response 实例允许你自定义响应的 HTTP 状态码和头信息。Response 实例继承自 `Symfony\Component\HttpFoundation\Response` 类，该类提供了一系列方法用于创建 HTTP 响应：

```
use Illuminate\Http\Response;
```

```
Route::get('home', function () {
return (new Response($content, $status))
->header('Content-Type', $value);
});
```

为方便起见，还可以使用辅助函数 `response`：

```
Route::get('home', function () {
    return response($content, $status)
        ->header('Content-Type', $value);
});
```

注：查看完整的 Response 方法列表，请移步相应的 API 文档 以及 Symfony API 文档。

添加响应头

大部分响应方法都是可以以方法链形式调用的，从而使得可以平滑的构建响应（流接口模式）。例如，可以使用 `header` 方法来添加一系列响应头：

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

或者你可以使用 `withHeaders` 方法来指定头信息数组并添加到响应：

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

添加 Cookie 到响应

使用响应实例的辅助函数 `cookie` 可以轻松添加 Cookie 到响应：

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

cookie 方法接收额外的可选参数从而允许对 Cookie 属性进行更多的自定义：

```
->cookie($name, $value, $minutes, $path, $domain, $secure, $httpOnly)
```

默认情况下，Laravel 框架生成的 Cookie 经过了加密和签名，以免在客户端被篡改。如果你想要让特定的 Cookie 子集在生成时取消加密，可以使用中间件 App\Http\Middleware\EncryptCookies 的 $except 属性来排除这些 Cookie：

```
/**
 * 不需要被加密的 cookies 名称
 *
 * @var array
 */
protected $except = [
    'cookie_name',
];
```

2、其它响应类型

辅助函数 response 可以很方便地用来生成其他类型的响应实例，当无参数调用 response 时会返回 Illuminate\Contracts\Routing\ResponseFactory 契约的一个实现类实例，该契约提供了一些有用的方法来生成响应。

视图

如果你需要控制响应状态和响应头，并且还需要返回一个视图作为响应内容，可以使用 view 方法：

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

当然，如果你不需要传递自定义的 HTTP 状态码和头信息，只需要简单使用全局辅助函数 view 即可。

JSON

json 方法会自动将 Content-Type 头设置为 application/json，并使用 PHP 函数 json_encode 方法将给定数组转化为 JSON：

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

如果你想要创建一个 JSONP 响应，可以在 json 方法之后调用 setCallback 方法：

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

```
    ->setCallback($request->input('callback'));
```

文件下载

download 方法用于生成强制用户浏览器下载给定路径文件的响应。download 方法接受文件名作为第二个参数，该参数决定用户下载文件的显示名称，你还可以将 HTTP 头信息作为第三个参数传递到该方法：

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

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

注意：管理文件下载的 Symfony HttpFoundation 类要求被下载文件有一个 ASCII 文件名。

3、重定向

重定向响应是 Illuminate\Http\RedirectResponse 类的实例，其中包含了必须的头信息将用户重定向到另一个 URL，有很多方式来生成 RedirectResponse 实例，最简单的方法就是使用全局辅助函数 redirect：

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

    return redirect('home/dashboard');

});
```

有时候你想要将用户重定向到上一个请求的位置，比如，表单提交后，验证不通过，你就可以使用辅助函数 back 返回到前一个 URL（使用该方法之前确保路由使用了 web 中间件组或者都使用了 session 中间件）：

```
Route::post('user/profile', function () {

    // 验证请求...

    return back()->withInput();

});
```

重定向到命名路由

如果调用不带参数的 redirect 方法，会返回一个 Illuminate\Routing\Redirector 实例，然后可以调用该实例上的任何方法。比如，为了生成一个 RedirectResponse 到命名路由，可以使用 route 方法：

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

如果路由中有参数，可以将其作为第二个参数传递到 route 方法：

```
// For a route with the following URI: profile/{id}

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

如果要重定向到带 ID 参数的路由（ Eloquent 模型绑定 ），可以传递模型本身，ID 会被自动解析出来：

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

重定向到控制器动作

你还可以生成重定向到控制器动作，只需简单传递控制器和动作名到 `action` 方法即可。记住，你不需要指定控制器的完整命名空间，因为 Laravel 的 `RouteServiceProvider` 将会自动设置默认的控制器命名空间：

```
return redirect()->action('HomeController@index');
```

当然，如果控制器路由要求参数，你可以将参数作为第二个参数传递给 `action` 方法：

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

带一次性 Session 数据的重定向

重定向到一个新的 URL 并将数据存储到一次性 Session 中通常是同时完成的，为了方便，可以创建一个 `RedirectResponse` 实例然后在同一个方法链上将数据存储到 Session，这种方式在 action 之后存储状态信息时特别方便：

```
Route::post('user/profile', function () {
    // 更新用户属性...
    return redirect('dashboard')->with('status', 'Profile updated!');
});
```

当然，用户重定向到新页面之后，你可以从 Session 中取出并显示这些一次性信息，比如，使用 Blade 语法实现如下：

```
@if (session('status'))
    <div class="alert alert-success">
        {{ session('status') }}
    </div>
@endif
```

4、响应宏

如果你想要定义一个自定义的响应并且在多个路由和控制器中复用，可以使用 `Illuminate\Contracts\Routing\ResponseFactory` 实现类或者 `Response` 门面上的 `macro` 方法。

比如，在某个服务提供者的 boot 方法中编写代码如下：

```
<?php


namespace App\Providers;


use Response;

use Illuminate\Support\ServiceProvider;


class ResponseMacroServiceProvider extends ServiceProvider

{

    /**

     * Perform post-registration booting of services.

     *

     * @return void

     */

    public function boot()

    {

        Response::macro('caps', function ($value) {

            return Response::make(strtoupper($value));

        });

    }

}
```

`macro` 方法接收响应名称作为第一个参数，闭包函数作为第二个参数，`macro` 的闭包在 `ResponseFactory` 实现类或辅助函数 `response` 中调用 `macro` 名称的时候被执行：

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

视图

1、基本使用

视图包含应用的 HTML 代码并将应用的控制器逻辑和表现逻辑进行分离。视图文件存放在 `resources/views` 目录。
下面是一个简单视图：

```
<!-- 该视图存放 resources/views/greeting.php -->


<html>

    <body>

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

    </body>

</html>
```

由于这个视图存放在 `resources/views/greeting.php`，我们可以在全局的辅助函数 `view` 中这样返回它：

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

    return view('greeting', ['name' => 'James']);

});
```

正如你所看到的，传递给 `view` 方法的第一个参数是 `resources/views` 目录下相应的视图文件的名字，第二个参数是一个数组，该数组包含了在该视图中所有有效的数据。在这个例子中，我们传递了一个 `name` 变量，在视图中通过执行 `echo` 将其显示出来。
当然，视图还可以嵌套在 `resources/views` 的子目录中，用“.”号来引用嵌套视图，比如，如果视图存放路径是 `resources/views/admin/profile.php`，那我们可以这样引用它：

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

判断视图是否存在

如果需要判断视图是否存在，可调用在不带参数的 `view` 之后使用 `exists` 方法，如果视图在磁盘存在则返回 `true`：

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

    //
```

```
}
```

调用不带参数的 `view` 时，将会返回一个 `Illuminate\Contracts\View\Factory` 实例，从而可以调用该工厂上的所有方法。

2、视图数据

传递数据到视图

在上述例子中可以看到，我们可以简单通过数组方式将数据传递到视图：

```
return view('greetings', ['name' => 'Victoria']);
```

以这种方式传递数据的话，`$data` 应该是一个键值对数组，在视图中，就可以使用相应的键来访问数据值，比如 `<?php echo $key; ?>`。除此之外，还可以通过 `with` 方法添加独立的数据片段到视图：

```
$view = view('greeting')->with('name', 'Victoria');
```

在视图间共享数据

有时候我们需要在所有视图之间共享数据片段，这时候可以使用视图工厂的 `share` 方法，通常，需要在服务提供者的 `boot` 方法中调用 `share` 方法，你可以将其添加到 `AppServiceProvider` 或生成独立的服务提供者来存放它们：

```
<?php


namespace App\Providers;


class AppServiceProvider extends ServiceProvider
{
    /**
     * 启动所有应用服务
     *
     * @return void
     */
    public function boot()
    {
        view()->share('key', 'value');
```

```
    }

    /**
     * 注册服务提供者
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

3、视图 Composer

视图 Composer 是当视图被渲染时的回调或类方法。如果你有一些数据要在视图每次渲染时都做绑定，可以使用视图 Composer 将逻辑组织到一个单独的地方。

首先要在服务提供者中注册视图 Composer，我们将会使用辅助函数 `view` 来访问 `Illuminate\Contracts\View\Factory` 的底层实现，记住，Laravel 不会包含默认的视图 Composer 目录，我们可以按照自己的喜好组织其路径，例如可以创建一个 `App\Http\ViewComposers` 目录：

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class ComposerServiceProvider extends ServiceProvider
{
    /**
     * 在容器中注册绑定.
     *
```

```
     * @return void
     * @author http://laravelacademy.org
     */
    public function boot()
    {
        // 使用基于类的 composers...
        view()->composer(
            'profile', 'App\Http\ViewComposers\ProfileComposer'
        );

        // 使用基于闭包的 composers...
        view()->composer('dashboard', function ($view) {
        });
    }

    /**
     * 注册服务提供者.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

如果创建一个新的服务提供者来包含视图 Composer 注册，需要添加该服务提供者到配置文件 `config/app.php` 的 `providers` 数组中。

现在我们已经注册了 Composer，每次 `profile` 视图被渲染时都会执行 `ProfileComposer@compose`，接下来我们来定义该 Composer 类：

```
<?php
```

```
namespace App\Http\ViewComposers;

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

class ProfileComposer
{
    /**
     * 用户仓库实现.
     *
     * @var UserRepository
     */
    protected $users;

    /**
     * 创建一个新的属性 composer.
     *
     * @param UserRepository $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        // Dependencies automatically resolved by service container...
        $this->users = $users;
    }

    /**
     * 绑定数据到视图.
     *
```

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

视图被渲染前，Composer 类的 compose 方法被调用，同时 `Illuminate\Contracts\View\View` 被注入该方法，从而可以使用其 `with` 方法来绑定数据到视图。

注意：所有视图 Composer 都通过服务容器被解析，所以你可以在 Composer 类的构造函数中声明任何你需要的依赖。

添加 Composer 到多个视图

你可以传递视图数组作为 `composer` 方法的第一个参数来一次性将视图 Composer 添加到多个视图：

```
view()->composer(
    ['profile', 'dashboard'],
    'App\Http\ViewComposers\MyViewComposer'
);
```

`composer` 方法接受 `*` 通配符，从而允许将一个 Composer 添加到所有视图：

```
view()->composer('*', function ($view) {
    //
});
```

4、视图创建器

视图创建器和视图 Composer 非常类似，不同之处在于前者在视图实例化之后立即失效而不是等到视图即将渲染。使用 `create` 方法即可注册一个视图创建器：

```
view()->creator('profile', 'App\Http\ViewCreators\ProfileCreator');
```

Blade 模板引擎

1、简介

Blade 是 Laravel 提供的一个非常简单但很强大的模板引擎，不同于其他流行的 PHP 模板引擎，Blade 在视图中并不约束你使用 PHP 原生代码。所有的 Blade 视图都会被编译成原生 PHP 代码并缓存起来直到被修改，这意味着对应用的性能而言 Blade 基本上是零开销。Blade 视图文件使用 `.blade.php` 文件扩展并存放在 `resources/views` 目录下。

2、模板继承

定义布局

使用 Blade 的两个最大优点是模板继承和切片，开始之前让我们先看一个例子。首先，我们检测一个“主”页面布局，由于大多数 Web 应用在不同页面中使用同一个布局，可以很方便的将这个布局定义为一个单独的 Blade 页面：

```
<!-- 存放在 resources/views/layouts/master.blade.php -->


<html>
    <head>
        <title>App Name - @yield('title')</title>
    </head>
    <body>
        @section('sidebar')
            This is the master sidebar.
        @show

        <div class="container">
            @yield('content')
        </div>
    </body>
</html>
```

正如你所看到的，该文件包含典型的 HTML 标记，然而，注意 `@section` 和 `@yield` 指令，前者正如其名字所暗示的，定义了一个内容的片段，而后者用于显示给定片段的内容。

现在我们已经为应用定义了一个布局，接下来让我们定义继承该布局的子页面吧。

扩展布局

定义子页面的时候，可以使用 Blade 的 `@extends` 指令来指定子页面所继承的布局，继承一个 Blade 布局的视图将会使用 `@section` 指令注入内容到布局的片段中，记住，如上面例子所示，这些片段的内容将会显示在布局中使用`@yield` 的地方：

```
<!-- 存放在 resources/views/child.blade.php -->

@extends('layouts.master')

@section('title', 'Page Title')

@section('sidebar')
    @parent

    <p>This is appended to the master sidebar.</p>
@endsection

@section('content')
    <p>This is my body content.</p>
@endsection
```

在本例中，`sidebar` 片段使用 `@parent` 指令来追加（而非覆盖）内容到布局中 `sidebar`，`@parent` 指令在视图渲染时将会被布局中的内容替换。
当然，和原生 PHP 视图一样，Blade 视图可以通过 `view` 方法直接从路由中返回：

```
Route::get('blade', function () {
   return view('child');
});
```

3、数据显示

可以通过两个花括号包裹变量来显示传递到视图的数据，比如，如果给出如下路由：

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

```
    return view('welcome', ['name' => 'Samantha']);
});
```

那么可以通过如下方式显示 `name` 变量的内容：

```
Hello, {{ $name }}.
```

当然，不限制显示到视图中的变量内容，你还可以输出任何 PHP 函数，实际上，可以将任何 PHP 代码放到 Blade 模板语句中：

```
The current UNIX timestamp is {{ time() }}.
```

注：Blade 的 `{{}}` 语句已经经过 PHP 的 `htmlentities` 函数处理以避免 XSS 攻击。

Blade & JavaScript 框架

由于很多 JavaScript 框架也是用花括号来表示要显示在浏览器中的表达式，可以使用 `@` 符号来告诉 Blade 渲染引擎该表达式应该保持原生格式不作改动。比如：

```
<h1>Laravel</h1>

Hello, @{{ name }}.
```

在本例中，`@` 符将会被 Blade 移除，然而，`{{ name }}` 表达式将会保持不变，避免被 JavaScript 框架渲染。

输出存在的数据

有时候你想要输出一个变量，但是不确定该变量是否被设置，我们可以通过如下 PHP 代码：

```
{{ isset($name) ? $name : 'Default' }}
```

除了使用三元运算符，Blade 还提供了更简单的方式：

```
{{ $name or 'Default' }}
```

在本例中，如果 `$name` 变量存在，其值将会显示，否则将会显示“Default”。

显示原生数据

默认情况下，Blade 的 `{{ }}` 语句已经通过 PHP 的 `htmlentities` 函数处理以避免 XSS 攻击，如果你不想要数据被处理，可以使用如下语法：

```
Hello, {!! $name !!}.
```

注：输出用户提供的内容时要当心，对用户提供的内容总是要使用双花括号包裹以避免直接输出 HTML 代码。

4、流程控制

除了模板继承和数据显示之外，Blade 还为常用的 PHP 流程控制提供了便利操作，比如条件语句和循环，这些快捷操作提供了一个干净、简单的方式来处理 PHP 的流程控制，同时保持和 PHP 相应语句的相似。

If 语句

可以使用 `@if`，`@elseif`，`@else` 和 `@endif` 来构造 if 语句，这些指令函数和 PHP 的相同：

```
@if (count($records) === 1)
    I have one record!
@elseif (count($records) > 1)
    I have multiple records!
@else
    I don't have any records!
@endif
```

为方便起见，Blade 还提供了 `@unless` 指令：

```
@unless (Auth::check())
    You are not signed in.
@endunless
```

循环

除了条件语句，Blade 还提供了简单指令处理 PHP 支持的循环结构，同样，这些指令函数和 PHP 的一样：

```
@for ($i = 0; $i < 10; $i++)
    The current value is {{ $i }}
@endfor

@foreach ($users as $user)
    <p>This is user {{ $user->id }}</p>
```

```
@endforeach


@forelse ($users as $user)
    <li>{{ $user->name }}</li>
    @empty
    <p>No users</p>
@endforelse


@while (true)
    <p>I'm looping forever.</p>
@endwhile
```

包含子视图

Blade 的 `@include` 指令允许你很简单的在一个视图中包含另一个 Blade 视图，所有父级视图中变量在被包含的子视图中依然有效：

```
<div>
    @include('shared.errors')


    <form>
        <!-- Form Contents -->
    </form>
</div>
```

尽管被包含的视图继承所有父视图中的数据，你还可以传递额外参数到被包含的视图：

```
@include('view.name', ['some' => 'data'])
```

注：不要在 Blade 视图中使用 `__DIR__` 和 `__FILE__` 常量，因为它们会指向缓存视图的路径。

为集合渲染视图

你可以使用 Blade 的 `@each` 指令通过一行代码循环引入多个局部视图：

```
@each('view.name', $jobs, 'job')
```

该指令的第一个参数是数组或集合中每个元素要渲染的局部视图，第二个参数是你希望迭代的数组或集合，第三个参数是要分配给当前视图的变量名。举个例子，如果你要迭代一个 `jobs` 数组，通常你需要在局部视图中访问 `$job` 变量。
你还可以传递第四个参数到 `@each` 指令，该参数用于指定给定数组为空时渲染的视图：

```
@each('view.name', $jobs, 'job', 'view.empty')
```

注释

Blade 还允许你在视图中定义注释，然而，不同于 HTML 注释，Blade 注释并不会包含到 HTML 中被返回：

```
{{-- This comment will not be present in the rendered HTML --}}
```

5、服务注入

`@inject` 指令可以用于从服务容器中获取服务，传递给 `@inject` 的第一个参数是服务将要被放置到的变量名，第二个参数是要解析的服务类名或接口名：

```
@inject('metrics', 'App\Services\MetricsService')


<div>
    Monthly Revenue: {{ $metrics->monthlyRevenue() }}.
</div>
```

6、扩展 Blade

Blade 甚至还允许你自定义指令，可以使用 `directive` 方法来注册一个指令。当 Blade 编译器遇到该指令，将会传入参数并调用提供的回调。
下面的例子创建了一个 `@datetime($var)` 指令格式化给定的 `$var`：

```
<?php


namespace App\Providers;


use Blade;
use Illuminate\Support\ServiceProvider;


class AppServiceProvider extends ServiceProvider
```

```
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Blade::directive('datetime', function($expression) {
            return "<?php echo with{$expression}->format('m/d/Y H:i'); ?>";
        });
    }

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

正如你所看到的，Laravel 的辅助函数 `with` 被用在该指令中，`with` 方法简单返回给定的对象/值，允许方法链。最终该指令生成的 PHP 代码如下：

```
<?php echo with($var)->format('m/d/Y H:i'); ?>
```

四、架构

一次请求的生命周期

1、简介

当我们使用现实世界中的任何工具时，如果理解了该工具的工作原理，那么用起来就会得心应手，应用开发也是如此。当你理解了开发工具如何工作，用起来就会更加游刃有余。本文档的目标就是从一个更好、更高层面向你阐述 Laravel 框架的工作原理。通过对框架更全面的了解，一切都不再那么神秘，你将会更加自信的构建应用。
如果你不能马上理解所有这些条款，不要失去信心！先试着掌握一些基本的东西，你的知识将会随着对本文档的探索而不断提高。

2、生命周期概览

第一件事

Laravel 应用的所有请求入口都是 `public/index.php` 文件，所有请求都会被 web 服务器（Apache/Nginx）导向这个文件。 `index.php` 文件包含的代码并不多，但是，这里是加载框架其它部分的起点。
`index.php` 文件载入 Composer 生成的自动加载设置，然后从 `bootstrap/app.php` 脚本获取 Laravel 应用实例，Laravel 的第一个动作就是创建服务容器实例。

HTTP/Console 内核

接下来，请求被发送到 HTTP 内核或 Console 内核，这取决于进入应用的请求类型。这两个内核是所有请求都要经过的中央处理器，现在，就让我们聚焦在位于 `app/Http/Kernel.php` 的 HTTP 内核。
HTTP 内核继承自 `Illuminate\Foundation\Http\Kernel` 类，该类定义了一个 `bootstrappers` 数组，这个数组中的类在请求被执行前运行，这些 `bootstrappers` 配置了错误处理、日志、检测应用环境以及其它在请求被处理前需要执行的任务。
HTTP 内核还定义了一系列所有请求在处理前需要经过的 HTTP 中间件，这些中间件处理 HTTP 会话的读写、判断应用是否处于维护模式、验证 CSRF 令牌等等。

HTTP 内核的标志性方法 `handle` 处理的逻辑相当简单：获取一个 Request，返回一个 Response，把该内核想象作一个代表整个应用的大黑盒子，输入 HTTP 请求，返回 HTTP 响应。
服务提供者
内核启动过程中最重要的动作之一就是为应用载入服务提供者，应用的所有服务提供者都被配置在 `config/app.php` 配置文件的 `providers` 数组中。首先，所有提供者的 `register` 方法被调用，然后，所有提供者被注册之后，`boot` 方法被调用。

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

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

3、聚焦服务提供者

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

对 Laravel 应用如何通过服务提供者构建和启动有一个牢固的掌握非常有价值，当然，应用默认的服务提供者存放在 `app/Providers` 目录下。
默认情况下，`AppServiceProvider` 是空的，这里是添加自定义启动和服务容器绑定的最佳位置，当然，对大型应用，你可能希望创建多个服务提供者，每一个都有着更加细粒度的启动。

应用目录结构

1、简介

Laravel 应用默认的目录结构试图为不管是大型应用还是小型应用提供一个好的起点，当然，你可以自己按照喜好重新组织应用目录结构，Laravel 对类在何处被加载没有任何限制——只要 Composer 可以自动载入它们即可。

2、根目录

新安装的 Laravel 应用包含许多文件夹：

`app` 目录包含了应用的核心代码；
`bootstrap` 目录包含了少许文件用于框架的启动和自动载入配置，还有一个 `cache` 文件夹用于包含框架生成的启动文件以提高性能；
`config` 目录包含了应用所有的配置文件；
`database` 目录包含了数据迁移及填充文件，如果你喜欢的话还可以将其作为 SQLite 数据库存放目录；
`public` 目录包含了前端控制器和资源文件（图片、JavaScript、CSS 等）；
`resources` 目录包含了视图文件及原生资源文件（LESS、SASS、CoffeeScript），以及本地化文件；
`storage` 目录包含了编译过的 Blade 模板、基于文件的 session、文件缓存，以及其它由框架生成的文件，该文件夹被细分为成 `app`、`framework` 和 `logs` 子目录，`app` 目录用于存放应用要使用的文件，`framework` 目录用于存放框架生成的文件和缓存，最后，`logs` 目录包含应用的日志文件；

`tests` 目录包含自动化测试，其中已经提供了一个开箱即用的 PHPUnit 示例；
`vendor` 目录包含 Composer 依赖；

3、App 目录

应用的核心代码位于 `app` 目录下，默认情况下，该目录位于命名空间 App 下， 并且被 Composer 通过 PSR-4 自动载入标准 自动加载。你可以通过 Artisan 命令 `app:name` 来修改该命名空间。
`app` 目录下包含多个子目录，如 `Console`、`Http`、`Providers` 等。`Console` 和 `Http` 目录提供了进入应用核心的 API，HTTP 协议和 CLI 是和应用进行交互的两种机制，但实际上并不包含应用逻辑。换句话说，它们只是两个向应用发布命令的方式。`Console` 目录包含了所有的 Artisan 命令，`Http` 目录包含了控制器、中间件和请求等。
`Jobs` 目录是放置队列任务的地方，应用中的任务可以被队列化，也可以在当前请求生命周期内同步执行。
`Events` 目录是放置事件类的地方，事件可以用于通知应用其它部分给定的动作已经发生，并提供灵活的解耦的处理。
`Listeners` 目录包含事件的处理器类，处理器接收一个事件并提供对该事件发生后的响应逻辑，比如，`UserRegistered` 事件可以被 `SendWelcomeEmail` 监听器处理。
`Exceptions` 目录包含应用的异常处理器，同时还是处理应用抛出的任何异常的好地方。
注意：`app` 目录中的很多类都可以通过 Artisan 命令生成，要查看所有有效的命令，可以在终端中运行 `php artisan list make` 命令。

服务提供者

1、简介

服务提供者是所有 Laravel 应用启动的中心，你自己的应用以及所有 Laravel 的核心服务都是通过服务提供者启动。
但是，我们所谓的“启动”指的是什么？通常，这意味着注册事物，包括注册服务容器绑定、事件监听器、中间件甚至路由。服务提供者是应用配置的中心。
如果你打开 Laravel 自带的 `config/app.php` 文件，将会看到一个 `providers` 数组，这里就是应用所要加载的所有服务提供者类，当然，其中很多是延迟加载的，也就是说不是每次请求都会被加载，只有真的用到它们的时候才会加载。
本章里你将会学习如何编写自己的服务提供者并在 Laravel 应用中注册它们。

2、编写服务提供者

所有的服务提供者继承自 `Illuminate\Support\ServiceProvider` 类。继承该抽象类要求至少在服务提供者中定义一个方法：`register`。在 `register` 方法内，你唯一要做的事情就是绑事物到服务容器，不要尝试在其中注册任何时间监听器，路由或者任何其它功能。
通过 Artisan 命令 `make:provider` 可以简单生成一个新的提供者：

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

register 方法 向容器注册“脚本”

正如前面所提到的，在 `register` 方法中只绑定事物到服务容器，而不要做其他事情，否则话，一不小心就能用到一个尚未被加载的服务提供者提供的服务。
现在让我们来看看一个基本的服务提供者长什么样：

```
<?php

namespace App\Providers;

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

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

该服务提供者只定义了一个 `register` 方法，并使用该方法在服务容器中定义了一个 `Riak\Contracts\Connection` 的实现。

boot 方法

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

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class EventServiceProvider extends ServiceProvider{
```

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

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

boot 方法的依赖注入

我们可以在 `boot` 方法中类型提示依赖，服务容器会自动注册你所需要的依赖：

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

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

3、注册服务提供者

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

要注册你自己的服务提供者，只需要将其追加到该数组中即可：

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

```
],
```

4、延迟加载服务提供者

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

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

```
<?php

namespace App\Providers;

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

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

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

    /**
     * 获取由提供者提供的服务.
     *
     * @return array
     */
```

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

}
```

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

服务容器

1、简介

Laravel 服务容器是一个用于管理类依赖和执行依赖注入的强大工具。依赖注入听上去很花哨，其实质是通过构造函数或者某些情况下通过 `set` 方法将类依赖注入到类中。
让我们看一个简单的例子：

```
<?php

namespace App\Jobs;

use App\User;
use Illuminate\Contracts\Mail\Mailer;
use Illuminate\Contracts\Bus\SelfHandling;

class PurchasePodcast implements SelfHandling{
    /**
     * 邮件实现
     */
    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
     *
     * @return void
     */
    public function handle()
    {
        //
    }
}
```

在本例中，当播客被购买后 `PurchasePodcast` 任务需要发送邮件，因此，你需要注入一个可以发送邮件的服务。由于该服务是被注入的，我们可以方便的使用其另一个实现来替换它，在测试的时候我们还可以“模拟”或创建一个假的邮件实现。
深入理解 Laravel 服务容器对于构建功能强大的大型 Laravel 应用而言至关重要，对于贡献代码到 Laravel 核心也很有帮助。

2、绑定

几乎所有的服务容器绑定都是在服务提供者中完成。因此本章节的演示例子用到的容器都是在这种上下文环境中，如果一个类没有基于任何接口那么就没有必要将其绑定到容器。容器并不需要被告知如何构建对象，因为它会使用 PHP 的反射服务自动解析出具体的对象。
在一个服务提供者中，可以通过 `$this->app` 变量访问容器，然后使用 `bind` 方法注册一个绑定，该方法需要两个参数，第一个参数是我们想要注册的类名或接口名称，第二个参数是返回类的实例的闭包：

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

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

绑定一个单例

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

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

绑定实例

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

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

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

绑定接口到实现

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

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

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

```
use App\Contracts\EventPusher;

/**
 * 创建一个新的类实例
 *
 * @param  EventPusher  $pusher
 * @return void
 */
public function __construct(EventPusher $pusher){
    $this->pusher = $pusher;
}
```

上下文绑定

有时候我们可能有两个类使用同一个接口，但我们希望在每个类中注入不同实现，例如，当系统接到一个新的订单的时候，我们想要通过 PubNub 而不是 Pusher 发送一个事件。Laravel 定义了一个简单、平滑的方式来定义这种行为：

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

你甚至还可以传递一个闭包到 `give` 方法：

```
$this->app->when('App\Handlers\Commands\CreateOrderHandler')
    ->needs('App\Contracts\EventPusher')
    ->give(function () {
        // Resolve dependency...
    });
```

绑定原始值

有时候你可能有一个获取若干注入类的类，但还需要一个注入的原始值，比如整型数据，你可以轻松使用上下文绑定来注入指定类所需要的任何值：

```
$this->app->when('App\Handlers\Commands\CreateOrderHandler')
    ->needs('$maxOrderCount')
    ->give(10);
```

标签

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

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

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

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

这些服务被打上标签后，可以通过 `tagged` 方法来轻松解析它们：

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

3、解析

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

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

其次，你可以以数组方式访问容器，因为其实现了 PHP 的 `ArrayAccess` 接口：

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

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

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

```
<?php

namespace App\Http\Controllers;

use Illuminate\Routing\Controller;
```

```
use App\Users\Repository as UserRepository;

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

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

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

4、容器事件

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

```
$this->app->resolving(function ($object, $app) {
    // 容器解析所有类型对象时调用
});

$this->app->resolving(function (FooBar $fooBar, $app) {
    // 容器解析“FooBar”对象时调用
});
```

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

门面（Facades）

1、简介

门面为应用的服务容器中的绑定类提供了一个“静态”接口。Laravel 内置了很多门面，你可能在不知道的情况下正在使用它们。Laravel 的门面作为服务容器中的底层类的“静态代理”，相比于传统静态方法，在维护时能够提供更加易于测试、更加灵活的、简明且富有表现力的语法。

```
'aliases' => [
    ...//其他门面类别名映射
    'TestClass' => App\Facades\TestClass::class,
],
```

facadeb编写完成后，需要进行配置：config/app.php中注册门面类别名

2、使用门面

在 Laravel 应用的上下文中，门面就是一个提供访问容器中对象的类。该机制原理由 Facade 类实现，Laravel 自带的门面，以及创建的自定义门面，都会继承自 Illuminate\Support\Facades\Facade 基类。

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

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

```
<?php

namespace App\Http\Controllers;

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

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

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

```
    }
}
```

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

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

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

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

3、门面类列表

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

门面	类	服务容器绑定别
App	Illuminate\Foundation\Application	`app`
Artisan	Illuminate\Console\Application	`artisan`
Auth	Illuminate\Auth\AuthManager	`auth`
Auth (Instance)	Illuminate\Auth\Guard	
Blade	Illuminate\View\Compilers\BladeCompiler	`blade.compiler`
Bus	Illuminate\Contracts\Bus\Dispatcher	
Cache	Illuminate\Cache\Repository	`cache`

门面	类	服务容器绑定别
Config	Illuminate\Config\Repository	config
Cookie	Illuminate\Cookie\CookieJar	cookie
Crypt	Illuminate\Encryption\Encrypter	encrypter
DB	Illuminate\Database\DatabaseManager	db
DB (Instance)	Illuminate\Database\Connection	
Event	Illuminate\Events\Dispatcher	events
File	Illuminate\Filesystem\Filesystem	files
Hash	Illuminate\Contracts\Hashing\Hasher	hash
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 (Instance)	Illuminate\Queue\QueueInterface	
Queue (Base Class)	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	

门面	类	服务容器绑定别
Session	Illuminate\Session\SessionManager	`session`
Session (Instance)	Illuminate\Session\Store	
Storage	Illuminate\Contracts\Filesystem\Factory	`filesystem`
URL	Illuminate\Routing\UrlGenerator	`url`
Validator	Illuminate\Validation\Factory	`validator`
Validator (Instance)	Illuminate\Validation\Validator	
View	Illuminate\View\Factory	`view`
View (Instance)	Illuminate\View\View	

五、数据库

起步

1、简介

Laravel 让连接多种数据库以及对数据库进行查询变得非常简单，不论使用原生 SQL、还是查询构建器，还是 Eloquent ORM。目前，Laravel 支持四种类型的数据库系统：

- MySQL
- Postgres
- SQLite
- SQL Server

配置

Laravel 让连接数据库和运行查询都变得非常简单。应用的数据库配置位于 `config/database.php`。在该文件中你可以定义所有的数据库连接，并指定哪个连接是默认连接。该文件中提供了所有支持数据库系统的配置示例。
默认情况下，Laravel 示例环境配置已经为 Laravel Homestead 做好了设置，当然，你也可以按照需要为本地的数据库修改该配置。

读/写连接

有时候你希望使用一个数据库连接做查询，另一个数据库连接做插入、更新和删除，Laravel 使得这件事情轻而易举，不管你用的是原生 SQL，还是查询构建器，还是 Eloquent ORM，合适的连接总是会被使用。
想要知道如何配置读/写连接，让我们看看下面这个例子：

```
'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'    => '',
],
```

注意我们在配置数组中新增了两个键：`read` 和 `write`，这两个键都对应一个包含单个键“host”的数组，读/写连接的其它数据库配置选项都共用 mysql 的主数组配置。
如果我们想要覆盖主数组中的配置，只需要将相应配置项放到 `read` 和 `write` 数组中即可。在本例中，`192.168.1.1` 将被用作“读”连接，而 `192.168.1.2` 将被用作“写”连接。两个数据库连接的凭证（用户名/密码）、前缀、字符集以及其它配置将会共享 `mysql` 数组中的设置。

2、运行原生 SQL 查询

配置好数据库连接后，就可以使用 DB 门面来运行查询。DB 门面为每种查询提供了相应方法：`select`, `update`, `insert`, `delete`, 和 `statement`。

运行 Select 查询

运行一个最基本的查询，可以使用 `DB` 门面的 `select` 方法：

```
<?php

namespace App\Http\Controllers;

use DB;
use App\Http\Controllers\Controller;
```

```
class UserController extends Controller{
    /**
     * 显示用户列表
     *
     * @return Response
     */
    public function index()
    {
        $users = DB::select('select * from users where active =
 ?', [1]);
        return view('user.index', ['users' => $users]);
    }
}
```

传递给 `select` 方法的第一个参数是原生的 SQL 语句，第二个参数需要绑定到查询的参数绑定，通常，这些都是 `where` 字句约束中的值。参数绑定可以避免 SQL 注入攻击。`select` 方法以数组的形式返回结果集，数组中的每一个结果都是一个 PHP `StdClass` 对象，从而允许你像下面这样访问结果值：

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

使用命名绑定

除了使用`?`占位符来代表参数绑定外，还可以使用命名绑定来执行查询：

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

运行插入语句

使用 `DB` 门面的 `insert` 方法执行插入语句。和 `select` 一样，改方法将原生 SQL 语句作为第一个参数，将绑定作为第二个参数：

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

运行更新语句

`update` 方法用于更新数据库中已存在的记录，该方法返回受更新语句影响的行数：

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

运行删除语句

`delete` 方法用于删除数据库中已存在的记录，和 `update` 一样，该语句返回被删除的行数：

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

运行一个通用语句

有些数据库语句不返回任何值，对于这种类型的操作，可以使用 `DB` 门面的 `statement` 方法：

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

监听查询事件

如果你想要获取应用中每次 SQL 语句的执行，可以使用 `listen` 方法，该方法对查询日志和调试非常有用，你可以在服务提供者中注册查询监听器：

```
<?php

namespace App\Providers;

use DB;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider{
    /**
     * 启动所有应用服务
     *
     * @return void
     */
    public function boot()
    {
        DB::listen(function($sql, $bindings, $time) {
            //
        });
    }

    /**
     * 注册服务提供者
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

3、数据库事务

想要在一个数据库事务中运行一连串操作，可以使用 DB 门面的 transaction 方法，如果事务闭包中抛出异常，事务将会自动回滚。如果闭包执行成功，事务将会自动提交。使用 transaction 方法时不需要担心手动回滚或提交：

```
DB::transaction(function () {
    DB::table('users')->update(['votes' => 1]);
    DB::table('posts')->delete();
});
```

手动使用事务

如果你想要手动开始事务从而对回滚和提交有一个完整的控制，可以使用 DB 门面的 beginTransaction 方法：

```
DB::beginTransaction();
```

你可以通过 rollBack 方法回滚事务：

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

最后，你可以通过 commit 方法提交事务：

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

注意：使用 DB 门面的事务方法还可以用于控制查询构建器和 Eloquent ORM 的事务。

4、使用多个数据库连接

使用多个数据库连接的时候，可以使用 DB 门面的 connection 方法访问每个连接。传递给 connection 方法的连接名对应配置文件 config/database.php 中相应的连接：

```
$users = DB::connection('foo')->select(...);
```

你还可以通过连接实例上的 getPdo 方法底层原生的 PDO 实例：

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

查询构建器

1、简介

数据库查询构建器提供了一个方便的、平滑的接口来创建和运行数据库查询。查询构建器可以用于执行应用中大部分数据库操作，并且能够在支持的所有数据库系统上工作。

注意：Laravel 查询构建器使用 PDO 参数绑定来避免 SQL 注入攻击，不再需要过滤传递到绑定的字符串。

2、获取结果集

从一张表中取出所有行

在查询之前，使用 DB 门面的 table 方法，table 方法为给定表返回一个查询构建器，允许你在查询上链接更多约束条件并最终返回查询结果。在本例中，我们使用 get 方法获取表中所有记录：

```
<?php

namespace App\Http\Controllers;

use DB;
use App\Http\Controllers\Controller;

class UserController extends Controller{
    /**
     * 显示用户列表
     *
     * @return Response
     */
    public function index()
    {
        $users = DB::table('users')->get();

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

和原生查询一样，get 方法返回结果集的数组，其中每一个结果都是 PHP 对象的 StdClass 实例。你可以像访问对象的属性一样访问列的值：

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

从一张表中获取一行/一列

如果你只是想要从数据表中获取一行数据，可以使用 first 方法，该方法将会返回单个 StdClass 对象：

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

如果你不需要完整的一行，可以使用 value 方法从结果中获取单个值，该方法会直接返回指定列的值：

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

从一张表中获取组块结果集

如果你需要处理成千上百条数据库记录，可以考虑使用 `chunk` 方法，该方法一次获取结果集的一小块，然后填充每一小块数据到要处理的闭包，该方法在编写处理大量数据库记录的 Artisan 命令的时候非常有用。比如，我们可以将处理全部 users 表数据处理成一次处理 100 记录的小组块：

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

你可以通过从闭包函数中返回 false 来中止组块的运行：

```
DB::table('users')->chunk(100, function($users) {
    // 处理结果集...
    return false;
});
```

获取数据列值列表

如果想要获取包含单个列值的数组，可以使用 `lists` 方法，在本例中，我们获取所有 `title` 的数组：

```
$titles = DB::table('roles')->lists('title');

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

在还可以在返回数组中为列值指定更多的自定义键（该自定义键必须是该表的其它字段列名，否则会报错）：

```
$roles = DB::table('roles')->lists('title', 'name');

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

聚合函数

队列构建器还提供了很多聚合方法，比如 `count`, `max`, `min`, `avg`, 和 `sum`，你可以在构造查询之后调用这些方法：

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

当然，你可以联合其它查询子句和聚合函数来构建查询：

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

3、查询（Select）

指定查询子句

当然，我们并不总是想要获取数据表的所有列，使用 `select` 方法，你可以为查询指定自定义的 `select` 子句：

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

`distinct` 方法允许你强制查询返回不重复的结果集：

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

如果你已经有了一个查询构建器实例并且希望添加一个查询列到已存在的 `select` 子句，可以使用 `addSelect` 方法：

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

原生表达式

有时候你希望在查询中使用原生表达式，这些表达式将会以字符串的形式注入到查询中，所以要格外小心避免被 SQL 注入。想要创建一个原生表达式，可以使用 `DB::raw` 方法：

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

4、连接（Join）

内连接（等值连接）

查询构建器还可以用于编写基本的 SQL“内连接”，你可以使用查询构建器实例上的 join 方法，传递给 join 方法的第一次参数是你需要连接到的表名，剩余的其它参数则是为连接指定的列约束，当然，正如你所看到的，你可以在单个查询中连接多张表：

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

左连接

如果你是想要执行“左连接”而不是“内连接”，可以使用 leftJoin 方法。该方法和 join 方法的使用方法一样：

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

高级连接语句

你还可以指定更多的高级连接子句，传递一个闭包到 join 方法作为该方法的第 2 个参数，该闭包将会返回允许你指定 join 子句约束的 JoinClause 对象：

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

如果你想要在连接中使用“where”风格的子句，可以在查询中使用 where 和 orWhere 方法。这些方法将会将列和值进行比较而不是列和列进行比较：

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

5、联合（Union）

查询构建器还提供了一条“联合”两个查询的快捷方式，比如，你要创建一个独立的查询，然后使用 union 方法将其和第二个查询进行联合：

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

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

unionAll 方法也是有效的，并且和 union 有同样的使用方法。

6、Where 子句

简单 where 子句

使用查询构建器上的 where 方法可以添加 where 子句到查询中，调用 where 最基本的方法需要三个参数，第一个参数是列名，第二个参数是一个数据库系统支持的任意操作符，第三个参数是该列要比较的值。
例如，下面是一个验证“votes”列的值是否等于 100 的查询：

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

为了方便，如果你只是简单比较列值和给定数值是否相等，可以将数值直接作为 where 方法的第二个参数：

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

当然，你可以使用其它操作符来编写 where 子句：

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

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

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

or

你可以通过方法链将多个 `where` 约束链接到一起，也可以添加 `or` 子句到查询，`orWhere` 方法和 `where` 方法接收参数一样：

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

更多 Where 子句

whereBetween

`whereBetween` 方法验证列值是否在给定值之间：

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

whereNotBetween

`whereNotBetween` 方法验证列值不在给定值之间：

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

whereIn/whereNotIn

`whereIn` 方法验证给定列的值是否在给定数组中：

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

`whereNotIn` 方法验证给定列的值不在给定数组中：

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

whereNull/whereNotNull

`whereNull` 方法验证给定列的值为 NULL：

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

`whereNotNull` 方法验证给定列的值不是 NULL：

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

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

高级 Where 子句

参数分组

有时候你需要创建更加高级的 `where` 子句比如“where exists”或者嵌套的参数分组。Laravel 查询构建器也可以处理这些。作为开始，让我们看一个在括号中进行分组约束的例子：

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

正如你所看到的，传递闭包到 `orWhere` 方法构造查询构建器来开始一个约束分组，，该闭包将会获取一个用于设置括号中包含的约束的查询构建器实例。上述语句等价于下面的 SQL：

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

exists 语句

`whereExists` 方法允许你编写 `where exist`SQL 子句，`whereExists` 方法接收一个闭包参数，该闭包获取一个查询构建器实例从而允许你定义放置在“exists”子句中的查询：

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

上述查询等价于下面的 SQL 语句：

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

7、排序、分组、限定

orderBy

`orderBy` 方法允许你通过给定列对结果集进行排序，`orderBy` 的第一个参数应该是你希望排序的列，第二个参数控制着排序的方向——asc 或 desc：

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

groupBy / having / havingRaw

groupBy 和 having 方法用于对结果集进行分组，having 方法和 where 方法的用法类似：

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

havingRaw 方法可以用于设置原生字符串作为 having 子句的值，例如，我们要找到所有售价大于$2500 的部分：

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

skip / take

想要限定查询返回的结果集的数目，或者在查询中跳过给定数目的结果，可以使用 skip 和 take 方法：

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

8、插入（Insert）

查询构建器还提供了 insert 方法来插入记录到数据表。insert 方法接收数组形式的列名和值进行插入操作：

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

你甚至可以一次性通过传入多个数组来插入多条记录，每个数组代表要插入数据表的记录：

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

自增 ID

如果数据表有自增 ID，使用 insertGetId 方法来插入记录将会返回 ID 值：

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

注意：当使用 PostgresSQL 时 insertGetId 方法默认自增列被命名为 id，如果你想要从其他“序列”获取 ID，可以将序列名作为第二个参数传递到 insertGetId 方法。

9、更新（Update）

当然，除了插入记录到数据库，查询构建器还可以通过使用 update 方法更新已有记录。update 方法和 insert 方法一样，接收列和值的键值对数组包含要更新的列，你可以通过 where 子句来对 update 查询进行约束：

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

增加/减少

查询构建器还提供了方便增减给定列名数值的方法。相较于编写 update 语句，这是一条捷径，提供了更好的体验和测试接口。
这两个方法都至少接收一个参数：需要修改的列。第二个参数是可选的，用于控制列值增加/减少的数目。

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

在操作过程中你还可以指定额外的列进行更新：

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

10、删除（Delete）

当然，查询构建器还可以通过 delete 方法从表中删除记录：

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

在调用 delete 方法之前可以通过添加 where 子句对 delete 语句进行约束：

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

如果你希望清除整张表，也就是删除所有列并将自增 ID 置为 0，可以使用 truncate 方法：

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

11、悲观锁

查询构建器还包含一些方法帮助你在 select 语句中实现“悲观锁”。可以在查询中使用 sharedLock 方法从而在运行语句时带一把”共享锁“。共享锁可以避免被选择的行被修改直到事务提交：

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

此外你还可以使用 lockForUpdate 方法。“for update”锁避免选择行被其它共享锁修改或删除：

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

迁移

1、简介

迁移就像数据库的版本控制，允许团队简单轻松的编辑并共享应用的数据库表结构，迁移通常和 Laravel 的结构构建器结对从而可以很容易地构建应用的数据库表结构。Laravel 的 Schema 门面提供了与数据库系统无关的创建和操纵表的支持，在 Laravel 所支持的所有数据库系统中提供一致的、优雅的、平滑的 API。

2、生成迁移

使用 Artisan 命令 make:migration 来创建一个新的迁移：

```
php artisan make:migration create_users_table
```

新的迁移位于 database/migrations 目录下，每个迁移文件名都包含时间戳从而允许 Laravel 判断其顺序。

--table 和 --create 选项可以用于指定表名以及该迁移是否要创建一个新的数据表。这些选项只需要简单放在上述迁移命令后面并指定表名：

```
php artisan make:migration add_votes_to_users_table --table=users
php artisan make:migration create_users_table --create=users
```

如果你想要指定生成迁移的自定义输出路径，在执行 make:migration 命令时可以使用 --path 选项，提供的路径应该是相对于应用根目录的。

3、迁移结构

迁移类包含了两个方法：`up` 和 `down`。`up` 方法用于新增表，列或者索引到数据库，而 `down` 方法就是 `up` 方法的反操作，和 `up` 里的操作相反。
在这两个方法中你都要用到 Laravel 的表结构构建器来创建和修改表，例如，让我们先看看创建 `flights` 表的简单示例：

```
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateFlightsTable extends Migration{
    /**
     * 运行迁移
     *
     * @return void
     */
    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }

    /**
     * 撤销迁移
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('flights');
    }
}
```

4、运行迁移

要运行应用中所有未执行的迁移，可以使用 Artisan 命令的 `migrate` 方法。如果你正在使用 Homestead 虚拟机，应该在你的虚拟机中运行如下这条命令：

```
php artisan migrate
```

如果再运行时遇到“class not found”的错误提示，尝试运行 `composer dump-autoload` 命令然后重新运行迁移命令。

在生产环境中强制运行迁移

有些迁移操作是毁灭性的，这意味着它们可能造成数据的丢失，为了避免在生产环境数据库中运行这些命令，你将会在运行这些命令之前被提示并确认。想要强制运行这些命令而不被提示，可以使用`--force`：

```
php artisan migrate --force
```

回滚迁移

想要回滚最新的一次迁移”操作“，可以使用 `rollback` 命令，注意这将会回滚最后一批运行的迁移，可能包含多个迁移文件：

```
php artisan migrate:rollback
```

`migrate:reset` 命令将会回滚所有的应用迁移：

```
php artisan migrate:reset
```

在单个命令中回滚/迁移

`migrate:refresh` 命令将会先回滚所有数据库迁移，然后运行 `migrate` 命令。这个命令可以有效的重建整个数据库：

```
php artisan migrate:refresh
php artisan migrate:refresh --seed
```

5、编写迁移

创建表

使用 `Schema` 门面上的 `create` 方法来创建新的数据表。`create` 方法接收两个参数，第一个是表名，第二个是获取用于定义新表的 `Blueprint` 对象的闭包：

```
Schema::create('users', function ($table) {
    $table->increments('id');
});
```

当然，创建新表的时候，可以使用表结构构建器中的任意列方法来定义数据表的列。

检查表/列是否存在

你可以轻松地使用 `hasTable` 和 `hasColumn` 方法检查表或列是否存在：

```
if (Schema::hasTable('users')) {
    //
}

if (Schema::hasColumn('users', 'email')) {
```

```
    //
}
```

连接&存储引擎

如果你想要在一个数据库连接上执行表结构操作，该数据库连接并不是默认数据库连接，使用 `connection` 方法：

```
Schema::connection('foo')->create('users', function ($table) {
    $table->increments('id');
});
```

要设置表的存储引擎，在表结构构建器上设置 `engine` 属性：

```
Schema::create('users', function ($table) {
    $table->engine = 'InnoDB';
    $table->increments('id');
});
```

重命名/删除表

要重命名一个已存在的数据表，使用 `rename` 方法：

```
Schema::rename($from, $to);
```

要删除一个已存在的数据表，可以使用 `drop` 或 `dropIfExists` 方法：

```
Schema::drop('users');

Schema::dropIfExists('users');
```

创建列

要更新一个已存在的表，使用 Schema 门面上的 `table` 方法，和 `create` 方法一样，`table` 方法接收两个参数：表名和获取用于添加列到表的 `Blueprint` 实例的闭包：

```
Schema::table('users', function ($table) {
    $table->string('email');
});
```

可用的列类型

当然，表结构构建器包含一系列你可以用来构建表的列类型：

命令	描述
`$table->bigIncrements('id');`	自增 ID，类型为 bigint
`$table->bigInteger('votes');`	等同于数据库中的 BIGINT 类型
`$table->binary('data');`	等同于数据库中的 BLOB 类型

命令	描述
`$table->boolean('confirmed');`	等同于数据库中的 BOOLEAN 类型
`$table->char('name', 4);`	等同于数据库中的 CHAR 类型
`$table->date('created_at');`	等同于数据库中的 DATE 类型
`$table->dateTime('created_at');`	等同于数据库中的 DATETIME 类型
`$table->decimal('amount', 5, 2);`	等同于数据库中的 DECIMAL 类型，带一个精度和范围
`$table->double('column', 15, 8);`	等同于数据库中的 DOUBLE 类型，带精度，总共 15 位数字，后 8 位.
`$table->enum('choices', ['foo', 'bar']);`	等同于数据库中的 ENUM 类型
`$table->float('amount');`	等同于数据库中的 FLOAT 类型
`$table->increments('id');`	数据库主键自增 ID
`$table->integer('votes');`	等同于数据库中的 INTEGER 类型
`$table->json('options');`	等同于数据库中的 JSON 类型
`$table->jsonb('options');`	等同于数据库中的 JSONB 类型
`$table->longText('description');`	等同于数据库中的 LONGTEXT 类型
`$table->mediumInteger('numbers');`	等同于数据库中的 MEDIUMINT 类型
`$table->mediumText('description');`	等同于数据库中的 MEDIUMTEXT 类型
`$table->morphs('taggable');`	添加一个 INTEGER 类型的 `taggable_id` 列和一个 STRIN 的 `taggable_type` 列
`$table->nullableTimestamps();`	和 `timestamps()`一样但允许 NULL 值.
`$table->rememberToken();`	添加一个 `remember_token` 列： VARCHAR(100) NULL.
`$table->smallInteger('votes');`	等同于数据库中的 SMALLINT 类型

命令	描述
`$table->softDeletes();`	新增一个 `deleted_at` 列 用于软删除.
`$table->string('email');`	等同于数据库中的 VARCHAR 列 .
`$table->string('name', 100);`	等同于数据库中的 VARCHAR，带一个长度
`$table->text('description');`	等同于数据库中的 TEXT 类型
`$table->time('sunrise');`	等同于数据库中的 TIME 类型
`$table->tinyInteger('numbers');`	等同于数据库中的 TINYINT 类型
`$table->timestamp('added_on');`	等同于数据库中的 TIMESTAMP 类型
`$table->timestamps();`	添加 `created_at` 和 `updated_at` 列.
`$table->uuid('id');`	等同于数据库的 UUID

列修改器

除了上面列出的列类型之外，在添加列的时候还可以使用一些其它列“修改器”，例如，要使列默认为 `null`，可以使用 `nullable` 方法：

```
Schema::table('users', function ($table) {
    $table->string('email')->nullable();
});
```

下面是所有可用的列修改器列表，该列表不包含索引修改器：

修改器	描述
`->first()`	将该列置为表中第一个列 (仅适用于 MySQL)
`->after('column')`	将该列置于另一个列之后 (仅适用于 MySQL)
`->nullable()`	允许该列的值为 NULL
`->default($value)`	指定列的默认值
`->unsigned()`	设置 `integer` 列为 `UNSIGNED`

修改列

先决条件

在修改列之前，确保已经将 doctrine/dbal 依赖添加到 composer.json 文件，Doctrine DBAL 库用于判断列的当前状态并在需要时创建 SQL 查询来对列进行指定的调整。

更新列属性

change 方法允许你修改已存在的列为新的类型，或者修改列的属性。例如，你可能想要增加 string 类型列的尺寸，让我们将 name 列的尺寸从 25 增加到 50：

```
Schema::table('users', function ($table) {
    $table->string('name', 50)->change();
});
```

我们还可以修改该列允许 NULL 值：

```
Schema::table('users', function ($table) {
    $table->string('name', 50)->nullable()->change();
});
```

重命名列

要重命名一个列，可以使用表结构构建器上的 renameColumn 方法，在重命名一个列之前，确保 doctrine/dbal 依赖已经添加到 composer.json 文件：

```
Schema::table('users', function ($table) {
    $table->renameColumn('from', 'to');
});
```

注意：暂不支持 enum 类型的列的重命名。

删除列

要删除一个列，使用表结构构建器上的 dropColumn 方法：

```
Schema::table('users', function ($table) {
    $table->dropColumn('votes');
});
```

你可以传递列名数组到 dropColumn 方法从表中删除多个列：

```
Schema::table('users', function ($table) {

    $table->dropColumn(['votes', 'avatar', 'location']);

});
```

注意：在从 SQLite 数据库删除列之前，需要添加 doctrine/dbal 依赖到 composer.json 文件并在终端中运行 composer update 命令来安装该库。此外，SQLite 数据库暂不支持在单个迁移中删除或修改多个列。

创建索引

表结构构建器支持多种类型的索引，首先，让我们看一个指定列值为唯一索引的例子。要创建索引，可以使用 unique 方法：

```
$table->string('email')->unique();
```

此外，你可以在定义列之后创建索引，例如：

```
$table->unique('email');
```

你甚至可以传递列名数组到索引方法来创建组合索引：

```
$table->index(['account_id', 'created_at']);
```

Laravel 会自动生成合理的索引名称，但是你可以传递第二个参数到该方法用于指定索引名称：

```
$table->index('email', 'my_index_name');
```

可用索引类型

命令	描述
`$table->primary('id');`	添加主键索引
`$table->primary(['first', 'last']);`	添加混合索引
`$table->unique('email');`	添加唯一索引
`$table->unique('state', 'my_index_name');`	指定自定义索引名称
`$table->index('state');`	添加普通索引

删除索引

要删除索引，必须指定索引名。默认情况下，Laravel 自动分配适当的名称给索引——简单连接表名、列名和索引类型。下面是一些例子：

命令	描述
`$table->dropPrimary('users_id_primary');`	从 “users”表中删除主键索引
`$table->dropUnique('users_email_unique');`	从 “users”表中删除唯一索引
`$table->dropIndex('geo_state_index');`	从 “geo”表中删除普通索引

外键约束

Laravel 还提供了创建外键约束的支持，用于在数据库层面强制引用完整性。例如，我们在 `posts` 表中定义了一个引用 `users` 表的 `id` 列的 `user_id` 列：

```
Schema::table('posts', function ($table) {
```

```
    $table->integer('user_id')->unsigned();
    $table->foreign('user_id')->references('id')->on('users');
});
```

你还可以为约束的“on delete”和“on update”属性指定期望的动作：

```
$table->foreign('user_id')
      ->references('id')->on('users')
      ->onDelete('cascade');
```

要删除一个外键，可以使用 `dropForeign` 方法。外键约束和索引使用同样的命名规则——连接表名、外键名然后加上“_foreign”后缀：

```
$table->dropForeign('posts_user_id_foreign');
```

填充数据

1、简介

Laravel 包含了一个简单方法来填充数据库——使用填充类和测试数据。所有的填充类都位于 `database/seeds` 目录。填充类的类名完全由你自定义，但最好还是遵循一定的规则，比如可读性，例如 `UserTableSeeder` 等等。安装完 Laravel 后，会默认提供一个 `DatabaseSeeder` 类。从这个类中，你可以使用 `call` 方法来运行其他填充类，从而允许你控制填充顺序。

2、编写填充器

要生成一个填充器，可以通过 Artisan 命令 `make:seeder`。所有框架生成的填充器都位于 `database/seeders` 目录：

```
php artisan make:seeder UserTableSeeder
```

一个填充器类默认只包含一个方法：`run`。当 Artisan 命令 `db:seed` 运行时该方法被调用。在 `run` 方法中，可以插入任何你想插入数据库的数据，你可以使用查询构建器手动插入数据，也可以使用 Eloquent 模型工厂。

举个例子，让我们修改 Laravel 安装时自带的 `DatabaseSeeder` 类，添加一个数据库插入语句到 `run` 方法：

```
<?php

use DB;
use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;

class DatabaseSeeder extends Seeder{
    /**
```

```
     * 运行数据库填充
     *
     * @return void
     */
    public function run()
    {
        DB::table('users')->insert([
            'name' => str_random(10),
            'email' => str_random(10).'@gmail.com',
            'password' => bcrypt('secret'),
        ]);
    }
}
```

使用模型工厂

当然，手动指定每一个模型填充的属性是很笨重累赘的，取而代之的，我们可以使用模型工厂来方便的生成大量的数据库记录。首先，查看模型工厂文档来学习如何定义工厂，定义工厂后，可以使用帮助函数 `factory` 来插入记录到数据库。
举个例子，让我们创建 50 个用户并添加关联关系到每个用户：

```
/**
 * 运行数据库填充
 *
 * @return void
 */
public function run(){
    factory('App\User', 50)->create()->each(function($u) {
        $u->posts()->save(factory('App\Post')->make());
    });
}
```

调用额外的填充器

在 `DatabaseSeeder` 类中，你可以使用 `call` 方法执行额外的填充类，使用 `call` 方法允许你将数据库填充分解成多个文件，这样单个填充器类就不会变得无比巨大，只需简单将你想要运行的填充器类名传递过去即可：

```
/**

* 运行数据库填充

*

* @return void

*/
```

```
public function run(){

    Model::unguard();


    $this->call(UserTableSeeder::class);

    $this->call(PostsTableSeeder::class);

    $this->call(CommentsTableSeeder::class);


    Model::reguard();

}
```

3、运行填充器

编写好填充器类之后，可以使用 Artisan 命令 `db:seed` 来填充数据库。默认情况下，`db:seed` 命令运行可以用来运行其它填充器类的 `DatabaseSeeder` 类，但是，你也可以使用 `--class` 选项来指定你想要运行的独立的填充器类：

```
php artisan db:seed
php artisan db:seed --class=UserTableSeeder
```

你还可以使用 `migrate:refresh` 命令来填充数据库，该命令还可以回滚并重新运行迁移，这在需要完全重建数据库时很有用：

```
php artisan migrate:refresh --seed
```

六、Eloquent ORM

起步

1、简介

Laravel 自带的 Eloquent ORM 提供了一个美观、简单的与数据库打交道的 ActiveRecord 实现，每张数据表都对应一个与该表进行交互的“模型”，模型允许你在表中进行数据查询，以及插入、更新、删除等操作。

在开始之前，确保在 `config/database.php` 文件中配置好了数据库连接。更多关于数据库配置的信息，请查看文档。

2、定义模型

作为开始，让我们创建一个 Eloquent 模型，模型通常位于 `app` 目录下，你也可以将其放在其他可以被 `composer.json` 文件自动加载的地方。所有 Eloquent 模型都继承自 `Illuminate\Database\Eloquent\Model` 类。

创建模型实例最简单的办法就是使用 Artisan 命令 `make:model`：

```
php artisan make:model User
```

如果你想要在生成模型时生成数据库迁移，可以使用`--migration`或`-m`选项：

```
php artisan make:model User --migration
php artisan make:model User -m
```

Eloquent 模型约定

现在，让我们来看一个 `Flight` 模型类例子，我们将用该类获取和存取数据表 `flights` 中的信息：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model{
    //
}
```

表名

注意我们并没有告诉 Eloquent 我们的 `Flight` 模型使用哪张表。默认规则是模型类名的复数作为与其对应的表名，除非在模型类中明确指定了其它名称。所以，在本例中，Eloquent 认为 `Flight` 模型存储记录在 `flights` 表中。你也可以在模型中定义 `table` 属性来指定自定义的表名：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model{
    /**
     * 关联到模型的数据表
     *
     * @var string
     */
    protected $table = 'my_flights';
```

```
}
```

主键

Eloquent 默认每张表的主键名为 `id`，你可以在模型类中定义一个`$primaryKey`属性来覆盖该约定。

时间戳

默认情况下，Eloquent 期望 `created_at` 和 `updated_at` 已经存在于数据表中，如果你不想要这些 Laravel 自动管理的列，在模型类中设置`$timestamps`属性为 `false`：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model{
    /**
     * 表明模型是否应该被打上时间戳
     *
     * @var bool
     */
    public $timestamps = false;
}
```

如果你需要自定义时间戳格式，设置模型中的`$dateFormat`属性。该属性决定日期被如何存储到数据库中，以及模型被序列化为数组或 JSON 时日期的格式：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model{
    /**
     * 模型日期列的存储格式
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

数据库连接

默认情况下，所有的 Eloquent 模型使用应用配置中的默认数据库连接，如果你想要为模型指定不同的连接，可以通过`$connection`属性来设置：

```
<?php


namespace App;


use Illuminate\Database\Eloquent\Model;


class Flight extends Model{
    /**
     * The connection name for the model.
     *
     * @var string
     */
    protected $connection = 'connection-name';
}
```

3、获取多个模型

创建完模型及其关联的数据表后，就要准备从数据库中获取数据。将 Eloquent 模型看作功能强大的查询构建器，你可以使用它来流畅的查询与其关联的数据表。例如：

```
<?php

namespace App\Http\Controllers;

use App\Flight;
use App\Http\Controllers\Controller;

class FlightController extends Controller{
    /**
     * 显示所有有效航班列表
     *
     * @return Response
     */
    public function index()
    {
        $flights = Flight::all();
        return view('flight.index', ['flights' => $flights]);
    }
```

```
}
```

访问列值

如果你有一个 Eloquent 模型实例，可以通过访问其相应的属性来访问模型的列值。例如，让我们循环查询返回的每一个 `Flight` 实例并输出 `name` 的值：

```
foreach ($flights as $flight) {
    echo $flight->name;
}
```

添加额外约束

Eloquent 的 `all` 方法返回模型表的所有结果，由于每一个 Eloquent 模型都是一个查询构建器，你还可以添加约束条件到查询，然后使用 `get` 方法获取对应结果：

```
$flights = App\Flight::where('active', 1)
              ->orderBy('name', 'desc')
              ->take(10)
              ->get();
```

注意：由于 Eloquent 模型本质上就是查询构建器，你可以在 Eloquent 查询中使用查询构建器的所有方法。

集合

对 Eloquent 中获取多个结果的方法（比如 `all` 和 `get`）而言，其返回值是 `Illuminate\Database\Eloquent\Collection` 的一个实例，`Collection` 类提供了多个有用的函数来处理 Eloquent 结果。当然，你可以像操作数组一样简单循环这个集合：

```
foreach ($flights as $flight) {
    echo $flight->name;
}
```

组块结果集

如果你需要处理成千上万个 Eloquent 结果，可以使用 `chunk` 命令。`chunk` 方法会获取一个“组块”的 Eloquent 模型，并将其填充到给定闭包进行处理。使用 `chunk` 方法能够在处理大量数据集合时有效减少内存消耗：

```
Flight::chunk(200, function ($flights) {
    foreach ($flights as $flight) {
        //
    }
});
```

传递给该方法的第一个参数是你想要获取的“组块”数目，闭包作为第二个参数被调用用于处理每个从数据库获取的区块数据。

4、获取单个模型/聚合

当然，除了从给定表中获取所有记录之外，还可以使用 `find` 和 `first` 获取单个记录。这些方法返回单个模型实例而不是返回模型集合：

```
// 通过主键获取模型...
$flight = App\Flight::find(1);
// 获取匹配查询条件的第一个模型...
$flight = App\Flight::where('active', 1)->first();
```

Not Found 异常

有时候你可能想要在模型找不到的时候抛出异常，这在路由或控制器中非常有用，`findOrFail` 和 `firstOrFail` 方法会获取查询到的第一个结果。然而，如果没有任何查询结果，`Illuminate\Database\Eloquent\ModelNotFoundException` 异常将会被抛出：

```
$model = App\Flight::findOrFail(1);
$model = App\Flight::where('legs', '>', 100)->firstOrFail();
```

如果异常没有被捕获，那么 HTTP 404 响应将会被发送给用户，所以在使用这些方法的时候没有必要对返回 404 响应编写明确的检查：

```
Route::get('/api/flights/{id}', function ($id) {
    return App\Flight::findOrFail($id);
});
```

获取聚合

当然，你还可以使用查询构建器聚合方法，例如 `count`、`sum`、`max`，以及其它查询构建器提供的聚合方法。这些方法返回计算后的结果而不是整个模型实例：

```
$count = App\Flight::where('active', 1)->count();
$max = App\Flight::where('active', 1)->max('price');
```

5、插入/更新模型

基本插入

想要在数据库中插入新的记录，只需创建一个新的模型实例，设置模型的属性，然后调用 `save` 方法：

```
<?php

namespace App\Http\Controllers;

use App\Flight;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;
```

```
class FlightController extends Controller{
    /**
     * 创建一个新的航班实例
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Validate the request...

        $flight = new Flight;

        $flight->name = $request->name;

        $flight->save();
    }
}
```

在这个例子中，我们只是简单分配 HTTP 请求中的 `name` 参数值给 `App\Flight` 模型实例的那么属性，当我们调用 `save` 方法时，一条记录将会被插入数据库。`created_at` 和 `updated_at` 时间戳在 `save` 方法被调用时会自动被设置，所以没必要手动设置它们。

基本更新

`save` 方法还可以用于更新数据库中已存在的模型。要更新一个模型，应该先获取它，设置你想要更新的属性，然后调用 `save` 方法。同样，`updated_at` 时间戳会被自动更新，所以没必要手动设置其值：

```
$flight = App\Flight::find(1);
$flight->name = 'New Flight Name';
$flight->save();
```

更新操作还可以同时修改给定查询提供的多个模型实例，在本例中，所有有效且 `destination=San Diego` 的航班都被标记为延迟：

```
App\Flight::where('active', 1)
          ->where('destination', 'San Diego')
          ->update(['delayed' => 1]);
```

`update` 方法要求以数组形式传递键值对参数，代表着数据表中应该被更新的列。

批量赋值

还可以使用 `create` 方法保存一个新的模型。该方法返回被插入的模型实例。但是，在此之前，你需要指定模型的 `fillable` 或 `guarded` 属性，因为所有 Eloquent 模型都通过批量赋值（Mass Assignment）进行保护。

当用户通过 HTTP 请求传递一个不被期望的参数值时就会出现安全隐患，然后该参数以不被期望的方式修改数据库中的列值。例如，恶意用户通过 HTTP 请求发送一个 `is_admin` 参数，然后该参数映射到模型的 `create` 方法，从而允许用户将自己变成管理员。

所以，你应该在模型中定义哪些属性是可以进行赋值的，使用模型上的`$fillable`属性即可实现。例如，我们设置 `Flight` 模型上的 `name` 属性可以被赋值：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model{
    /**
     * 可以被批量赋值的属性.
     *
     * @var array
     */
    protected $fillable = ['name'];
}
```

设置完可以被赋值的属性之后，我们就可以使用 `create` 方法在数据库中插入一条新的记录。`create` 方法返回保存后的模型实例：

```
$flight = App\Flight::create(['name' => 'Flight 10']);
```

`$fillable` 就像是可以被赋值属性的“白名单”，还可以选择使用`$guarded`。`$guarded` 属性包含你不想被赋值的属性数组。所以不被包含在其中的属性都是可以被赋值的，因此，$guarded 方法就像“黑名单”。当然，你只能同时使用其中一个——而不是一起使用：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model{
    /**
     * 不能被批量赋值的属性
     *
     * @var array
     */
    protected $guarded = ['price'];
}
```

在这个例子中，除了`$price`之外的所有属性都是可以被赋值的。

其它创建方法

还有其它两种可以用来创建模型的方法：`firstOrCreate` 和 `firstOrNew`。`firstOrCreate` 方法先尝试通过给定列/值对在数据库中查找记录，如果没有找到的话则通过给定属性创建一个新的记录。

`firstOrNew` 方法和 `firstOrCreate` 方法一样先尝试在数据库中查找匹配的记录，如果没有找到，则返回一个的模型实例。注意通过 `firstOrNew` 方法返回的模型实例并没有持久化到数据库中，你还需要调用 `save` 方法手动持久化：

```
// 通过属性获取航班，如果不存在则创建...

$flight = App\Flight::firstOrCreate(['name' => 'Flight 10']);

// 通过属性获取航班，如果不存在初始化一个新的实例...

$flight = App\Flight::firstOrNew(['name' => 'Flight 10']);
```

6、删除模型

要删除一个模型，调用模型实例上的 `delete` 方法：

```
$flight = App\Flight::find(1);
$flight->delete();
```

通过主键删除模型

在上面的例子中，我们在调用 `delete` 方法之前从数据库中获取该模型，然而，如果你知道模型的主键的话，可以调用 destroy 方法直接删除而不需要获取它：

```
App\Flight::destroy(1);
App\Flight::destroy([1, 2, 3]);
App\Flight::destroy(1, 2, 3);
```

通过查询删除模型

当然，你还可以通过查询删除多个模型，在本例中，我们删除所有被标记为无效的航班：

```
$deletedRows = App\Flight::where('active', 0)->delete();
```

软删除

除了从数据库删除记录外，Eloquent 还可以对模型进行“软删除”。当模型被软删除后，它们并没有真的从数据库删除，而是在模型上设置一个 `deleted_at` 属性并插入数据库，如果模型有一个非空 `deleted_at` 值，那么该模型已经被软删除了。要启用模型的软删除功能，可以使用模型上的 `Illuminate\Database\Eloquent\SoftDeletes`trait 并添加 `deleted_at` 列到`$dates` 属性：

```
<?php
```

```
namespace App;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Flight extends Model{
    use SoftDeletes;

    /**
     * 应该被调整为日期的属性
     *
     * @var array
     */
    protected $dates = ['deleted_at'];
}
```

当然，应该添加 `deleted_at` 列到数据表。Laravel Schema 构建器包含一个帮助函数来创建该列：

```
Schema::table('flights', function ($table) {
    $table->softDeletes();
});
```

现在，当调用模型的 `delete` 方法时，`deleted_at` 列将被设置为当前日期和时间，并且，当查询一个使用软删除的模型时，被软删除的模型将会自动从查询结果中排除。
判断给定模型实例是否被软删除，可以使用 `trashed` 方法：

```
if ($flight->trashed()) {
    //
}
```

查询被软删除的模型

包含软删除模型

正如上面提到的，软删除模型将会自动从查询结果中排除，但是，如果你想要软删除模型出现在查询结果中，可以使用 `withTrashed` 方法：

```
$flights = App\Flight::withTrashed()
                ->where('account_id', 1)
                ->get();
```

`withTrashed` 方法也可以用于关联查询中：

```
$flight->history()->withTrashed()->get();
```

只获取软删除模型

`onlyTrashed` 方法之获取软删除模型：

```
$flights = App\Flight::onlyTrashed()
```

```
                ->where('airline_id', 1)
                ->get();
```

恢复软删除模型

有时候你希望恢复一个被软删除的模型，可以使用 `restore` 方法：

```
$flight->restore();
```

你还可以在查询中使用 `restore` 方法来快速恢复多个模型：

```
App\Flight::withTrashed()
        ->where('airline_id', 1)
        ->restore();
```

和 `withTrashed` 方法一样，`restore` 方法也可以用于关联查询：

```
$flight->history()->restore();
```

永久删除模型

有时候你真的需要从数据库中删除一个模型，可以使用 `forceDelete` 方法：

```
// 强制删除单个模型实例...
$flight->forceDelete();
// 强制删除所有关联模型...
$flight->history()->forceDelete();
```

7、查询作用域

全局作用域

全局作用域允许我们为给定模型的所有查询添加条件约束。Laravel 自带的软删除功能就使用了全局作用域来从数据库中拉出所有没有被删除的模型。编写自定义的全局作用域可以提供一种方便的、简单的方式来确保给定模型的每个查询都有特定的条件约束。

编写全局作用域

自定义全局作用域很简单，首先定义一个实现 `Illuminate\Database\Eloquent\Scope` 接口的类，该接口要求你实现一个方法：apply。需要的话可以在 `apply` 方法中添加 `where` 条件到查询：

```
<?php


namespace App\Scopes;


use Illuminate\Database\Eloquent\Scope;

use Illuminate\Database\Eloquent\Model;
```

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


class AgeScope implements Scope{
    /**
     * Apply the scope to a given Eloquent query builder.
     *
     * @param  \Illuminate\Database\Eloquent\Builder  $builder
     * @param  \Illuminate\Database\Eloquent\Model  $model
     * @return void
     */
    public function apply(Builder $builder, Model $model)
    {
        return $builder->where('age', '>', 200);
    }
}
```

应用全局作用域

要将全局作用域分配给模型，需要重写给定模型的 `boot` 方法并使用 `addGlobalScope` 方法：

```
<?php


namespace App;


use App\Scopes\AgeScope;
use Illuminate\Database\Eloquent\Model;


class User extends Model{
    /**
     * The "booting" method of the model.
     *
     * @return void
```

```
     */
    protected static function boot()
    {
        parent::boot();

        static::addGlobalScope(new AgeScope);
    }
}
```

添加作用域后，如果使用 `User::all()` 查询则会生成如下 SQL 语句：

```
select * from `users` where `age` > 200
```

匿名的全局作用域

Eloquent 还允许我们使用闭包定义全局作用域，这在实现简单作用域的时候特别有用，这样的话，我们就没必要定义一个单独的类了：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Builder;

class User extends Model{
    /**
     * The "booting" method of the model.
     *
     * @return void
     */
    protected static function boot()
    {
        parent::boot();
```

```
        static::addGlobalScope('age', function(Builder $builder) {
            $builder->where('age', '>', 200);
        });
    }
}
```

我们还可以通过以下方式移除全局作用：

```
User::withoutGlobalScope('age')->get();
```

移除全局作用域

如果想要在给定查询中移除指定全局作用域，可以使用 `withoutGlobalScope`：

```
User::withoutGlobalScope(AgeScope::class)->get();
```

如果你想要移除某几个或全部全局作用域，可以使用 `withoutGlobalScopes` 方法：

```
User::withoutGlobalScopes()->get();
User::withoutGlobalScopes([FirstScope::class, SecondScope::class])->get();
```

本地作用域

本地作用域允许我们定义通用的约束集合以便在应用中复用。例如，你可能经常需要获取最受欢迎的用户，要定义这样的一个作用域，只需简单在对应 Eloquent 模型方法前加上一个 `scope` 前缀，作用域总是返回查询构建器：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 只包含活跃用户的查询作用域
     *
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopePopular($query)
    {
        return $query->where('votes', '>', 100);
    }
```

```
    /**
     * 只包含激活用户的查询作用域
     *
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopeActive($query)
    {
        return $query->where('active', 1);
    }
}
```

使用查询作用域

作用域被定义好了之后，就可以在查询模型的时候调用作用域方法，但调用时不需要加上`scope`前缀，你甚至可以在同时调用多个作用域，例如：

```
$users = App\User::popular()->active()->orderBy('created_at')
->get();
```

动态作用域

有时候你可能想要定义一个可以接收参数的作用域，你只需要将额外的参数添加到你的作用域即可。作用域参数应该被定义在`$query`参数之后：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 只包含给用类型用户的查询作用域
     *
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopeOfType($query, $type)
    {
        return $query->where('type', $type);
    }
}
```

现在，你可以在调用作用域时传递参数了：

```
$users = App\User::ofType('admin')->get();
```

8、事件

Eloquent 模型可以触发事件，允许你在模型生命周期中的多个时间点调用如下这些方法：`creating`, `created`, `updating`, `updated`, `saving`, `saved`,`deleting`, `deleted`, `restoring`, `restored`。事件允许你在一个指定模型类每次保存或更新的时候执行代码。

基本使用

一个新模型被首次保存的时候，`creating` 和 `created` 事件会被触发。如果一个模型已经在数据库中存在并调用 `save` 方法，`updating/updated` 事件会被触发，无论是创建还是更新，saving/saved 事件都会被调用。
举个例子，我们在服务提供者中定义一个 Eloquent 事件监听器，在事件监听器中，我们会调用给定模型的 `isValid` 方法，如果模型无效会返回 `false`。如果从 Eloquent 事件监听器中返回 `false` 则取消 `save/update` 操作：

```
<?php

namespace App\Providers;

use App\User;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider{
    /**
     * 启动所有应用服务
     *
     * @return void
     */
    public function boot()
    {
        User::creating(function ($user) {
            if ( ! $user->isValid()) {
                return false;
            }
        });
    }

    /**
     * 注册服务提供者.
     *
     * @return void
     */
    public function register()
```

```
    {
        //
    }
}
```

关联关系

1、简介

数据表经常要与其它表做关联，比如一篇博客文章可能有很多评论，或者一个订单会被关联到下单用户，Eloquent 使得组织和处理这些关联关系变得简单，并且支持多种不同类型的关联关系：

- 一对一
- 一对多
- 多对多
- 远层一对多
- 多态关联
- 多对多的多态关联

2、定义关联关系

Eloquent 关联关系以 Eloquent 模型类方法的形式被定义。和 Eloquent 模型本身一样，关联关系也是强大的查询构建器，定义关联关系为函数能够提供功能强大的方法链和查询能力。例如：

```
$user->posts()->where('active', 1)->get();
```

但是，在深入使用关联关系之前，让我们先学习如何定义每种关联类型：

一对一

一对一关联是一个非常简单的关联关系，例如，一个 `User` 模型有一个与之对应的 `Phone` 模型。要定义这种模型，我们需要将 `phone` 方法置于 `User` 模型中，`phone` 方法应该返回 Eloquent 模型基类上 `hasOne` 方法的结果：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
```

```
     * 获取关联到用户的手机
     */
    public function phone()
    {
        return $this->hasOne('App\Phone');
    }
}
```

传递给 `hasOne` 方法的第一个参数是关联模型的名称，关联关系被定义后，我们可以使用 Eloquent 的动态属性获取关联记录。动态属性允许我们访问关联函数就像它们是定义在模型上的属性一样：

```
$phone = User::find(1)->phone;
```

Eloquent 默认关联关系的外键基于模型名称，在本例中，`Phone` 模型默认有一个 `user_id` 外键，如果你希望重写这种约定，可以传递第二个参数到 `hasOne` 方法：

```
return $this->hasOne('App\Phone', 'foreign_key');
```

此外，Eloquent 假设外键应该在父级上有一个与之匹配的 `id`，换句话说，Eloquent 将会通过 `user` 表的 `id` 值去 `phone` 表中查询 `user_id` 与之匹配的 `Phone` 记录。如果你想要关联关系使用其他值而不是 `id`，可以传递第三个参数到 `hasOne` 来指定自定义的主键：

```
return $this->hasOne('App\Phone', 'foreign_key', 'local_key');
```

定义相对的关联

我们可以从 `User` 中访问 `Phone` 模型，相应的，我们也可以在 `Phone` 模型中定义关联关系从而让我们可以拥有该 `phone` 的 `User`。我们可以使用 `belongsTo` 方法定义与 `hasOne` 关联关系相对的关联：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Phone extends Model{
    /**
     * 获取手机对应的用户
     */
    public function user()
    {
        return $this->belongsTo('App\User');
    }
}
```

在上面的例子中，Eloquent 将会尝试通过 `Phone` 模型的 `user_id` 去 `User` 模型查找与之匹配的记录。Eloquent 通过关联关系方法名并在方法名后加 `_id` 后缀来生成默认的外键名。

然而，如果 `Phone` 模型上的外键不是 `user_id`，也可以将自定义的键名作为第二个参数传递到 `belongsTo` 方法：

```
/**
 * 获取手机对应的用户
 */
public function user(){
    return $this->belongsTo('App\User', 'foreign_key');
}
```

如果父模型不使用 `id` 作为主键，或者你希望使用别的列来连接子模型，可以将父表自定义键作为第三个参数传递给 `belongsTo` 方法：

```
/**
 * 获取手机对应的用户
 */
public function user(){
    return $this->belongsTo('App\User', 'foreign_key', 'other_key');
}
```

一对多

“一对多”是用于定义单个模型拥有多个其它模型的关联关系。例如，一篇博客文章拥有无数评论，和其他关联关系一样，一对多关联通过在 Eloquent 模型中定义方法来定义：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model{
    /**
     * 获取博客文章的评论
     */
    public function comments()
    {
        return $this->hasMany('App\Comment');
    }
}
```

记住，Eloquent 会自动判断 `Comment` 模型的外键，为方便起见，Eloquent 将拥有者模型名称加上 `id` 后缀作为外键。因此，在本例中，Eloquent 假设 `Comment` 模型上的外键是 `post_id`。

关联关系被定义后，我们就可以通过访问 `comments` 属性来访问评论集合。记住，由于 Eloquent 提供“动态属性”，我们可以像访问模型的属性一样访问关联方法：

```
$comments = App\Post::find(1)->comments;

foreach ($comments as $comment) {
    //
}
```

当然，由于所有关联同时也是查询构建器，我们可以添加更多的条件约束到通过调用 `comments` 方法获取到的评论上：

```
$comments = App\Post::find(1)->comments()->where('title', 'foo
')->first();
```

和 `hasOne` 方法一样，你还可以通过传递额外参数到 `hasMany` 方法来重新设置外键和本地主键：

```
return $this->hasMany('App\Comment', 'foreign_key');
return $this->hasMany('App\Comment', 'foreign_key', 'local_key
');
```

定义相对的关联

现在我们可以访问文章的所有评论了，接下来让我们定义一个关联关系允许通过评论访问所属文章。要定义与 `hasMany` 相对的关联关系，需要在子模型中定义一个关联方法去调用 `belongsTo` 方法：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model{
    /**
     * 获取评论对应的博客文章
     */
    public function post()
    {
        return $this->belongsTo('App\Post');
    }
}
```

关联关系定义好之后，我们可以通过访问动态属性 `post` 来获取一条 `Comment` 对应的 `Post`：

```
$comment = App\Comment::find(1);
echo $comment->post->title;
```

在上面这个例子中，Eloquent 尝试匹配 `Comment` 模型的 `post_id` 与 `Post` 模型的 `id`，Eloquent 通过关联方法名加上 `_id` 后缀生成默认外键，当然，你也可以通过传递自定义外

键名作为第二个参数传递到 `belongsTo` 方法，如果你的外键不是 `post_id`，或者你想自定义的话：

```
/**
 * 获取评论对应的博客文章
 */
public function post(){
    return $this->belongsTo('App\Post', 'foreign_key');
}
```

如果你的父模型不使用 `id` 作为主键，或者你希望通过其他列来连接子模型，可以将自定义键名作为第三个参数传递给 `belongsTo` 方法：

```
/**
 * 获取评论对应的博客文章
 */
public function post(){
    return $this->belongsTo('App\Post', 'foreign_key', 'other_key');
}
```

多对多

多对多关系比 `hasOne` 和 `hasMany` 关联关系要稍微复杂一些。这种关联关系的一个例子就是一个用户有多个角色，同时一个角色被多个用户共用。例如，很多用户可能都有一个“Admin”角色。要定义这样的关联关系，需要三个数据表：`users`、`roles` 和 `role_user`，`role_user` 表按照关联模型名的字母顺序命名，并且包含 `user_id` 和 `role_id` 两个列。多对多关联通过编写一个调用 Eloquent 基类上的 `belongsToMany` 方法的函数来定义：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 用户角色
     */
    public function roles()
    {
        return $this->belongsToMany('App\Role');
    }
}
```

关联关系被定义之后，可以使用动态属性 `roles` 来访问用户的角色：

```
$user = App\User::find(1);
```

```
foreach ($user->roles as $role) {
    //
}
```

当然，和所有其它关联关系类型一样，你可以调用 `roles` 方法来添加条件约束到关联查询上：

```
$roles = App\User::find(1)->roles()->orderBy('name')->get();
```

正如前面所提到的，为了决定关联关系连接表的表名，Eloquent 以字母顺序连接两个关联模型的名字。然而，你可以重写这种约定——通过传递第二个参数到 `belongsToMany` 方法：

```
return $this->belongsToMany('App\Role', 'user_roles');
```

除了自定义连接表的表名，你还可以通过传递额外参数到 `belongsToMany` 方法来自定义该表中字段的列名。第三个参数是你定义的关系模型的外键名称，第四个参数你要连接到的模型的外键名称：

```
return $this->belongsToMany('App\Role', 'user_roles', 'user_id', 'role_id');
```

定义相对的关联关系

要定义与多对多关联相对的关联关系，只需在关联模型中在调用一下 `belongsToMany` 方法即可。让我们在 `Role` 模型中定义 `users` 方法：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Role extends Model{
    /**
     * 角色用户
     */
    public function users()
    {
        return $this->belongsToMany('App\User');
    }
}
```

正如你所看到的，定义的关联关系和与其对应的 `User` 中定义的一模一样，只是前者引用 `App\Role`，后者引用 `App\User`，由于我们再次使用了 `belongsToMany` 方法，所有的常用表和键自定义选项在定义与多对多相对的关联关系时都是可用的。

获取中间表的列

正如你已经学习到的，处理多对多关联要求一个中间表。Eloquent 提供了一些有用的方法来与其进行交互，例如，我们假设 `User` 对象有很多与之关联的 `Role` 对象，访问这些关联关系之后，我们可以使用模型上的 `pivot` 属性访问中间表：

```
$user = App\User::find(1);

foreach ($user->roles as $role) {
    echo $role->pivot->created_at;
}
```

注意我们获取到的每一个 `Role` 模型都被自动赋上了 `pivot` 属性。该属性包含一个代表中间表的模型，并且可以像其它 Eloquent 模型一样使用。

默认情况下，只有模型键才能用在 `privot` 对象上，如果你的 `privot` 表包含额外的属性，必须在定义关联关系时进行指定：

```
return $this->belongsToMany('App\Role')->withPivot('column1', '
column2');
```

如果你想要你的 `privot` 表自动包含 `created_at` 和 `updated_at` 时间戳，在关联关系定义时使用 `withTimestamps` 方法：

```
return $this->belongsToMany('App\Role')->withTimestamps();
```

远层的一对多

“远层一对多”关联为通过中间关联访问远层的关联关系提供了一个便利之道。例如，`Country` 模型通过中间的 `User` 模型可能拥有多个 `Post` 模型。在这个例子中，你可以轻易的聚合给定国家的所有文章，让我们看看定义这个关联关系需要哪些表：

```
countries
    id - integer
    name - string

users
    id - integer
    country_id - integer
    name - string

posts
    id - integer
    user_id - integer
    title - string
```

尽管 `posts` 表不包含 `country_id` 列，`hasManyThrough` 关联提供了通过`$country->posts` 来访问一个国家的所有文章。要执行该查询，Eloquent 在中间表`$users` 上检查 `country_id`，查找到相匹配的用户 ID 后，通过用户 ID 来查询 `posts` 表。

既然我们已经查看了该关联关系的数据表结构，接下来让我们在 `Country` 模型上进行定义：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Country extends Model{
    /**
     * 获取指定国家的所有文章
     */
    public function posts()
    {
        return $this->hasManyThrough('App\Post', 'App\User');
    }
}
```

第一个传递到 `hasManyThrough` 方法的参数是最终我们希望访问的模型的名称，第二个参数是中间模型名称。

当执行这种关联查询时通常 Eloquent 外键规则会被使用，如果你想要自定义该关联关系的外键，可以将它们作为第三个、第四个参数传递给 `hasManyThrough` 方法。第三个参数是中间模型的外键名，第四个参数是最终模型的外键名。

```
class Country extends Model{
    public function posts()
    {
        return $this->hasManyThrough('App\Post', 'App\User', 'country_id', 'user_id');
    }
}
```

多态关联

表结构

多态关联允许一个模型在单个关联下属于多个不同模型。例如，假如应用用户可以对文章点赞也可以对评论点赞，使用多态关联，你可以在这两种场景下使用单个 `likes` 表，首先，让我们看看构建这种关联关系需要的表结构：

```
posts

    id - integer

    title - string

    body - text
```

```
comments
    id - integer
    post_id - integer
    body - text

likes
    id - integer
    likeable_id - integer
    likeable_type - string
```

两个重要的需要注意的列是 `likes` 表上的 `likeable_id` 和 `likeable_type`。`likeable_id` 列对应 `Post` 或 `Comment` 的 ID 值，而 `likeable_type` 列对应所属模型的类名。当访问 `likeable` 关联时，ORM 根据 `likeable_type` 列来判断所属模型的类型并返回相应模型实例。

模型结构

接下来，让我们看看构建这种关联关系需要在模型中定义什么：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Like extends Model{
    /**
     * 获取所属的 likeable 模型
     */
    public function likeable()
    {
        return $this->morphTo();
    }
}
```

```
class Post extends Model{
    /**
     * 获取该文章的所有点赞
     */
    public function likes()
    {
        return $this->morphMany('App\Like', 'likeable');
    }
}

class Comment extends Model{
    /**
     * 获取该评论的所有点赞
     */
    public function likes()
    {
        return $this->morphMany('App\Like', 'likeable');
    }
}
```

获取多态关联

数据表和模型定义好以后，可以通过模型访问关联关系。例如，要访问一篇文章的所有点赞，可以通过使用动态属性 likes ：

```
$post = App\Post::find(1);

foreach ($post->likes as $like) {
    //
}
```

你还可以通过调用 `morphTo` 方法从多态模型中获取多态关联的所属对象。在本例中，就是 `Like` 模型中的 `likeable` 方法。因此，我们可以用动态属性的方式访问该方法：

```
$like = App\Like::find(1);

$likeable = $like->likeable;
```

`Like` 模型的 `likeable` 关联返回 `Post` 或 `Comment` 实例，这取决于哪个类型的模型拥有该点赞。

多对多的多态关联

表结构

除了传统的多态关联，还可以定义“多对多”的多态关联，例如，一个博客的 `Post` 和 `Video` 模型可能共享一个 `Tag` 模型的多态关联。使用对多对的多态关联允许你在博客文章和视频之间有唯一的标签列表。首先，让我们看看表结构：

```
posts
    id - integer
    name - string

videos
    id - integer
    name - string

tags
    id - integer
    name - string

taggables
    tag_id - integer
    taggable_id - integer
    taggable_type - string
```

模型结构

接下来，我们准备在模型中定义该关联关系。`Post` 和 `Video` 模型都有一个 `tags` 方法调用 Eloquent 基类的 `morphToMany` 方法：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model{
    /**
     * 获取指定文章所有标签
     */
    public function tags()
```

```
    {
        return $this->morphToMany('App\Tag', 'taggable');
    }
}
```

定义相对的关联关系

接下来，在 `Tag` 模型中，应该为每一个关联模型定义一个方法，例如，我们定义一个 `posts` 方法和 `videos` 方法：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Tag extends Model{
    /**
     * 获取所有分配该标签的文章
     */
    public function posts()
    {
        return $this->morphedByMany('App\Post', 'taggable');
    }

    /**
     * 获取分配该标签的所有视频
     */
    public function videos()
    {
        return $this->morphedByMany('App\Video', 'taggable');
    }
}
```

获取关联关系

定义好数据库和模型后可以通过模型访问关联关系。例如，要访问一篇文章的所有标签，可以使用动态属性 `tags`：

```
$post = App\Post::find(1);

foreach ($post->tags as $tag) {
    //
}
```

还可以通过访问调用 `morphedByMany` 的方法名从多态模型中获取多态关联的所属对象。在本例中，就是 Tag 模型中的 `posts` 或者 `videos` 方法：

```
$tag = App\Tag::find(1);

foreach ($tag->videos as $video) {
    //
}
```

3、关联查询

由于 Eloquent 所有关联关系都是通过函数定义，你可以调用这些方法来获取关联关系的实例而不需要再去手动执行关联查询。此外，所有 Eloquent 关联关系类型同时也是查询构建器，允许你在最终在数据库执行 SQL 之前继续添加条件约束到关联查询上。

例如，假定在一个博客系统中一个 User 模型有很多相关的 Post 模型：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 获取指定用户的所有文章
     */
    public function posts()
    {
        return $this->hasMany('App\Post');
    }
}
```

你可以像这样查询 `posts` 关联并添加额外的条件约束到该关联关系上：

```
$user = App\User::find(1);
$user->posts()->where('active', 1)->get();
```

你可以在关联关系上使用任何查询构建器！

关联关系方法 VS 动态属性

如果你不需要添加额外的条件约束到 Eloquent 关联查询，你可以简单通过动态属性来访问关联对象，例如，还是拿 `User` 和 `Post` 模型作为例子，你可以像这样访问所有用户的文章：

```
$user = App\User::find(1);

foreach ($user->posts as $post) {
```

```
    //
}
```

动态属性就是“懒惰式加载”，意味着当你真正访问它们的时候才会加载关联数据。正因为如此，开发者经常使用渴求式加载来预加载他们知道在加载模型时要被访问的关联关系。渴求式加载有效减少了必须要被执以加载模型关联的 SQL 查询。

查询已存在的关联关系

访问一个模型的记录的时候，你可能希望基于关联关系是否存在来限制查询结果的数目。例如，假设你想要获取所有至少有一个评论的博客文章，要实现这个，可以传递关联关系的名称到 `has` 方法：

```
// 获取所有至少有一条评论的文章...
$posts = App\Post::has('comments')->get();
```

你还可以指定操作符和大小来自定义查询：

```
// 获取所有至少有三条评论的文章...
$posts = Post::has('comments', '>=', 3)->get();
```

还可以使用”.“来构造嵌套 has 语句，例如，你要获取所有至少有一条评论及投票的所有文章：

```
// 获取所有至少有一条评论获得投票的文章...
$posts = Post::has('comments.votes')->get();
```

如果你需要更强大的功能，可以使用 `whereHas` 和 `orWhereHas` 方法将 `where` 条件放到 `has` 查询上，这些方法允许你添加自定义条件约束到关联关系条件约束，例如检查一条评论的内容：

```
// 获取所有至少有一条评论包含 foo 字样的文章
$posts = Post::whereHas('comments', function ($query) {
    $query->where('content', 'like', 'foo%');
})->get();
```

渴求式加载

当以属性方式访问数据库关联关系的时候，关联关系数据时“懒惰式加载”的，这意味着关联关系数据直到第一次访问的时候才被加载。然而，Eloquent 可以在查询父级模型的同时”渴求式加载“关联关系。渴求式加载缓解了 `N+1` 查询问题，要阐明 `N+1` 查询问题，考虑下关联到 `Author` 的 `Book` 模型：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
```

```
class Book extends Model{
    /**
     * 获取写这本书的作者
     */
    public function author()
    {
        return $this->belongsTo('App\Author');
    }
}
```

现在，让我们获取所有书及其作者：

```
$books = App\Book::all();

foreach ($books as $book) {
    echo $book->author->name;
}
```

该循环先执行 1 次查询获取表中的所有书，然后另一个查询获取每一本书的作者，因此，如果有 25 本书，要执行 26 次查询：1 次是获取书本身，剩下的 25 次查询是为每一本书获取其作者。

谢天谢地，我们可以使用渴求式加载来减少该操作到 2 次查询。当查询的时候，可以使用 `with` 方法指定应该被渴求式加载的关联关系：

```
$books = App\Book::with('author')->get();

foreach ($books as $book) {
    echo $book->author->name;
}
```

在该操作中，只执行两次查询即可：

```
select * from books
select * from authors where id in (1, 2, 3, 4, 5, ...)
```

渴求式加载多个关联关系

有时候你需要在单个操作中渴求式加载几个不同的关联关系。要实现这个，只需要添加额外的参数到 `with` 方法即可：

```
$books = App\Book::with('author', 'publisher')->get();
```

嵌套的渴求式加载

要渴求式加载嵌套的关联关系，可以使用”.“语法。例如，让我们在一个 Eloquent 语句中渴求式加载所有书的作者及所有作者的个人联系方式：

```
$books = App\Book::with('author.contacts')->get();
```

带条件约束的渴求式加载

有时候我们希望渴求式加载一个关联关系，但还想为渴求式加载指定更多的查询条件：

```
$users = App\User::with(['posts' => function ($query) {
    $query->where('title', 'like', '%first%');
}])->get();
```

在这个例子中，Eloquent 只渴求式加载 title 包含 first 的文章。当然，你可以调用其它查询构建器来自定义渴求式加载操作：

```
$users = App\User::with(['posts' => function ($query) {
    $query->orderBy('created_at', 'desc');
}])->get();
```

懒惰渴求式加载

有时候你需要在父模型已经被获取后渴求式加载一个关联关系。例如，这在你需要动态决定是否加载关联模型时可能很有用：

```
$books = App\Book::all();

if ($someCondition) {
    $books->load('author', 'publisher');
}
```

如果你需要设置更多的查询条件到渴求式加载查询上，可以传递一个闭包到 `load` 方法：

```
$books->load(['author' => function ($query) {
    $query->orderBy('published_date', 'asc');
}]);
```

4、插入关联模型

基本使用

save 方法

Eloquent 提供了便利的方法来添加新模型到关联关系。例如，也许你需要插入新的 `Comment` 到 `Post` 模型，你可以从关联关系的 `save` 方法直接插入 `Comment` 而不是手动设置 `Comment` 的 `post_id` 属性：

```
$comment = new App\Comment(['message' => 'A new comment.']);
$post = App\Post::find(1);
$post->comments()->save($comment);
```

注意我们没有用动态属性方式访问 `comments`，而是调用 `comments` 方法获取关联关系实例。`save` 方法会自动添加 `post_id` 值到新的 `Comment` 模型。

如果你需要保存多个关联模型，可以使用 `saveMany` 方法：

```
$post = App\Post::find(1);

$post->comments()->saveMany([
    new App\Comment(['message' => 'A new comment.']),
    new App\Comment(['message' => 'Another comment.']),
]);
```

save & 多对多关联

当处理多对多关联的时候，`save` 方法以数组形式接收额外的中间表属性作为第二个参数：

```
App\User::find(1)->roles()->save($role, ['expires' => $expire
s]);
```

create 方法

除了 `save` 和 `saveMany` 方法外，还可以使用 `create` 方法，该方法接收属性数组、创建模型、然后插入数据库。`save` 和 `create` 的不同之处在于 `save` 接收整个 Eloquent 模型实例而 `create` 接收原生 PHP 数组：

```
$post = App\Post::find(1);

$comment = $post->comments()->create([
    'message' => 'A new comment.',
]);
```

使用 `create` 方法之前确保先浏览属性批量赋值文档。

更新”属于“关联

更新 `belongsTo` 关联的时候，可以使用 `associate` 方法，该方法会在子模型设置外键：

```
$account = App\Account::find(10);
$user->account()->associate($account);
$user->save();
```

移除 `belongsTo` 关联的时候，可以使用 `dissociate` 方法。该方法在子模型上取消外键和关联：

```
$user->account()->dissociate();
$user->save();
```

多对多关联

附加/分离

处理多对多关联的时候，Eloquent 提供了一些额外的帮助函数来使得处理关联模型变得更加方便。例如，让我们假定一个用户可能有多个角色同时一个角色属于多个用户，要通过在连接模型的中间表中插入记录附加角色到用户上，可以使用 `attach` 方法：

```
$user = App\User::find(1);
```

```
$user->roles()->attach($roleId);
```

附加关联关系到模型，还可以以数组形式传递额外被插入数据到中间表：

```
$user->roles()->attach($roleId, ['expires' => $expires]);
```

当然，有时候有必要从用户中移除角色，要移除一个多对多关联记录，使用 `detach` 方法。`detach` 方法将会从中间表中移除相应的记录；然而，两个模型在数据库中都保持不变：

```
// 从指定用户中移除角色...
$user->roles()->detach($roleId);
// 从指定用户移除所有角色...
$user->roles()->detach();
```

为了方便，`attach` 和 `detach` 还接收数组形式的 ID 作为输入：

```
$user = App\User::find(1);
$user->roles()->detach([1, 2, 3]);
$user->roles()->attach([1 => ['expires' => $expires], 2, 3]);
```

同步

你还可以使用 `sync` 方法构建多对多关联。`sync` 方法接收数组形式的 ID 并将其放置到中间表。任何不在该数组中的 ID 对应记录将会从中间表中移除。因此，该操作完成后，只有在数组中的 ID 对应记录还存在于中间表：

```
$user->roles()->sync([1, 2, 3]);
```

你还可以和 ID 一起传递额外的中间表值：

```
$user->roles()->sync([1 => ['expires' => true], 2, 3]);
```

触发父级时间戳

当一个模型属于另外一个时，例如 `Comment` 属于 `Post`，子模型更新时父模型的时间戳也被更新将很有用，例如，当 `Comment` 模型被更新时，你可能想要”触发“创建其所属模型 `Post` 的 `updated_at` 时间戳。Eloquent 使得这项操作变得简单，只需要添加包含关联关系名称的 `touches` 属性到子模型中即可：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model{
    /**
     * 要触发的所有关联关系
     *
     * @var array
```

```
     */
    protected $touches = ['post'];

    /**
     * 评论所属文章
     */
    public function post()
    {
        return $this->belongsTo('App\Post');
    }
}
```

现在，当你更新 `Comment` 时，所属模型 `Post` 将也会更新其 `updated_at` 值：

```
$comment = App\Comment::find(1);
$comment->text = 'Edit to this comment!';
$comment->save();
```

集合

1、简介

Eloquent 返回的所有的包含多条记录的结果集都是 `Illuminate\Database\Eloquent\Collection` 对象的实例，包括通过 `get` 方法或者通过访问关联关系获取的结果。Eloquent 集合对象继承自 Laravel 的集合基类，因此很自然的继承了很多处理 Eloquent 模型底层数组的方法。
当然，所有集合也是迭代器，允许你像数组一样对其进行循环：

```
$users = App\User::where('active', 1)->get();


foreach ($users as $user) {

    echo $user->name;

}
```

然而，集合使用直观的接口提供了各种映射/简化操作，因此比数组更加强大。例如，我们可以通过以下方式移除所有无效的模型并聚合还存在的用户的名字：

```
$users = App\User::where('active', 1)->get();

$names = $users->reject(function ($user) {
    return $user->active === false;
```

```
})->map(function ($user) {
    return $user->name;
});
```

注意：尽管大多数 Eloquent 集合返回的是一个新的 Eloquent 集合实例，但是 `pluck`、`keys`、`zip`、`collapse`、`flatten` 和 `flip` 方法返回的是集合基类实例。

2、可用方法

所有的 Eloquent 集合继承自 Laravel 集合对象基类，因此，它们继承所有集合基类提供的强大方法：

all
chunk
collapse
contains
count
diff
each
filter
first
flatten
flip
forget
forPage
get
groupBy
has
implode
intersect
isEmpty
keyBy
keys
last
map
merge
pluck
pop
prepend
pull
push
put
random
reduce
reject
reverse

search
shift
shuffle
slice
sort
sortBy
sortByDesc
splice
sum
take
toArray
toJson
transform
unique
values
where
whereLoose
zip

3、自定义集合

如果你需要在自己扩展的方法中使用自定义的集合对象，可以重写模型上的 `newCollection` 方法：

```
<?php

namespace App;

use App\CustomCollection;
use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 创建一个新的 Eloquent 集合实例
     *
     * @param  array  $models
     * @return \Illuminate\Database\Eloquent\Collection
     */
    public function newCollection(array $models = [])
    {
        return new CustomCollection($models);
    }
}
```

定义好 `newCollection` 方法后，无论何时 Eloquent 返回该模型的 `Collection` 实例你都会获取到自定义的集合。如果你想要在应用中的每一个模型中使用自定义集合，需要在模型基类中重写 `newCollection` 方法。

访问器&修改器

1、简介

访问器和修改器允许你在获取模型属性或设置其值时格式化 Eloquent 属性。例如，你可能想要使用 Laravel 加密器对存储在数据库中的数据进行加密，并且在 Eloquent 模型中访问时自动进行解密。
除了自定义访问器和修改器，Eloquent 还可以自动转换日期字段为 Carbon 实例甚至将文本转换为 JSON。

2、访问器 & 修改器

定义访问器

要定义一个访问器，需要在模型中创建一个 `getFooAttribute` 方法，其中 `Foo` 是你想要访问的字段名（使用驼峰式命名规则）。在本例中，我们将会为 `first_name` 属性定义一个访问器，该访问器在获取 `first_name` 的值时被 Eloquent 自动调用：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 获取用户的名字
     *
     * @param  string  $value
     * @return string
     */
    public function getFirstNameAttribute($value)
    {
        return ucfirst($value);
    }
}
```

正如你所看到的，该字段的原生值被传递给访问器，然后返回处理过的值。要访问该值只需要简单访问 `first_name` 即可：

```
$user = App\User::find(1);
$firstName = $user->first_name;
```

定义修改器

要定义一个修改器，需要在模型中定义 `setFooAttribute` 方法，其中 `Foo` 是你想要访问的字段（使用驼峰式命名规则）。接下来让我们为 `first_name` 属性定义一个修改器，当我们为模型上的 `first_name` 赋值时该修改器会被自动调用：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 设置用户的名字
     *
     * @param  string  $value
     * @return string
     */
    public function setFirstNameAttribute($value)
    {
        $this->attributes['first_name'] = strtolower($value);
    }
}
```

该修改器获取要被设置的属性值，允许你操纵该值并设置 Eloquent 模型内部属性值为操作后的值。例如，如果你尝试设置 `Sally` 的 `first_name` 属性：

```
$user = App\User::find(1);
$user->first_name = 'Sally';
```

在本例中，`setFirstNameAttribute` 方法会被调用，传入参数为 `Sally`，修改器会对其调用 `strtolower` 函数并将处理后的值设置为内部属性的值。

3、日期修改器

默认情况下，Eloquent 将会转化 `created_at` 和 `updated_at` 列的值为 Carbon 实例，该类继承自 PHP 原生的 `Datetime` 类，并提供了各种有用的方法。
你可以自定义哪些字段被自动调整修改，甚至可以通过重写模型中的`$dates` 属性完全禁止调整：

```
<?php
```

```
namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 应该被调整为日期的属性
     *
     * @var array
     */
    protected $dates = ['created_at', 'updated_at', 'disabled_a
t'];
}
```

如果字段是日期格式时，你可以将其值设置为 UNIX 时间戳，日期字符串（`Y-m-d`），日期-时间字符串，`Datetime/Carbon` 实例，日期的值将会自动以正确格式存储到数据库中：

```
$user = App\User::find(1);
$user->disabled_at = Carbon::now();
$user->save();
```

正如上面提到的，当获取被罗列在`$dates` 数组中的属性时，它们会被自动转化为 `Carbon` 实例，允许你在属性上使用任何 `Carbon` 的方法：

```
$user = App\User::find(1);
return $user->disabled_at->getTimestamp();
```

默认情况下，时间戳的格式是“Y-m-d H:i:s”，如果你需要自定义时间戳格式，在模型中设置`$dateFormat` 属性，该属性决定日期属性存储在数据库以及序列化为数组或 JSON 时的格式：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model{
    /**
     * 模型日期的存储格式
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

4、属性转换

模型中的`$casts`属性为属性字段转换到通用数据类型提供了便利方法 。`$casts`属性是数组格式，其键是要被转换的属性名称，其值时你想要转换的类型。目前支持的转换类型包括：`integer`, `real`, `float`, `double`, `string`, `boolean`, `object`， `array`， `collection`， `date`和`datetime`。

例如，让我们转换`is_admin`属性，将其由`integer`类型（0 或 1）转换为`boolean`类型：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 应该被转化为原生类型的属性
     *
     * @var array
     */
    protected $casts = [
        'is_admin' => 'boolean',
    ];
}
```

现在，`is_admin`属性在被访问时总是被转换为`boolean`，即使底层存储在数据库中的值是`integer`：

```
$user = App\User::find(1);

if ($user->is_admin) {
    //
}
```

数组转换

`array`类型转换在处理被存储为序列化 JSON 格式的字段时特别有用，例如，如果数据库有一个 TEXT 字段类型包含了序列化 JSON，添加`array`类型转换到该属性将会在 Eloquent 模型中访问其值时自动将其反序列化为 PHP 数组：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
```

```
class User extends Model{
    /**
     * 应该被转化为原生类型的属性
     *
     * @var array
     */
    protected $casts = [
        'options' => 'array',
    ];
}
```

类型转换被定义后，访问 `options` 属性将会自动从 JSON 反序列化为 PHP 数组，反之，当你设置 `options` 属性的值时，给定数组将会自动转化为 JSON 以供存储：

```
$user = App\User::find(1);
$options = $user->options;
$options['key'] = 'value';
$user->options = $options;
$user->save();
```

序列化

1、简介

当构建 JSON API 时，经常需要转化模型和关联关系为数组或 JSON。Eloquent 包含便捷方法实现这些转换，以及控制哪些属性被包含到序列化中。

2、基本使用

转化模型为数组

要转化模型及其加载的关联关系为数组，可以使用 `toArray` 方法。这个方法是递归的，所以所有属性及其关联对象属性（包括关联的关联）都会被转化为数组：

```
$user = App\User::with('roles')->first();
return $user->toArray();
```

还可以转化集合为数组：

```
$users = App\User::all();
return $users->toArray();
```

转化模型为 JSON

要转化模型为 JSON，可以使用 `toJson` 方法，和 `toArray` 一样，`toJson` 方法也是递归的，所有属性及其关联属性都会被转化为 JSON：

```
$user = App\User::find(1);
return $user->toJson();
```

你还可以转化模型或集合为字符串，这将会自动调用 `toJson` 方法：

```
$user = App\User::find(1);
return (string) $user;
```

由于模型和集合在转化为字符串的时候会被转化为 JSON，你可以从应用的路由或控制器中直接返回 Eloquent 对象：

```
Route::get('users',function(){
    return App\User::all();
});
```

3、在 JSON 中隐藏属性

有时候你希望在模型数组或 JSON 显示中隐藏某些属性，比如密码，要实现这个，在定义模型的时候设置 `$hidden` 属性：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 在数组中隐藏的属性
     *
     * @var array
     */
    protected $hidden = ['password'];
}
```

注意：如果要隐藏关联关系，使用关联关系的方法名，而不是动态属性名。
此外，可以使用 `visible` 属性来定义模型数组和 JSON 显示的属性白名单：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
```

```
class User extends Model{
    /**
     * 在数组中显示的属性
     *
     * @var array
     */
    protected $visible = ['first_name', 'last_name'];
}
```

临时暴露隐藏属性

如果你想要在特定模型中临时显示隐藏的属性，可以使用 `makeVisible` 方法，该方法以方法链的方式返回模型实例：

```
return $user->makeVisible('attribute')->toArray();
```

4、追加值到 JSON

有时候，需要添加数据库中没有的字段到数组中，要实现这个功能，首先要为这个值定义一个访问器：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 为用户获取管理员标识
     *
     * @return bool
     */
    public function getIsAdminAttribute()
    {
        return $this->attributes['admin'] == 'yes';
    }
}
```

定义好访问器后，添加字段名到该模型的 `appends` 属性：

```
<?php

namespace App;
```

```
use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 追加到模型数组表单的访问器
     *
     * @var array
     */
    protected $appends = ['is_admin'];
}
```

字段被添加到 `appends` 列表之后，将会被包含到模型数组和 JSON 中，`appends` 数组中的字段还会遵循模型中配置的 `visible` 和 `hidden` 设置。

七、服务

用户认证

1、简介

Laravel 中实现用户认证非常简单。实际上，几乎所有东西都已经为你配置好了。配置文件位于 `config/auth.php`，其中包含了用于调整认证服务行为的、文档友好的选项配置。
在底层代码中，Laravel 的认证组件由“guards”和“providers”组成，Guard 定义了用户在每个请求中如何实现认证，例如，Laravel 通过 `session` guard 来维护 Session 存储的状态、Cookie 以及 `token` guard，`token` guard 是认证用户发送请求时带的“API token”。
Provider 定义了如何从持久化存储中获取用户信息，Laravel 底层支持通过 Eloquent 和数据库查询构建器两种方式来获取用户，如果需要的话，你还可以定义额外的 Provider。
如果看到这些名词觉得云里雾里，大可不必太过担心，因为对绝大多数应用而言，只需使用默认认证配置即可，不需要做什么改动。

数据库考量

默认情况下，Laravel 在 `app` 目录下包含了一个 Eloquent 模型 `App\User`，这个模型可以和默认的 Eloquent 认证驱动一起使用。如果你的应用不使用 Eloquent，你可以使用 `database` 认证驱动，该驱动使用了 Laravel 查询构建器。
为 `App\User` 模型构建数据库表结构的时候，确保 `password` 字段长度至少有 60 位。
还有，你应该验证 `users` 表包含了可以为空的、字符串类型的 `remember_token` 字段长度为 100，该字段用于存储被应用维护的”记住我（remember me）“的 Session 令牌，这可以通过在迁移中使用 `$table->rememberToken();` 来实现。

2、快速入门

Laravel 开箱提供了两个认证控制器，位于 `App\Http\Controllers\Auth` 命名空间下，`AuthController` 处理新用户注册和登录，`PasswordController` 用于帮助用户找回密码。每个控制器都使用 trait 来引入它们需要的方法。对很多应用而言，你根本不需要修改这两个控制器。

路由

Laravel 通过运行如下命令可快速生成认证所需要的路由和视图：

```
php artisan make:auth
```

运行该命令会生成注册和登录视图，以及所有的认证路由，同时生成 `HomeController`，因为登录成功后会跳转到该控制器下的动作。当然，你也可以不用这个命令根据应用需求完全自定义或者移除这个控制器。

视图

正如上面所提到的，`php artisan make:auth` 命令会在 `resources/views/auth` 目录下创建所有认证需要的视图。
`make:auth` 命令还创建了 `resources/views/layouts` 目录，该目录下包含了应用的基础布局文件。所有这些视图都使用了 Bootstrap CSS 框架，你也可以根据需要对其进行自定义。

认证

现在你已经为自带的认证控制器设置好了路由和视图，接下来我们来实现新用户注册和登录认证。你可以在浏览器中访问定义好的路由，认证控制器默认已经包含了注册及登录逻辑（通过 trait）。

自定义路径

当一个用户成功进行登录认证后，默认将会跳转到`/`，你可以通过在 `AuthController` 中设置 `redirectTo` 属性来自定义登录认证成功之后的跳转路径：

```
protected $redirectTo = '/home';
```

当一个用户登录认证失败后，默认将会跳转回登录表单对应的页面。

自定义 Guard

你还可以自定义实现用户认证的“guard”，要实现这一功能，需要在 `AuthController` 中定义 `guard` 属性，该属性的值对应认证配置文件 `auth.php` 中的相应 guard 配置：

```
protected $guard = 'admin';
```

自定义验证/存储

要修改新用户注册所必需的表单字段，或者自定义新用户字段如何存储到数据库，你可以修改 `AuthController` 类。该类负责为应用验证输入参数和创建新用户。

`AuthController` 的 `validator` 方法包含了新用户的验证规则，你可以按需要自定义该方法。

`AuthController` 的 `create` 方法负责使用 Eloquent ORM 在数据库中创建新的 `App\User` 记录。当然，你也可以基于自己的需要自定义该方法。

获取认证用户

你可以通过 `Auth` 门面访问认证用户：

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

此外，用户通过认证后，你还可以通过 `Illuminate\Http\Request` 实例访问认证用户：

```
<?php

namespace App\Http\Controllers;

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

class ProfileController extends Controller{
    /**
     * 更新用户属性.
     *
     * @param  Request  $request
     * @return Response
     */
    public function updateProfile(Request $request)
    {
        if ($request->user()) {
            // $request->user() 返回认证用户实例...
        }
    }
}
```

判断当前用户是否通过认证

要判断某个用户是否登录到应用，可以使用 `Auth` 门面的 `check` 方法，如果用户通过认证则返回 `true`：

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

此外，你还可以在用户访问特定路由/控制器之前使用中间件来验证用户是否通过认证，想要了解更多，可以查看下面的路由保护：

路由保护

路由中间件可用于只允许通过认证的用户访问给定路由。Laravel 通过定义在 `app\Http\Middleware\Authenticate.php` 的 `auth` 中间件来处理这一操作。你所要做的仅仅是将该中间件加到相应的路由定义中：

```
// 使用路由闭包...
Route::get('profile', ['middleware' => 'auth', function() {
    // 只有认证用户可以进入...
}]);
// 使用控制器...
Route::get('profile', [
    'middleware' => 'auth',
    'uses' => 'ProfileController@show'
]);
```

当然，如果你正在使用控制器类，也可以在控制器的构造方法中调用 `middleware` 方法而不是在路由器中直接定义：

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

指定一个 Guard

添加 `auth` 中间件到路由后，还需要指定使用哪个 guard 来实现认证：

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

指定的 guard 对应配置文件 `auth.php` 中 `guards` 数组的某个键。

登录失败次数限制

如果你使用了 Laravel 内置的 `AuthController` 类， 可以使用 `Illuminate\Foundation\Auth\ThrottlesLogins` trait 来限制用户登录失败次数。默认情况下，用户在几次登录失败后将在一分钟内不能登录，这种限制基于用户的用户名/邮箱地址+IP 地址：

```
<?php

namespace App\Http\Controllers\Auth;
```

```
use App\User;use Validator;
use App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\ThrottlesLogins;
use Illuminate\Foundation\Auth\AuthenticatesAndRegistersUsers;

class AuthController extends Controller{
    use AuthenticatesAndRegistersUsers, ThrottlesLogins;

    // AuthController 类的其它部分...
}
```

3、手动认证用户

当然，你也可以不使用 Laravel 自带的认证控制器。如果你选择移除这些控制器，你需要直接使用 Laravel 认证类来管理用户认证。别担心，这很简单！

我们将会通过 `Auth` 门面来访问认证服务，因此我们需要确保在类的顶部导入了 `Auth` 门面，让我们看看 `attempt` 方法：

```
<?php

namespace App\Http\Controllers;

use Auth;
use Illuminate\Routing\Controller;

class AuthController extends Controller{
    /**
     * 处理登录认证
     *
     * @return Response
     */
    public function authenticate()
    {
        if (Auth::attempt(['email' => $email, 'password' => $pas
sword])) {
            // 认证通过...
            return redirect()->intended('dashboard');
        }
    }
}
```

`attempt` 方法接收键值数组对作为第一个参数，数组中的值被用于从数据表中查找用户，因此，在上面的例子中，用户将会通过 `email` 的值获取，如果用户被找到，经哈希运算后

存储在数据中的密码将会和传递过来的经哈希运算处理的密码值进行比较。如果两个经哈希运算的密码相匹配那么将会为这个用户开启一个认证 Session。
如果认证成功的话 `attempt` 方法将会返回 `true`。否则，返回 `false`。
重定向器上的 `intended` 方法将用户重定向到登录之前用户想要访问的 URL，在目标 URL 无效的情况下备用 URI 将会传递给该方法。

指定额外条件

如果需要的话，除了用户邮件和密码之外还可以在认证查询时添加额外的条件，例如，我们可以验证被标记为有效的用户：

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

注：在这些例子中，并不仅仅限于使用 email 进行登录认证，这里只是作为演示示例，你可以将其修改为数据库中任何其他可用作“username”的字段。

访问指定 Guard 实例

你可以使用 `Auth` 门面的 `guard` 方法指定想要使用的 guard 实例，这种机制允许你在同一个应用中对不同的认证模型或用户表实现完全独立的用户认证。
传递给 `guard` 方法的 guard 名称对应配置文件 `auth.php` 中 `guards` 配置的某个键：

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

退出

要退出应用，可以使用 `Auth` 门面的 `logout` 方法，这将会清除用户 Session 中的认证信息：

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

记住用户

如果你想要在应用中提供“记住我”的功能，可以传递一个布尔值作为第二个参数到 `attempt` 方法，这样用户登录认证状态就会一直保持直到他们手动退出。当然，你的 `users` 表必须包含 `remember_token` 字段，该字段用于存储“记住我”令牌。

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

如果你要“记住”用户，可以使用 `viaRemember` 方法来判断用户是否使用“记住我”cookie 进行认证：

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

其它认证方法

认证一个用户实例

如果你需要将一个已存在的用户实例登录到应用中，可以调用 `Auth` 门面的 `login` 方法并传入用户实例，传入实例必须是 `Illuminate\Contracts\Auth\Authenticatable` 契约的实现，当然，Laravel 自带的 `App\User` 模型已经实现了该接口：

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

通过 ID 认证用户

要通过用户 ID 登录到应用，可以使用 `loginUsingId` 方法，该方法接收你想要认证用户的主键作为参数：

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

一次性认证用户

你可以使用 `once` 方法只在单个请求中将用户登录到应用，而不存储任何 Session 和 Cookie，这在构建无状态的 API 时很有用。`once` 方法和 `attempt` 方法用法差不多：

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

4、基于 HTTP 的基本认证

HTTP 基本认证能够帮助用户快速实现登录认证而不用设置专门的登录页面，首先要在路由中加上 `auth.basic` 中间件。该中间件是 Laravel 自带的，所以不需要自己定义：

```
Route::get('profile', ['middleware' => 'auth.basic', function()
 {
    // 只有认证用户可以进入...
}]);
```

中间件加到路由中后，当在浏览器中访问该路由时，会自动提示需要认证信息，默认情况下，`auth.basic` 中间件使用用户记录上的 `email` 字段作为“用户名”。

FastCGI 上注意点

如果你使用 PHP FastCGI，HTTP 基本认证将不能正常工作，需要在 `.htaccess` 文件加入如下内容：

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

无状态的 HTTP 基本认证

使用 HTTP 基本认证也不需要在 Session 中设置用户标识 Cookie，这在 API 认证中非常有用。要实现这个，需要定义一个调用 `onceBasic` 方法的中间件。如果该方法没有返回任何响应，那么请求会继续走下去：

```
<?php

namespace Illuminate\Auth\Middleware;

use Auth;
use Closure;

class AuthenticateOnceWithBasicAuth{
    /**
     * 处理输入请求.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        return Auth::onceBasic() ?: $next($request);
    }

}
```

接下来，注册路由中间件并将其添加到路由中：

```
Route::get('api/user', ['middleware' => 'auth.basic.once', func
tion() {
    // 只有认证用户可以进入...
}]);
```

5、重置密码

数据库考量

大多数 web 应用提供了用户重置密码的功能，Laravel 提供了便利方法用于发送密码提示及执行密码重置而不需要你在每个应用中重新实现。
开始之前，先验证 `App\User` 模型实现了 `Illuminate\Contracts\Auth\CanResetPassword` 契约。当然，Laravel 自带

的 App\User 模型已经实现了该接口，并使用 Illuminate\Auth\Passwords\CanResetPassword trait 来包含实现该接口需要的方法。

生成重置令牌表迁移

接下来，用来存储密码重置令牌的表必须被创建，Laravel 已经自带了这张表的迁移，就存放在 database/migrations 目录。所有，你所要做的仅仅是运行迁移：

```
php artisan migrate
```

路由

Laravel 自带了 Auth\PasswordController，其中包含了重置用户密码的相应逻辑。重置密码所需的路由都已经通过 make:auth 命令自动生成了：

```
php artisan make:auth
```

视图

和路由一样，重置密码所需的视图文件也通过 make:auth 命令一并生成了，这些视图文件位于 resources/views/auth/passwords 目录下，你可以按照所需对生成的文件进行相应修改。

重置密码后

定义好重置用户密码路由和视图后，只需要在浏览器中访问这些路由即可。框架自带的 PasswordController 已经包含了发送密码重置链接邮件以及更新数据库中密码的逻辑。密码被重置后，用户将会自动登录到应用并重定向到 /home。你可以通过定义上 PasswordController 的 redirectTo 属性来自定义密码重置成功后的跳转链接：

```
protected $redirectTo = '/dashboard';
```

注意：默认情况下，密码重置令牌一小时内有效，你可以通过修改 config/auth.php 文件中的选项 reminder.expire 来改变有效时间。

自定义

自定义认证 Guard

在配置文件 auth.php 中，可以配置多个"guards"，以便用于实现多用户表独立认证，你可以通过添加 $guard 属性到自带的 PasswordController 控制器的方法来使用你选择的 guard：

```
/**
* The authentication guard that should be used.
*
* @var string
*/
protected $guard = 'admins';
```

自定义密码 broker

在配置文件 `auth.php` 中，可以配置多个密码，以便用于重置多个用户表的密码 broker，同样，可以通过在自带的 `PasswordController` 控制器中添加 `$broker` 属性来使用你选择的 broker：

```
/**
* The password broker that should be used.
*
* @var string
*/
protected $broker = 'admins';
```

6、社会化登录认证

Laravel 中还可以使用 Laravel Socialite 通过 OAuth 提供者进行简单、方便的认证，也就是社会化登录，目前支持使用 Facebook、Twitter、LinkedIn、GitHub 和 Bitbucket 进行登录认证。

要使用社会化登录，需要在 `composer.json` 文件中添加依赖：

```
composer require laravel/socialite
```

配置

安装完社会化登录库后，在配置文件 `config/app.php` 中注册 `Laravel\Socialite\SocialiteServiceProvider`：

```
'providers' => [
    // 其它服务提供者...
    Laravel\Socialite\SocialiteServiceProvider::class,
],
```

还要在 `app` 配置文件中添加 `Socialite` 门面到 `aliases` 数组：

```
'Socialite' => Laravel\Socialite\Facades\Socialite::class,
```

你还需要为应用使用的 OAuth 服务添加认证信息，这些认证信息位于配置文件 `config/services.php`，而且键为 `facebook`, `twitter`,`linkedin`, `google`, `github` 或 `bitbucket`，这取决于应用需要的提供者。例如：

```
'github' => [
    'client_id' => 'your-github-app-id',
    'client_secret' => 'your-github-app-secret',
    'redirect' => 'http://your-callback-url',
],
```

基本使用

接下来，准备好认证用户！你需要两个路由：一个用于重定向用户到 OAuth 提供者，另一个用户获取认证后来自提供者的回调。我们使用 `Socialite` 门面访问 Socialite：

```
<?php

namespace App\Http\Controllers;

use Socialite;
use Illuminate\Routing\Controller;

class AuthController extends Controller{
    /**
     * 将用户重定向到 GitHub 认证页面.
     *
     * @return Response
     */
    public function redirectToProvider()
    {
        return Socialite::driver('github')->redirect();
    }

    /**
     * 从 GitHub 获取用户信息.
     *
     * @return Response
     */
    public function handleProviderCallback()
    {
        $user = Socialite::driver('github')->user();

        // $user->token;
    }
}
```

`redirect` 方法将用户发送到 OAuth 提供者，`user` 方法读取请求信息并从提供者中获取用户信息，在重定向用户之前，你还可以在请求上使用 `scope` 方法设置"作用域"，该方法将会重写已存在的所有作用域：

```
return Socialite::driver('github')
            ->scopes(['scope1', 'scope2'])->redirect();
```

当然，你需要定义路由到控制器方法：

```
 Route::get('auth/github', 'Auth\AuthController@redirectToProvi
der');
 Route::get('auth/github/callback', 'Auth\AuthController@handle
ProviderCallback');
```

获取用户信息

有了用户实例之后，可以获取用户的更多详情：

```
$user = Socialite::driver('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();
```

7、添加自定义的 Guard

你可以通过 `Auth` 门面的 `extend` 方法定义自己的认证 guard，该功能需要在某个服务提供者的 `boot` 方法中实现：

```
<?php

namespace App\Providers;

use Auth;
use App\Services\Auth\JwtGuard;
use Illuminate\Support\ServiceProvider;

class AuthServiceProvider extends ServiceProvider{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Auth::extend('jwt', function($app, $name, array $config) {
```

```
            // Return an instance of Illuminate\Contracts\Auth\Guard...

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

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

正如你在上述例子中所看到的，传递给 `extend` 方法的闭包回调需要返回 `Illuminate\Contracts\Auth\Guard` 的实现实例，该接口包含了自定义认证 guard 需要的一些方法。

定义好自己的认证 guard 之后，可以在配置文件的 `guards` 配置中使用话这个 guard：

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

8、添加自定义用户提供者

如果你没有使用传统的关系型数据库存储用户信息，则需要使用自己的认证用户提供者来扩展 Laravel。我们使用 Auth 门面上的 `provider` 方法定义自定义该提供者，在服务提供者调用 `provider` 方法如下：

```
<?php

namespace App\Providers;

use Auth;
use App\Extensions\RiakUserProvider;
use Illuminate\Support\ServiceProvider;

class AuthServiceProvider extends ServiceProvider{
```

```
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Auth::provider('riak', function($app,array $config) {
            // 返回 Illuminate\Contracts\Auth\UserProvider 实例...
            return new RiakUserProvider($app['riak.connection']);
        });
    }

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

通过 `provider` 方法注册用户提供者后，你可以在配置文件 `config/auth.php` 中切换到新的用户提供者。首先，在该配置文件定义一个使用新驱动的 `providers` 数组：

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

然后，可以在你的 `guards` 配置中使用这个提供者：

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

UserProvider 契约

`Illuminate\Contracts\Auth\UserProvider` 实现只负责从持久化存储系统中获取 `Illuminate\Contracts\Auth\Authenticatable` 实现，例如 MySQL、Riak 等等。这两个接口允许 Laravel 认证机制继续起作用而不管用户数据如何存储或者何种类来展现。

让我们先看看 `Illuminate\Contracts\Auth\UserProvider` 契约：

```
<?php

namespace Illuminate\Contracts\Auth;

interface UserProvider {

    public function retrieveById($identifier);
    public function retrieveByToken($identifier, $token);
    public function updateRememberToken(Authenticatable $user,
$token);
    public function retrieveByCredentials(array $credentials);
    public function validateCredentials(Authenticatable $user,
array $credentials);

}
```

`retrieveById` 方法通常获取一个代表用户的键，例如 MySQL 数据中的自增 ID。该方法获取并返回匹配该 ID 的 `Authenticatabl` 实现。

`retrieveByToken` 函数通过唯一标识和存储在 `remember_token` 字段中的“记住我”令牌获取用户。和上一个方法一样，该方法也返回 `Authenticatabl` 实现。

`updateRememberToken` 方法使用新的 `$token` 更新 `$user` 的 `remember_token` 字段，新令牌可以是新生成的令牌（在登录是选择“记住我”被成功赋值）或者 null（用户退出）。

`retrieveByCredentials` 方法在尝试登录系统时获取传递给 `Auth::attempt` 方法的认证信息数组。该方法接下来去底层持久化存储系统查询与认证信息匹配的用户，通常，该方法运行一个带“where”条件（`$credentials['username']`）的查询。然后该方法返回 `UserInterface` 的实现。这个方法不做任何密码校验和认证。

`validateCredentials` 方法比较给定 `$user` 和`$credentials` 来认证用户。例如，这个方法比较 `$user->getAuthPassword()` 字符串和经 `Hash::make` 处理的 `$credentials['password']`。这个方法只验证用户认证信息并返回布尔值。

Authenticatable 契约

既然我们已经探索了 `UserProvider` 上的每一个方法，接下来让我们看看 `Authenticatable`。该提供者应该从 `retrieveById` 和 `retrieveByCredentials` 方法中返回接口实现：

```
<?php

namespace Illuminate\Contracts\Auth;

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，`getAuthIdentifier` 方法返回用户“主键”，在 MySQL 后台中是 ID，`getAuthPassword` 返回经哈希处理的用户密码，这个接口允许认证系统处理任何用户类，不管是你使用的是 ORM 还是存储抽象层。默认情况下，Laravel 自带的 `app` 目录下的 `User` 类实现了这个接口，所以你可以将这个类作为实现例子。

9、事件

Laravel 支持在认证过程中触发多种事件，你可以在自己的 `EventServiceProvider` 中监听这些事件：

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Auth\Events\Attempting' => [
        'App\Listeners\LogAuthenticationAttempt',
    ],

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

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

用户授权

1、简介

除了提供开箱即用的认证服务之外，Laravel 还提供了一个简单的方式来管理授权逻辑以便控制对资源的访问权限。在 Laravel 中，有多种方法和辅助函数来协助你管理授权逻辑，本文档将会一一覆盖这些方法。

2、定义权限（Abilities）

判断用户是否有权限执行给定动作的最简单方式就是使用 `Illuminate\Auth\Access\Gate` 类来定义一个“权限”。我们在 `AuthServiceProvider` 中定义所有权限，例如，我们来定义一个接收当前 `User` 和 `Post` 模型的 `update-post` 权限，在该权限中，我们判断用户 `id` 是否和文章的 `user_id` 匹配：

```
<?php

namespace App\Providers;

use Illuminate\Contracts\Auth\Access\Gate as GateContract;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider
 as ServiceProvider;

class AuthServiceProvider extends ServiceProvider{
    /**
     * 注册应用所有的认证/授权服务.
     *
     * @param  \Illuminate\Contracts\Auth\Access\Gate  $gate
     * @return void
     */
    public function boot(GateContract $gate)
    {
        parent::registerPolicies($gate);

        $gate->define('update-post', function ($user, $post) {
            return $user->id === $post->user_id;
        });
    }
}
```

注意我们并没有检查给定 `$user` 是否为 `NULL`，当用户未经过登录认证或者用户没有通过 `forUser` 方法指定，`Gate` 会自动为所有权限返回 `false`。

基于类的权限

除了注册授权回调闭包之外，还可以通过传递包含权限类名和类方法的方式来注册权限方法，当需要的时候，该类会通过服务容器进行解析：

```
$gate->define('update-post', 'PostPolicy@update');
```

拦截认证检查

有时候，你可能希望对指定用户授予所有权限，在这种场景中，需要使用 `before` 方法定义一个在所有其他授权检查之前运行的回调：

```
$gate->before(function ($user, $ability) {
    if ($user->isSuperAdmin()) {
        return true;
    }
});
```

如果 `before` 回调返回一个非空结果，那么该结果则会被当做检查的结果。
你也可以使用 `after` 方法定义一个在所有其他授权检查之后运行的回调，所不同的是，在 `after` 回调中你不能编辑授权检查的结果：

```
$gate->after(function ($user, $ability, $result, $arguments) {
    //
});
```

3、检查权限（Abilities）

通过 Gate 门面

权限定义好之后，可以使用多种方式来“检查”。首先，可以使用 `Gate` 门面的 `check`, `allows`, 或者 `denies` 方法。所有这些方法都接收权限名和传递给该权限回调的参数作为参数。你不需要传递当前用户到这些方法，因为 `Gate` 会自动附加当前用户到传递给回调的参数，因此，当检查我们之前定义的 `update-post` 权限时，我们只需要传递一个 `Post` 实例到 `denies` 方法：

```
<?php

namespace App\Http\Controllers;

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

class PostController extends Controller{
    /**
     * 更新给定文章
     *
     * @param  int  $id
     * @return Response
     */
    public function update($id)
    {
        $post = Post::findOrFail($id);
```

```
        if (Gate::denies('update-post', $post)) {
            abort(403);
        }

        // 更新文章...
    }
}
```

当然，`allows` 方法和 `denies` 方法是相对的，如果动作被授权会返回 `true`，`check` 方法是 `allows` 方法的别名。

为指定用户检查权限

如果你想要使用 `Gate` 门面判断非当前用户是否有权限，可以使用 `forUser` 方法：

```
if (Gate::forUser($user)->allows('update-post', $post)) {
    //
}
```

传递多个参数

当然，权限回调还可以接收多个参数：

```
Gate::define('delete-comment', function ($user, $post, $commen
t) {
    //
});
```

如果权限需要多个参数，简单传递参数数组到 `Gate` 方法：

```
if (Gate::allows('delete-comment', [$post, $comment])) {
    //
}
```

通过 User 模型

还可以通过 `User` 模型实例来检查权限。默认情况下，Laravel 的 `App\User` 模型使用一个 `Authorizable` trait 来提供两种方法：`can` 和 `cannot`。这两个方法的功能和 `Gate` 门面上的 `allows` 和 `denies` 方法类似。因此，使用我们前面的例子，可以修改代码如下：

```
<?php

namespace App\Http\Controllers;

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

```
class PostController extends Controller{
    /**
     * 更新给定文章
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  int  $id
     * @return Response
     */
    public function update(Request $request, $id)
    {
        $post = Post::findOrFail($id);

        if ($request->user()->cannot('update-post', $post)) {
            abort(403);
        }

        // 更新文章...
    }
}
```

当然，`can` 方法和 `cannot` 方法相反：

```
if ($request->user()->can('update-post', $post)) {
    // 更新文章...
}
```

在 Blade 模板引擎中检查

为了方便，Laravel 提供了 Blade 指令 `@can` 来快速检查当前用户是否有指定权限。例如：

```
<a href="/post/{{ $post->id }}">View Post</a>

@can('update-post', $post)
    <a href="/post/{{ $post->id }}/edit">Edit Post</a>
@endcan
```

你还可以将 `@can` 指令和 `@else` 指令联合起来使用：

```
@can('update-post', $post)
    <!-- The Current User Can Update The Post -->
@else
    <!-- The Current User Can't Update The Post -->
@endcan
```

在表单请求中检查

你还可以选择在表单请求的 `authorize` 方法中使用 `Gate` 定义的权限。例如：

```
/**
 * 判断请求用户是否经过授权
 *
 * @return bool
 */
public function authorize(){
    $postId = $this->route('post');
    return Gate::allows('update', Post::findOrFail($postId));
}
```

4、策略类（Policies）

创建策略类

由于在 `AuthServiceProvider` 中定义所有的授权逻辑将会变得越来越臃肿笨重，尤其是在大型应用中，所以 Laravel 允许你将授权逻辑分割到多个“策略”类中，策略类是原生的 PHP 类，基于授权资源对授权逻辑进行分组。
首先，让我们生成一个策略类来管理对 `Post` 模型的授权，你可以使用 Artisan 命令 `make:policy` 来生成该策略类。生成的策略类位于 `app/Policies` 目录：

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

注册策略类

策略类生成后我们需要将其注册到 `Gate` 类。`AuthServiceProvider` 包含了一个 `policies` 属性来映射实体及管理该实体的策略类。因此，我们指定 `Post` 模型的策略类是 `PostPolicy`：

```
<?php

namespace App\Providers;

use App\Post;
use App\Policies\PostPolicy;
use Illuminate\Contracts\Auth\Access\Gate as GateContract;
use Illuminate\Foundation\Support\Providers\AuthServiceProvide
r as ServiceProvider;

class AuthServiceProvider extends ServiceProvider{
    /**
     * 应用的策略映射
     *
     * @var array
     */
```

```
    protected $policies = [
        Post::class => PostPolicy::class,
    ];

    /**
     * 注册所有应用认证/授权服务
     *
     * @param \Illuminate\Contracts\Auth\Access\Gate $gate
     * @return void
     */
    public function boot(GateContract $gate)
    {
        $this->registerPolicies($gate);
    }
}
```

编写策略类

策略类生成和注册后，我们可以为授权的每个权限添加方法。例如，我们在 `PostPolicy` 中定义一个 `update` 方法，该方法判断给定 `User` 是否可以更新某个 `Post`：

```
<?php

namespace App\Policies;

use App\User;
use App\Post;

class PostPolicy{
    /**
     * 判断给定文章是否可以被给定用户更新
     *
     * @param  \App\User  $user
     * @param  \App\Post  $post
     * @return bool
     */
    public function update(User $user, Post $post)
    {
        return $user->id === $post->user_id;
    }
}
```

你可以继续在策略类中为授权的权限定义更多需要的方法，例如，你可以定义 `show`, `destroy`, 或者 `addComment` 方法来认证多个 `Post` 动作。

注意：所有策略类都通过服务容器进行解析，这意味着你可以在策略类的构造函数中类型提示任何依赖，它们将会自动被注入。

拦截所有检查

有时候，你可能希望对指定用户授予所有权限，在这种场景中，需要使用 `before` 方法定义一个在所有其他授权检查之前运行的回调：

```
$gate->before(function ($user, $ability) {
    if ($user->isSuperAdmin()) {
        return true;
    }
});
```

如果 `before` 回调返回一个非空结果，那么该结果则会被当做检查的结果。

检查策略

策略类方法的调用方式和基于授权回调的闭包一样，你可以使用 `Gate` 门面，`User` 模型，`@can` 指令或者辅助函数 `policy`。

通过 Gate 门面

`Gate` 将会自动通过检测传递过来的类参数来判断使用哪一个策略类，因此，如果传递一个 `Post` 实例给 `denies` 方法，相应的，`Gate` 会使用 `PostPolicy` 来进行动作授权：

```
<?php

namespace App\Http\Controllers;

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

class PostController extends Controller{
    /**
     * 更新给定文章
     *
     * @param  int  $id
     * @return Response
     */
    public function update($id)
    {
        $post = Post::findOrFail($id);

        if (Gate::denies('update', $post)) {
```

```
            abort(403);
        }

        // 更新文章...
    }
}
```

通过 User 模型

User 模型的 can 和 cannot 方法也会自动判断给定参数对应的策略类。这些方法提供了便利的方式来为应用接收到的任意 User 实例进行授权：

```
if ($user->can('update', $post)) {
    //
}

if ($user->cannot('update', $post)) {
    //
}
```

在 Blade 模板中使用

类似的，Blade 指令 @can 将会使用参数中有效的策略类：

```
@can('update', $post)
    <!-- The Current User Can Update The Post -->
@endcan
```

通过辅助函数 policy

全局的辅助函数 policy 用于为给定类实例接收策略类。例如，我们可以传递一个 Post 实例给帮助函数 policy 来获取相应的 PostPolicy 类的实例：

```
if (policy($post)->update($user, $post)) {
    //
}
```

5、控制器授权

默认情况下，Laravel 自带的控制器基类 App\Http\Controllers\Controller 使用了 AuthorizesRequests trait，该 trait 提供了可用于快速授权给定动作的 authorize 方法，如果授权不通过，则抛出 HttpException 异常。
该 authorize 方法和其他多种授权方法使用方法一致，例如 Gate::allows 和 $user->can()。因此，我们可以这样使用 authorize 方法快速授权更新 Post 的请求：

```
<?php
```

```
namespace App\Http\Controllers;

use App\Post;
use App\Http\Controllers\Controller;

class PostController extends Controller{
    /**
     * 更新给定文章
     *
     * @param  int  $id
     * @return Response
     */
    public function update($id)
    {
        $post = Post::findOrFail($id);

        $this->authorize('update', $post);

        // 更新文章...
    }
}
```

如果授权成功，控制器继续正常执行；然而，如果 `authorize` 方法判断该动作授权失败，将会抛出 `HttpException` 异常并生成带 `403 Not Authorized` 状态码的 HTTP 响应。正如你所看到的，`authorize` 方法是一个授权动作、抛出异常的便捷方法。

`AuthorizesRequests` trait 还提供了 `authorizeForUser` 方法用于授权非当前用户：

```
$this->authorizeForUser($user, 'update', $post);
```

自动判断策略类方法

通常，一个策略类方法对应一个控制器上的方法，例如，在上面的 `update` 方法中，控制器方法和策略类方法共享同一个方法名：`update`。

正是因为这个原因，Laravel 允许你简单传递实例参数到 `authorize` 方法，被授权的权限将会自动基于调用的方法名进行判断。在本例中，由于 `authorize` 在控制器的 `update` 方法中被调用，那么对应的，`PostPolicy` 上 `update` 方法将会被调用：

```
/**
 * 更新给定文章
 *
 * @param  int  $id
 * @return Response
 */
public function update($id){
    $post = Post::findOrFail($id);
```

```
    $this->authorize($post);

    // 更新文章...
}
```

Artisan Console

1、简介

Artisan 是 Laravel 自带的命令行接口名称，它为我们在开发过程中提供了很多有用的命令。通过强大的 Symfony Console 组件驱动。想要查看所有可用的 Artisan 命令，可使用 `list` 命令：

```
php artisan list
```

每个命令都可以用 `help` 指令显示命令描述及命令参数和选项。想要查看帮助界面，只需要在命令前加上 `help` 就可以了：

```
php artisan help migrate
```

2、编写命令

除了 Artisan 提供的命令之外，还可以构建自己的命令。你可以将自定义命令存放在 `app/Console/Commands` 目录；当然，你可以自己选择存放位置，只要该命令可以被 `composer.json` 自动加载即可。
要创建一个新命令，你可以使用 Artisan 命令 `make:console`：

```
php artisan make:console SendEmails
```

上述命令将会生成一个类 `app/Console/Commands/SendEmails.php`，当创建命令时，`--command` 选项可用于分配终端命令名（在终端调用命令时用）：

```
php artisan make:console SendEmails --command=emails:send
```

命令结构

命令生成以后，需要填写该类的 `signature` 和 `description` 属性，这两个属性在调用 `list` 显示命令的时候会被用到。
`handle` 方法在命令执行时被调用，你可以将所有命令逻辑都放在这个方法里面，让我们先看一个命令例子。
我们可以在命令控制器的构造函数中注入任何依赖，Laravel 服务提供者将会在构造函数中自动注入所有依赖类型提示。要增强代码的复用性，保持代码轻量级并让它们延迟到应用服务中完成任务是个不错的实践：

```
<?php
```

```
namespace App\Console\Commands;

use App\User;
use App\DripEmailer;
use Illuminate\Console\Command;
use Illuminate\Foundation\Inspiring;

class Inspire extends Command{
    /**
     * 控制台命令名称
     *
     * @var string
     */
    protected $signature = 'email:send {user}';

    /**
     * 控制台命令描述
     *
     * @var string
     */
    protected $description = 'Send drip e-mails to a user';

    /**
     * The drip e-mail service.
     *
     * @var DripEmailer
     */
    protected $drip;

    /**
     * 创建新的命令实例
     *
     * @param  DripEmailer  $drip
     * @return void
     */
    public function __construct(DripEmailer $drip)
    {
        parent::__construct();
        $this->drip = $drip;
    }

    /**
     * 执行控制台命令
```

```
     *
     * @return mixed
     */
    public function handle()
    {
        $this->drip->send(User::find($this->argument('user')));
    }
}
```

3、命令 I/O

3.1 定义期望输入

编写控制台命令的时候，通常通过参数和选项收集用户输入，Laravel 使这项操作变得很方便：在命令中使用 `signature` 属性来定义我们期望的用户输入。`signature` 属性通过一个优雅的、路由风格的语法允许你定义命令的名称、参数以及选项。所有用户提供的参数和选项都包含在大括号里，下面这个例子定义的命令要求用户输入必选参数 user：

```
/**
 * 控制台命令名称
 *
 * @var string
 */
protected $signature = 'email:send {user}';
```

你还可以让该参数可选可不选，以及定义默认的可选参数值：

```
// 选项参数...
email:send {user?}
// 带默认值的选项参数...
email:send {user=foo}
```

选项，和参数一样，也是用户输入的一种格式，不同之处在于选项前面有两个短划线（--），我们可以这样定义选项：

```
/**
 * 控制台命令名称
 *
 * @var string
 */
protected $signature = 'email:send {user} {--queue}';
```

在本例中，`--queue` 开关在调用 Artisan 命令的时候被指定。如果`--queue` 开关被传递，其值是 `true`，否则其值是 `false`：

```
php artisan email:send 1 --queue
```

你还可以指定选项值被用户通过=来分配：

```
/**
 * 控制台命令名称
 *
 * @var string
 */
protected $signature = 'email:send {user} {--queue=}';
```

在本例中，用户可以通过这样的方式传值：

```
php artisan email:send 1 --queue=default
```

还可以给选项分配默认值：

```
email:send {user} {--queue=default}
```

如果想要为命令选项分配一个简写，可以在选项前指定并使用分隔符|将简写和完整选项名分开：

```
email:send {user} {--Q|queue}
```

如果你想要定义参数和选项以便指定输入数组，可以使用字符*：

```
email:send {user*}
email:send {user} {--id=*}
```

输入描述

你可以通过冒号将参数和描述进行分隔的方式分配描述到输入参数和选项：

```
/**
 * 控制台命令名称
 *
 * @var string
 */
protected $signature = 'email:send
    {user : The ID of the user}
    {--queue= : Whether the job should be queued}';
```

3.2 获取输入

在命令被执行的时候，很明显，你需要访问命令获取的参数和选项的值。使用 `argument` 和 `option` 方法即可实现：
要获取参数的值，通过 `argument` 方法：

```
/**
 * 执行控制台命令
 *
 * @return mixed
 */
public function handle(){
    $userId = $this->argument('user');
}
```

如果你需要以数组形式获取所有参数值，使用不带参数的 `argument`：

```
$arguments = $this->argument();
```

选项值和参数值的获取一样简单，使用 `option` 方法，同 `argument` 一样如果要获取所有选项值，可以调用不带参数的 `option` 方法：

```
// 获取指定选项...
$queueName = $this->option('queue');
// 获取所有选项...
$options = $this->option();
```

如果参数或选项不存在，返回 null。

3.3 输入提示

除了显示输出之外，你可能还要在命令执行期间要用户提供输入。`ask` 方法将会使用给定问题提示用户，接收输入，然后返回用户输入到命令：

```
/**
 * 执行控制台命令
 *
 * @return mixed
 */
public function handle(){
    $name = $this->ask('What is your name?');
}
```

`secret` 方法和 `ask` 方法类似，但用户输入在终端对他们而言是不可见的，这个方法在问用户一些敏感信息如密码时很有用：

```
$password = $this->secret('What is the password?');
```

让用户确认

如果你需要让用户确认信息，可以使用 `confirm` 方法，默认情况下，该方法返回 `false`，如果用户输入 `y`，则该方法返回 `true`：

```
if ($this->confirm('Do you wish to continue? [y|N]')) {
    //
```

```
}
```

给用户提供选择

`anticipate` 方法可用于为可能的选项提供自动完成功能，用户仍然可以选择答案，而不管这些选择：

```
$name = $this->anticipate('What is your name?', ['Taylor', 'Dayle']);
```

如果你需要给用户预定义的选择，可以使用 `choice` 方法。用户选择答案的索引，但是返回给你的是答案的值。如果用户什么都没选的话你可以设置默认返回的值：

```
$name = $this->choice('What is your name?', ['Taylor', 'Dayle'], false);
```

3.4 编写输出

要将输出发送到控制台，使用 line,`info`, `comment`, `question` 和 `error` 方法，每个方法都会使用相应的 ANSI 颜色以作标识。
要显示一条信息消息给用户，使用 `info` 方法。通常，在终端显示为绿色：

```
/**
 * 执行控制台命令
 *
 * @return mixed
 */
public function handle(){
    $this->info('Display this on the screen');
}
```

要显示一条错误消息，使用 `error` 方法。错误消息文本通常是红色：

```
$this->error('Something went wrong!');
```

如果你想要显示原生输出，可以使用 line 方法，该方法输出的字符不带颜色：

```
$this->line('Display this on the screen');
```

表格布局

`table` 方法使输出多行/列格式的数据变得简单，只需要将头和行传递给该方法，宽度和高度将基于给定数据自动计算：

```
$headers = ['Name', 'Email'];
$users = App\User::all(['name', 'email'])->toArray();
$this->table($headers, $users);
```

进度条

对需要较长时间运行的任务，显示进度指示器很有用，使用该输出对象，我们可以开始、前进以及停止该进度条。在开始进度时你必须定义步数，然后每走一步进度条前进一格：

```
$users = App\User::all();

$this->output->progressStart(count($users));

foreach ($users as $user) {
    $this->performTask($user);
    $this->output->progressAdvance();
}

$this->output->progressFinish();
```

想要了解更多，查看 Symfony 进度条组件文档。

4、注册命令

命令编写完成后，需要注册到 Artisan 才可以使用，这可以在 `app/Console/Kernel.php` 文件中完成。
在该文件中，你会在 `commands` 属性中看到一个命令列表，要注册你的命令，只需将其加到该列表中即可。当 Artisan 启动的时候，该属性中列出的命令将会被服务容器解析被注册到 Artisan：

```
protected $commands = [
    'App\Console\Commands\SendEmails'
];
```

5、通过代码调用命令

有时候你可能希望在 CLI 之外执行 Artisan 命令，比如，你可能希望在路由或控制器中触发 Artisan 命令，你可以使用 Artisan 门面上的 `call` 方法来完成这个。`call` 方法接收被执行的命令名称作为第一个参数，命令参数数组作为第二个参数，退出代码被返回：

```
Route::get('/foo', function () {
    $exitCode = Artisan::call('email:send', [
        'user' => 1, '--queue' => 'default'
    ]);
});
```

使用 Artisan 上的 `queue` 方法，你甚至可以将 Artisan 命令放到队列中，这样它们就可以通过后台的队列工作者来处理：

```
Route::get('/foo', function () {
    Artisan::queue('email:send', [
        'user' => 1, '--queue' => 'default'
    ]);
```

```
});
```

如果你需要指定不接收字符串的选项值，例如 `migrate:refresh` 命令上的`--force` 标识，可以传递布尔值 `true` 或 `false`：

```
$exitCode = Artisan::call('migrate:refresh', [
    '--force' => true,
]);
```

通过其他命令调用命令

有时候你希望从一个已存在的 Artisan 命令中调用其它命令。你可以通过使用 `call` 方法开实现这一目的。`call` 方法接收命令名称和数组形式的命令参数：

```
/**
 * 执行控制台命令
 *
 * @return mixed
 */
public function handle(){
    $this->call('email:send', [
        'user' => 1, '--queue' => 'default'
    ]);
}
```

如果你想要调用其它控制台命令并阻止其所有输出，可以使用 `callSilent` 方法。`callSilent` 方法和 `call` 方法用法一致：

```
$this->callSilent('email:send', [
    'user' => 1, '--queue' => 'default'
]);
```

订阅支付实现：Laravel Cashier

1、简介

Laravel Cashier 为通过 Stripe 实现订阅支付服务提供了一个优雅平滑的接口。它封装了几乎所有你恐惧编写的样板化的订阅支付代码。除了基本的订阅管理外，Cashier 还支持处理优惠券、订阅升级/替换、订阅“数量”、取消宽限期，甚至生成 PDF 发票。

1.1 安装&配置

Composer

首先，添加 Cashier 包到 `composer.json` 文件并运行 `composer update` 命令：

```
"laravel/cashier": "~6.0"
```

服务提供者

接下来，在 `config/app.php` 配置文件中注册服务提供者：`Laravel\Cashier\CashierServiceProvider`。

迁移

使用 Cashier 之前，我们需要准备好数据库。我们需要添加一个字段到 `users` 表，还要创建新的 `subscriptions` 表来处理所有用户订阅：

```
Schema::table('users', function ($table) {
    $table->string('stripe_id')->nullable();
    $table->string('card_brand')->nullable();
    $table->string('card_last_four')->nullable();
});

Schema::create('subscriptions', function ($table) {
    $table->increments('id');
    $table->integer('user_id');
    $table->string('name');
    $table->string('stripe_id');
    $table->string('stripe_plan');
    $table->integer('quantity');
    $table->timestamp('trial_ends_at')->nullable();
    $table->timestamp('ends_at')->nullable();
    $table->timestamps();
});
```

创建好迁移后，只需简单运行 `migrate` 命令，相应修改就会更新到数据库。

设置模型

接下来，添加 `Billable` trait 到 `User` 模型类：

```
use Laravel\Cashier\Billable;

class User extends Authenticatable{
    use Billable;
}
```

Stripe 键

最后，在配置文件 `config/services.php` 中设置 Stripe 键：

```
'stripe' => [
    'model'  => 'User',
    'secret' => env('STRIPE_API_SECRET'),
],
```

2、订阅实现

2.1 创建订阅

要创建一个订阅，首先要获取一个账单模型的实例，通常是 `App\User` 的实例。获取到该模型实例之后，你可以使用 `newSubscription` 方法来创建该模型的订阅：

```
$user = User::find(1);
$user->newSubscription('main', 'monthly')->create($creditCardToken);
```

第一个传递给 `newSubscription` 方法的参数是该订阅的名字，如果应用只有一个订阅，可以将其称作 `main` 或 `primary`，第二个参数用于指定用户订阅的 Stripe 计划，该值对应 Stripe 中相应计划的 id。

`create` 方法会自动创建这个 Stripe 订阅，同时更新数据库中 Stripe 的客户 ID（即 `users` 表中的 `stripe_id`）和其它相关的账单信息。如果你的订阅计划有试用期，试用期结束时间也会自动被设置到数据库相应字段。

额外的用户信息

如果你想要指定额外的客户信息，你可以将其作为第二个参数传递给 `create` 方法：

```
$user->newSubscription('main', 'monthly')->create($creditCardToken, [
    'email' => $email,
    'description' => 'Our First Customer'
]);
```

要了解更多 Stripe 支持的字段，可以查看 Stripe 关于创建消费者的文档。

优惠券

如果你想要在创建订阅的时候使用优惠券，可以使用 `withCoupon` 方法：

```
$user->newSubscription('main', 'monthly')
     ->withCoupon('code')
     ->create($creditCardToken);
```

2.2 检查订阅状态

用户订阅你的应用后，你可以使用各种便利的方法来简单检查订阅状态。首先，如果用户有一个有效的订阅，则 `subscribed` 方法返回 `true`，即使订阅现在出于试用期：

```
if ($user->subscribed('main')) {
    //
}
```

`subscribed` 方法还可以用于路由中间件，基于用户订阅状态允许你对路由和控制器的访问进行过滤：

```
public function handle($request, Closure $next){
```

```
    if ($request->user() && ! $request->user()->subscribed('mai
n')) {
        // This user is not a paying customer...
        return redirect('billing');
    }

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

如果你想要判断一个用户是否还在试用期，可以使用 `onTrial` 方法，该方法在为还处于试用期的用户显示警告信息很有用：

```
if ($user->->subscription('main')->onTrial()) {
    //
}
```

`onPlan` 方法可用于判断用户是否基于 Stripe ID 订阅了给定的计划：

```
if ($user->onPlan('monthly')) {
    //
}
```

已取消的订阅状态

要判断用户是否曾经是有效的订阅者，但现在取消了订阅，可以使用 `cancelled` 方法：

```
if ($user->subscription('main')->cancelled()) {
    //
}
```

你还可以判断用户是否曾经取消过订阅，但现在仍然在“宽限期”直到完全失效。例如，如果一个用户在 3 月 5 号取消了一个实际有效期到 3 月 10 号的订阅，该用户处于“宽限期”直到 3 月 10 号。注意 `subscribed` 方法在此期间仍然返回 `true`。

```
if ($user->subscription('main')->onGracePeriod()) {
    //
}
```

2.3 修改订阅

用户订阅应用后，偶尔想要改变到新的订阅计划，要将用户切换到新的订阅，使用 `swap` 方法。例如，我们可以轻松切换用户到 `premium` 订阅：

```
$user = App\User::find(1);
$user->subscription('main')->swap('stripe-plan-id');
```

如果用户在试用，试用期将会被维护。还有，如果订阅存在数量，数量也可以被维护。切换订阅计划后，
可以使用 `invoice` 方法立即给用户开发票：

```
$user->subscription('main')->swap('stripe-plan-id');$user->inv
oice();
```

2.4 订阅数量

有时候订阅也会被数量影响，例如，应用中每个账户每月需要付费$10，要简单增加或减少订阅数量，使用 `incrementQuantity` 和 `decrementQuantity` 方法：

```
$user = User::find(1);

$user->subscription('main')->incrementQuantity();

// Add five to the subscription's current quantity...
$user->subscription('main')->incrementQuantity(5);

$user->subscription('main')->decrementQuantity();

// Subtract five to the subscription's current quantity...
$user->subscription('main')->decrementQuantity(5);
```

你也可以使用 `updateQuantity` 方法指定数量：

```
$user->subscription('main')->updateQuantity(10);
```

想要了解更多订阅数量信息，查阅相关 Stripe 文档。

2.5 订阅税金

在 Cashier 中，提供 `tax_percent` 值发送给 Stripe 很简单。要指定用户支付订阅的税率，实现账单模型的 `getTaxPercent` 方法，并返回一个在 0 到 100 之间的数值，不要超过两位小数：

```
public function getTaxPercent() {
    return 20;
}
```

这将使你可以在模型基础上使用税率，对跨越不同国家的用户很有用。

2.6 取消订阅

要取消订阅，可以调用用户订阅上的 `cancel` 方法：

```
$user->subscription('main')->cancel();
```

当订阅被取消时，Cashier 将会自动设置数据库中的 `subscription_ends_at` 字段。该字段用于了解 `subscribed` 方法什么时候开始返回 `false`。例如，如果客户 3 月 1 号份取消订阅，但订阅直到 3 月 5 号才会结束，那么 `subscribed` 方法继续返回 `true` 直到 3 月 5 号。你可以使用 `onGracePeriod` 方法判断用户是否已经取消订阅但仍然在“宽限期”：

```
if ($user->subscription('main')->onGracePeriod()) {
```

```
    //
}
```

2.7 恢复订阅

如果用户已经取消订阅但想要恢复该订阅，可以使用 `resume` 方法，前提是该用户必须在宽限期内：

```
$user->subscription('main')->resume();
```

如果该用户取消了一个订阅然后在订阅失效之前恢复了这个订阅，则不会立即支付该账单，取而代之的，他们的订阅只是被重新激活，并回到正常的支付周期。

3、处理 Stripe Webhook

3.1 订阅失败处理

如果客户的信用卡失效怎么办？不用担心—— Cashier 自带了 Webhook 控制器，该控制器可以很方便地为你取消客户订阅。只需要定义如下控制器路由：

```
Route::post('stripe/webhook', 'Laravel\Cashier\WebhookControlle
r@handleWebhook');
```

就是这样！失败的支付将会被该控制器捕获和处理。当 Stripe 判断订阅失败（正常情况下尝试支付失败三次后）时该控制器将会取消客户的订阅。不要忘了：你需要在 Stripe 控制面板设置中配置相应的 webhook URI，否则不能正常工作。

由于 Stripe webhooks 需要通过 Laravel 的 CSRF 验证，所以我们将该 URI 置于 `VerifyCsrfToken` 中间件排除列表中：

```
protected $except = [
    'stripe/*',
];
```

3.2 其它 Webhooks

如果你有额外想要处理的 Stripe webhook 事件，只需简单继承 Webhook 控制器， 你的方法名应该和 Cashier 期望的约定一致，尤其是方法应该以“handle”开头并以驼峰命名法命名。例如，如果你想要处理 `invoice.payment_succeeded`webhook，你应该添加 `handleInvoicePaymentSucceeded` 方法到控制器：

```
<?php

namespace App\Http\Controller;

use Laravel\Cashier\WebhookController as BaseController;

class WebhookController extends BaseController{
    /**
```

```
     * 处理 stripe webhook.
     *
     * @param  array  $payload
     * @return Response
     */
    public function handleInvoicePaymentSucceeded($payload)
    {
        // 处理该事件
    }
}
```

4、一次性付款

如果你想要使用订阅客户的信用卡一次性结清账单，可以使用账单模型实例上的 `charge` 方法，该方法接收付款金额（应用使用的货币的最小单位对应的金额数值）作为参数，例如，下面的例子使用信用卡支付 100 美分，或 1 美元：

```
$user->charge(100);
```

`charge` 方法接收一个数组作为第二个参数，允许你传递任何你想要传递的底层 Stripe 账单创建参数：

```
$user->charge(100, [
    'source' => $token,
    'receipt_email' => $user->email,]
);
```

如果支付失败 `charge` 方法将返回 `false`，这通常表明付款被拒绝：

```
if ( ! $user->charge(100)) {
    // The charge was denied...
}
```

如果支付成功，该方法将会返回一个完整的 Stripe 响应。

5、发票

你可以使用 `invoices` 方法轻松获取账单模型的发票数组：

```
$invoices = $user->invoices();
```

当列出客户发票时，你可以使用发票的辅助函数来显示相关的发票信息。例如，你可能想要在表格中列出每张发票，从而方便用户下载它们：

```
<table>
    @foreach ($invoices as $invoice)
        <tr>
```

```
            <td>{{ $invoice->dateString() }}</td>
            <td>{{ $invoice->dollars() }}</td>
            <td><a href="/user/invoice/{{ $invoice->id }}">Downl
oad</a></td>
        </tr>
    @endforeach
</table>
```

生成 PDF 发票

在生成 PDF 分票之前，需要安装 PHP 库 `dompdf`：

```
composer require dompdf/dompdf
```

在路由或控制器中，使用 `downloadInvoice` 方法生成发票的 PDF 下载，该方法将会自动生成相应的 HTTP 响应发送下载到浏览器：

```
Route::get('user/invoice/{invoice}', function ($invoiceId) {
    return Auth::user()->downloadInvoice($invoiceId, [
        'vendor'  => 'Your Company',
        'product' => 'Your Product',
    ]);
});
```

缓存

1、配置

Laravel 为不同的缓存系统提供了统一的 API。缓存配置位于 `config/cache.php`。在该文件中你可以指定在应用中默认使用哪个缓存驱动。Laravel 目前支持主流的缓存后端如 Memcached 和 Redis 等。
缓存配置文件还包含其他文档化的选项，确保仔细阅读这些选项。默认情况下，Laravel 被配置成使用文件缓存，这会将序列化数据和缓存对象存储到文件系统。对大型应用，建议使用内存缓存如 Memcached 或 APC，你甚至可以为同一驱动配置多个缓存配置。

1.1 缓存预备知识

数据库

使用 `database` 缓存驱动时，你需要设置一张表包含缓存缓存项。下面是该表的 `Schema` 声明：

```
Schema::create('cache', function($table) {
    $table->string('key')->unique();
    $table->text('value');
```

```
    $table->integer('expiration');
});
```

Memcached

使用 Memcached 缓存要求安装了 Memcached PECL 包，即 PHP Memcached 扩展。Memcached::addServer 默认配置使用 TCP/IP 协议：

```
'memcached' => [
    [
        'host' => '127.0.0.1',
        'port' => 11211,
        'weight' => 100
    ],
],
```

你还可以设置 `host` 选项为 UNIX socket 路径，如果你这样做，`port` 选项应该置为 `0`：

```
'memcached' => [
    [
        'host' => '/var/run/memcached/memcached.sock',
        'port' => 0,
        'weight' => 100
    ],
],
```

Redis

使用 Laravel 的 Redis 缓存之前，你需要通过 Composer 安装 `predis/predis` 包（~1.0）。

想要了解更多关于 Redis 的配置，查看 Larave 的 Redis 文档。

2、缓存使用

2.1 获取缓存实例

`Illuminate\Contracts\Cache\Factory` 和 `Illuminate\Contracts\Cache\Repository` 契约提供了访问 Laravel 的缓存服务的方法。`Factory` 契约提供了所有访问应用定义的缓存驱动的方法。`Repository` 契约通常是应用中 `cache` 配置文件中指定的默认缓存驱动的一个实现。

然而，你还可以使用 `Cache` 门面，这也是我们在整个文档中使用的方式，`Cache` 门面提供了简单方便的方式对底层 Laravel 缓存契约实现进行访问。

例如，让我们在控制器中导入 `Cache` 门面：

```
<?php

namespace App\Http\Controllers;

use Cache;
```

```
use Illuminate\Routing\Controller;

class UserController extends Controller{
    /**
     * 显示应用所有用户列表
     *
     * @return Response
     */
    public function index()
    {
        $value = Cache::get('key');

        //
    }
}
```

访问多个缓存存储

使用 `Cache` 门面，你可以使用 `store` 方法访问不同的缓存存储器，传入 `store` 方法的键就是 cache 配置文件中 `stores` 配置数组里列出的相应的存储器：

```
$value = Cache::store('file')->get('foo');
Cache::store('redis')->put('bar', 'baz', 10);
```

2.2 从缓存中获取数据

`Cache` 门面的 `get` 方法用于从缓存中获取缓存项，如果缓存项不存在，返回 null。如果需要的话你可以传递第二个参数到 `get` 方法指定缓存项不存在时返回的自定义默认值：

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

你甚至可以传递一个闭包作为默认值，如果缓存项不存在的话闭包的结果将会被返回。传递闭包允许你可以从数据库或其它外部服务获取默认值：

```
$value = Cache::get('key', function() {
    return DB::table(...)->get();
});
```

检查缓存项是否存在

`has` 方法用于判断缓存项是否存在：

```
if (Cache::has('key')) {
    //
}
```

数值增加/减少

`increment` 和 `decrement` 方法可用于调整缓存中的整型数值。这两个方法都可以接收第二个参数来指明缓存项数值增加和减少的数目：

```
Cache::increment('key');
Cache::increment('key', $amount);

Cache::decrement('key');
Cache::decrement('key', $amount);
```

获取或更新

有时候你可能想要获取缓存项，但如果请求的缓存项不存在时给它存储一个默认值。例如，你可能想要从缓存中获取所有用户，或者如果它们不存在的话，从数据库获取它们并将其添加到缓存中，你可以通过使用 `Cache::remember` 方法实现：

```
$value = Cache::remember('users', $minutes, function() {
    return DB::table('users')->get();
});
```

如果缓存项不存在，传递给 `remember` 方法的闭包被执行并且将结果存放到缓存中。
你还可以联合 `remember` 和 `forever` 方法：

```
$value = Cache::rememberForever('users', function() {
    return DB::table('users')->get();
});
```

获取并删除

如果你需要从缓存中获取缓存项然后删除，你可以使用 `pull` 方法，和 `get` 方法一样，如果缓存项不存在的话返回 null：

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

2.3 存储缓存项到缓存

你可以使用 `Cache` 门面上的 `put` 方法在缓存中存储缓存项。当你在缓存中存储缓存项的时候，你需要指定数据被缓存的时间（分钟数）：

```
Cache::put('key', 'value', $minutes);
```

除了传递缓存项失效时间，你还可以传递一个代表缓存项有效时间的 PHP `Datetime` 实例：

```
$expiresAt = Carbon::now()->addMinutes(10);
Cache::put('key', 'value', $expiresAt);
```

`add` 方法只会在缓存项不存在的情况下添加缓存项到缓存，如果缓存项被添加到缓存返回 `true`，否则，返回 `false`：

```
Cache::add('key', 'value', $minutes);
```

forever 方法用于持久化存储缓存项到缓存，这些值必须通过 forget 方法手动从缓存中移除：

```
Cache::forever('key', 'value');
```

2.4 从缓存中移除数据

你可以使用 Cache 门面上的 forget 方法从缓存中移除缓存项：

```
Cache::forget('key');
```

还可以使用 flush 方法清除所有缓存：

```
Cache::flush();
```

清除缓存并不管什么缓存键前缀，而是从缓存系统中移除所有数据，所以在使用这个方法时如果其他应用与本应用有共享缓存时需要格外注意。

3、缓存标签

注意：缓存标签目前不支持 file 或 database 缓存驱动，此外，当使用多标签的缓存被设置为永久存储时，使用 memcached 驱动的缓存有着最佳性能表现，因为 Memcached 会自动清除陈旧记录。

3.1 存储被打上标签的缓存项

缓存标签允许你给相关缓存项打上同一个标签以便于后续清除这些缓存值，被打上标签的缓存可以通过传递一个被排序的标签数组来访问。例如，我们可以通过以下方式在添加缓存的时候设置标签：

```
Cache::tags(['people', 'artists'])->put('John', $john, $minutes);Cache::tags(['people', 'authors'])->put('Anne', $anne, $minutes);
```

你可以给多个缓存项打上相同标签，这是没有数目限制的。

3.2 访问被打上标签的缓存项

要获取被打上标签的缓存项，传递同样的有序标签数组到 tags 方法：

```
$john = Cache::tags(['people', 'artists'])->get('John');$anne = Cache::tags(['people', 'authors'])->get('Anne');
```

你可以同时清除被打上同一标签/标签列表的所有缓存项，例如，以下语句会移除被打上 people 或 authors 标签的所有缓存：

```
Cache::tags(['people', 'authors'])->flush();
```

这样，上面设置的 Anne 和 John 缓存项都会从缓存中移除。

相反，以下语句只移除被打上 `authors` 标签的语句，所以只有 `Anne` 会被移除而 `John` 不会：

```
Cache::tags('authors')->flush();
```

4、添加自定义缓存驱动

要使用自定义驱动扩展 Laravel 缓存，可以使用 `Cache` 门面提供的 `extend` 方法，该方法用于绑定定义驱动解析器到管理器，通常，这可以在服务提供者中完成。
例如，要注册一个新的命名为“mongo”的缓存驱动：

```
<?php

namespace App\Providers;

use Cache;
use App\Extensions\MongoStore;
use Illuminate\Support\ServiceProvider;

class CacheServiceProvider extends ServiceProvider{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Cache::extend('mongo', function($app) {
            return Cache::repository(new MongoStore);
        });
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

传递给 `extend` 方法的第一个参数是驱动名称。该值对应配置文件 `config/cache.php` 中的 `driver` 选项。第二个参数是返回 `Illuminate\Cache\Repository` 实例的闭包。该闭包中被传入一个 `$app` 实例，也就是服务容器的一个实例。

调用 `Cache::extend` 可以在默认 `App\Providers\AppServiceProvider` 中的 `boot` 方法中完成，或者你也可以创建自己的服务提供者来存放该扩展——只是不要忘了在配置文件 `config/app.php` 中注册该提供者。

要创建自定义的缓存驱动，首先需要实现 `Illuminate\Contracts\Cache\Store` 契约，所以，我们的 `MongoDB` 缓存实现看起来像这样子：

```
<?php

namespace App\Extensions;

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() {}
    public function getPrefix() {}
}
```

我们只需要使用 `MongoDB` 连接实现每一个方法，实现完成后，我们可以完成自定义驱动注册：

```
Cache::extend('mongo', function($app) {
    return Cache::repository(new MongoStore);
});
```

扩展完成后，只需要更新配置文件 `config/cache.php` 的 `driver` 选项为你的扩展名称。

如果你在担心将自定义缓存驱动代码放到哪，考虑将其放到 Packgist！或者，你可以在 `app` 目录下创建一个 `Extensions` 命名空间。然而，记住 Laravel 并没有一个严格的应用目录结构，你可以基于你的需要自由的组织目录结构。

5、缓存事件

要在每次缓存操作时执行代码，你可以监听缓存触发的事件，通常，你可以将这些缓存处理器代码放到 `EventServiceProvider` 中：

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Cache\Events\CacheHit' => [
```

```
        'App\Listeners\LogCacheHit',
    ],

    'Illuminate\Cache\Events\CacheMissed' => [
        'App\Listeners\LogCacheMissed',
    ],

    'Illuminate\Cache\Events\KeyForgotten' => [
        'App\Listeners\LogKeyForgotten',
    ],

    'Illuminate\Cache\Events\KeyWritten' => [
        'App\Listeners\LogKeyWritten',
    ],
];
```

集合

1、简介

`Illuminate\Support\Collection` 类为处理数组数据提供了平滑、方便的封装。例如，查看下面的代码，我们使用辅助函数 `collect` 创建一个新的集合实例，为每一个元素运行 `strtoupper` 函数，然后移除所有空元素：

```
$collection = collect(['taylor', 'abigail', null])->map(function ($name) {
                return strtoupper($name);
            })->reject(function ($name) {
                return empty($name);
            });
```

正如你所看到的，`Collection` 类允许你使用方法链对底层数组执行匹配和减少操作，通常，没个 `Collection` 方法都会返回一个新的 `Collection` 实例。

2、创建集合

正如上面所提到的，辅助函数 `collect` 为给定数组返回一个新的 `Illuminate\Support\Collection` 实例，所以，创建集合很简单：

```
$collection = collect([1, 2, 3]);
```

默认情况下，Eloquent 模型的集合总是返回 `Collection` 实例，此外，不管是在何处，只要方法都可以自由使用 `Collection` 类。

3、集合方法

本文档接下来的部分我们将会讨论 `Collection` 类上每一个有效的方法，所有这些方法都可以以方法链的方式平滑的操作底层数组。此外，几乎每个方法返回一个新的 `Collection` 实例，允许你在必要的时候保持原来的集合备份。

all()

`all` 方法简单返回集合表示的底层数组：

```
collect([1, 2, 3])->all();

 // [1, 2, 3]
```

avg()

avg 方法返回所有集合项的平均值：

```
collect([1, 2, 3, 4, 5])->avg();
// 3
```

如果集合包含嵌套的数组或对象，需要指定要使用的键以判定计算那些值的平均值：

```
$collection = collect([
    ['name' => 'JavaScript: The Good Parts', 'pages' => 176],
    ['name' => 'JavaScript: The Definitive Guide', 'pages' => 10
96],]);

$collection->avg('pages');
// 636
```

chunk()

`chunk` 方法将一个集合分割成多个小尺寸的小集合：

```
$collection = collect([1, 2, 3, 4, 5, 6, 7]);

$chunks = $collection->chunk(4);

$chunks->toArray();

// [[1, 2, 3, 4], [5, 6, 7]]
```

当处理栅栏系统如 Bootstrap 时该方法在视图中尤其有用，建设你有一个想要显示在栅栏中的 Eloquent 模型集合：

```
@foreach ($products->chunk(3) as $chunk)
    <div class="row">
        @foreach ($chunk as $product)
            <div class="col-xs-4">{{ $product->name }}</div>
        @endforeach
    </div>
@endforeach
```

collapse()

`collapse` 方法将一个多维数组集合收缩成一个一维数组：

```
$collection = collect([[1, 2, 3], [4, 5, 6], [7, 8, 9]]);
$collapsed = $collection->collapse();
$collapsed->all();
// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

contains()

`contains` 方法判断集合是否包含一个给定项：

```
$collection = collect(['name' => 'Desk', 'price' => 100]);


$collection->contains('Desk');
// true
$collection->contains('New York');
// false
```

你还可以传递一个键值对到 `contains` 方法，这将会判断给定键值对是否存在于集合中：

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
]);
```

```
$collection->contains('product', 'Bookcase');

// false
```

最后，你还可以传递一个回调到 `contains` 方法来执行自己的真实测试：

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->contains(function ($key, $value) {

    return $value > 5;

});

// false
```

count()

`count` 方法返回集合中所有项的数目：

```
$collection = collect([1, 2, 3, 4]);

$collection->count();

// 4
```

diff()

`diff` 方法将集合和另一个集合或原生 PHP 数组作比较：

```
$collection = collect([1, 2, 3, 4, 5]);

$diff = $collection->diff([2, 4, 6, 8]);

$diff->all();

// [1, 3, 5]
```

each()

`each` 方法迭代集合中的数据项并传递每个数据项到给定回调：

```
$collection = $collection->each(function ($item, $key) {

    //

});
```

回调返回 `false` 将会终止循环：

```
$collection = $collection->each(function ($item, $key) {
```

```
    if (/* some condition */) {

        return false;

    }

});
```

every()

`every` 方法创建一个包含数组第 `n-th` 个元素的新集合：

```
$collection = collect(['a', 'b', 'c', 'd', 'e', 'f']);

$collection->every(4);
// ['a', 'e']
```

还可以选择指定从第几个元素开始：

```
$collection->every(4, 1);
// ['b', 'f']
```

except()

`except` 方法返回集合中除了指定键的所有集合项：

```
$collection = collect(['product_id' => 1, 'name' => 'Desk', 'pr
ice' => 100, 'discount' => false]);

$filtered = $collection->except(['price', 'discount']);

$filtered->all();
// ['product_id' => 1, 'name' => 'Desk']
```

与 `except` 相对的是 only 方法。

filter()

`filter` 方法通过给定回调过滤集合，只有通过给定测试的数据项才会保留下来：

```
$collection = collect([1, 2, 3, 4]);


$filtered = $collection->filter(function ($item) {

    return $item > 2;

});
```

```
$filtered->all();

// [3, 4]
```

和 `filter` 相对的方法是 `reject`。

first()

`first` 方法返回通过测试集合的第一个元素：

```
collect([1, 2, 3, 4])->first(function ($key, $value) {
    return $value > 2;
});

// 3
```

你还可以调用不带参数的 `first` 方法来获取集合的第一个元素，如果集合是空的，返回 null：

```
collect([1, 2, 3, 4])->first();

// 1
```

flatten()

`flatten` 方法将多维度的集合变成一维的：

```
$collection = collect(['name' => 'taylor', 'languages' => ['php', 'javascript']]);

$flattened = $collection->flatten();

$flattened->all();

// ['taylor', 'php', 'javascript'];
```

flip()

`flip` 方法将集合的键值做交换：

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$flipped = $collection->flip();

$flipped->all();

// ['taylor' => 'name', 'laravel' => 'framework']
```

forget()

`forget` 方法通过键从集合中移除数据项：

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$collection->forget('name');

$collection->all();

// [framework' => 'laravel']
```

注意：不同于大多数的集合方法，`forget` 不返回新的修改过的集合；它只修改所调用的集合。

forPage()

`forPage` 方法返回新的包含给定页数数据项的集合：

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9])->forPage(2, 3);


$collection->all();

// [4, 5, 6]
```

该方法需要传入页数和每页显示数目参数。

get()

`get` 方法返回给定键的数据项，如果不存在，返回 null：

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('name');

// taylor
```

你可以选择传递默认值作为第二个参数：

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('foo', 'default-value');

// default-value
```

你甚至可以传递回调作为默认值，如果给定键不存在的话回调的结果将会返回：

```
$collection->get('email', function () {
```

```
return 'default-value';});

// default-value
```

groupBy()

`groupBy` 方法通过给定键分组集合数据项：

```
$collection = collect([
    ['account_id' => 'account-x10', 'product' => 'Chair'],
    ['account_id' => 'account-x10', 'product' => 'Bookcase'],
    ['account_id' => 'account-x11', 'product' => 'Desk'],
]);

$grouped = $collection->groupBy('account_id');

$grouped->toArray();

/*
[
    'account-x10' => [
        ['account_id' => 'account-x10', 'product' => 'Chair'],
        ['account_id' => 'account-x10', 'product' => 'Bookcase'],
    ],
    'account-x11' => [
        ['account_id' => 'account-x11', 'product' => 'Desk'],
    ],
]
*/
```

除了传递字符串 key，还可以传递一个回调，回调应该返回分组后的值：

```
$grouped = $collection->groupBy(function ($item, $key) {
    return substr($item['account_id'], -3);
```

```
});


$grouped->toArray();


/*
[
    'x10' => [
        ['account_id' => 'account-x10', 'product' => 'Chair'],
        ['account_id' => 'account-x10', 'product' => 'Bookcase'],
    ],
    'x11' => [
        ['account_id' => 'account-x11', 'product' => 'Desk'],
    ],
]
*/
```

has()

`has` 方法判断给定键是否在集合中存在：

```
$collection = collect(['account_id' => 1, 'product' => 'Desk
']);
$collection->has('email');
// false
```

implode()

`implode` 方法连接集合中的数据项。其参数取决于集合中数据项的类型。
如果集合包含数组或对象，应该传递你想要连接的属性键，以及你想要放在值之间的“粘合”字符串：

```
$collection = collect([
    ['account_id' => 1, 'product' => 'Desk'],
    ['account_id' => 2, 'product' => 'Chair'],
]);

$collection->implode('product', ', ');
// Desk, Chair
```

如果集合包含简单的字符串或数值，只需要传递“粘合”字符串作为唯一参数到该方法：

```
collect([1, 2, 3, 4, 5])->implode('-');
// '1-2-3-4-5'
```

intersect()

`intersect` 方法返回两个集合的交集：

```
$collection = collect(['Desk', 'Sofa', 'Chair']);
$intersect = $collection->intersect(['Desk', 'Chair', 'Bookcase
']);
$intersect->all();
// [0 => 'Desk', 2 => 'Chair']
```

正如你所看到的，结果集合只保持原来集合的键。

isEmpty()

如果集合为空的话 `isEmpty` 方法返回 `true`；否则返回 `false`：

```
collect([])->isEmpty();
// true
```

keyBy()

将指定键的值作为集合的键：

```
$collection = collect([
    ['product_id' => 'prod-100', 'name' => 'desk'],
    ['product_id' => 'prod-200', 'name' => 'chair'],
]);

$keyed = $collection->keyBy('product_id');

$keyed->all();

/*
    [
        'prod-100' => ['product_id' => 'prod-100', 'name' => 'De
sk'],
        'prod-200' => ['product_id' => 'prod-200', 'name' => 'Ch
air'],
    ]
*/
```

如果多个数据项有同一个键，只有最后一个会出现在新的集合中。

你可以传递自己的回调，将会返回经过处理的键的值作为新的键：

```
$keyed = $collection->keyBy(function ($item) {
    return strtoupper($item['product_id']);
});

$keyed->all();

/*
    [
        'PROD-100' => ['product_id' => 'prod-100', 'name' => 'De
sk'],
        'PROD-200' => ['product_id' => 'prod-200', 'name' => 'Ch
air'],
    ]
*/
```

keys()

`keys` 方法返回所有集合的键：

```
$collection = collect([
    'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk
'],
    'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair
'],
]);

$keys = $collection->keys();

$keys->all();
// ['prod-100', 'prod-200']
```

last()

`last` 方法返回通过测试的集合的最后一个元素：

```
collect([1, 2, 3, 4])->last(function ($key, $value) {
    return $value < 3;
});
// 2
```

还可以调用无参的 `last` 方法来获取集合的最后一个元素。如果集合为空。返回 null：

```
collect([1, 2, 3, 4])->last();
// 4
```

map()

`map` 方法遍历集合并传递每个值给给定回调。该回调可以修改数据项并返回，从而生成一个新的经过修改的集合：

```
$collection = collect([1, 2, 3, 4, 5]);

$multiplied = $collection->map(function ($item, $key) {
    return $item * 2;
});

$multiplied->all();
// [2, 4, 6, 8, 10]
```

注意：和大多数集合方法一样，`map` 返回新的集合实例；它并不修改所调用的实例。如果你想要改变原来的集合，使用 `transform` 方法。

max()

`max` 方法返回集合中最大值：

```
$max = collect([['foo' => 10], ['foo' => 20]])->max('foo');
// 20
$max = collect([1, 2, 3, 4, 5])->max();
// 5
```

merge()

`merge` 方法合并给定数组到集合。该数组中的任何字符串键匹配集合中的字符串键的将会重写集合中的值：

```
$collection = collect(['product_id' => 1, 'name' => 'Desk']);
$merged = $collection->merge(['price' => 100, 'discount' => fal
se]);
$merged->all();
// ['product_id' => 1, 'name' => 'Desk', 'price' => 100, 'disco
unt' => false]
```

如果给定数组的键是数字，数组的值将会附加到集合后面：

```
$collection = collect(['Desk', 'Chair']);
$merged = $collection->merge(['Bookcase', 'Door']);
$merged->all();
// ['Desk', 'Chair', 'Bookcase', 'Door']
```

min()

`min` 方法返回集合中的最小值：

```
$min = collect([['foo' => 10], ['foo' => 20]])->min('foo');
// 10
$min = collect([1, 2, 3, 4, 5])->min();
// 1
```

only()

`only` 方法返回集合中指定键的集合项：

```
$collection = collect(['product_id' => 1, 'name' => 'Desk', 'pr
ice' => 100, 'discount' => false]);

$filtered = $collection->only(['product_id', 'name']);

$filtered->all();
// ['product_id' => 1, 'name' => 'Desk']
```

与 `only` 方法相对的是 except 方法。

pluck()

`pluck` 方法为给定键获取所有集合值：

```
$collection = collect([
    ['product_id' => 'prod-100', 'name' => 'Desk'],
    ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$plucked = $collection->pluck('name');

$plucked->all();
// ['Desk', 'Chair']
```

还可以指定你想要结果集合如何设置键：

```
$plucked = $collection->pluck('name', 'product_id');
$plucked->all();
// ['prod-100' => 'Desk', 'prod-200' => 'Chair']
```

pop()

`pop` 方法移除并返回集合中最后面的数据项：

```
$collection = collect([1, 2, 3, 4, 5]);
```

```
$collection->pop();
// 5
$collection->all();
// [1, 2, 3, 4]
```

prepend()

`prepend` 方法添加数据项到集合开头：

```
$collection = collect([1, 2, 3, 4, 5]);
$collection->prepend(0);
$collection->all();
// [0, 1, 2, 3, 4, 5]
```

你还可以传递第二个参数到该方法用于设置前置项的键：

```
$collection = collect(['one' => 1, 'two', => 2]);

$collection->prepend(0, 'zero');

$collection->all();
// ['zero' => 0, 'one' => 1, 'two', => 2]
```

pull()

`pull` 方法通过键从集合中移除并返回数据项：

```
$collection = collect(['product_id' => 'prod-100', 'name' => 'D
esk']);
$collection->pull('name');
// 'Desk'
$collection->all();
// ['product_id' => 'prod-100']
```

push()

`push` 方法附加数据项到集合结尾：

```
$collection = collect([1, 2, 3, 4]);
$collection->push(5);
$collection->all();
// [1, 2, 3, 4, 5]
```

put()

`put` 方法在集合中设置给定键和值：

```
$collection = collect(['product_id' => 1, 'name' => 'Desk']);
$collection->put('price', 100);
$collection->all();
// ['product_id' => 1, 'name' => 'Desk', 'price' => 100]
```

random()

`random` 方法从集合中返回随机数据项：

```
$collection = collect([1, 2, 3, 4, 5]);
$collection->random();
// 4 - (retrieved randomly)
```

你可以传递一个整型数据到 `random` 函数，如果该整型数值大于 1，将会返回一个集合：

```
$random = $collection->random(3);
$random->all();
// [2, 4, 5] - (retrieved randomly)
```

reduce()

`reduce` 方法用于减少集合到单个值，传递每个迭代结果到随后的迭代：

```
$collection = collect([1, 2, 3]);
$total = $collection->reduce(function ($carry, $item) {
    return $carry + $item;
});
// 6
```

在第一次迭代时 `$carry` 的值是 null；然而，你可以通过传递第二个参数到 `reduce` 来指定其初始值：

```
$collection->reduce(function ($carry, $item) {
    return $carry + $item;
}, 4);
// 10
```

reject()

`reject` 方法使用给定回调过滤集合，该回调应该为所有它想要从结果集合中移除的数据项返回 `true`：

```
$collection = collect([1, 2, 3, 4]);

$filtered = $collection->reject(function ($item) {
    return $item > 2;
});
```

```
$filtered->all();
// [1, 2]
```

和 `reduce` 方法相对的方法是 filter 方法。

reverse()

`reverse` 方法将集合数据项的顺序颠倒：

```
$collection = collect([1, 2, 3, 4, 5]);
$reversed = $collection->reverse();
$reversed->all();
// [5, 4, 3, 2, 1]
```

search()

`search` 方法为给定值查询集合，如果找到的话返回对应的键，如果没找到，则返回 `false`：

```
$collection = collect([2, 4, 6, 8]);
$collection->search(4);
// 1
```

上面的搜索使用的是松散比较，要使用严格比较，传递 `true` 作为第二个参数到该方法：

```
$collection->search('4', true);
// false
```

此外，你还可以传递自己的回调来搜索通过测试的第一个数据项：

```
$collection->search(function ($item, $key) {
    return $item > 5;});
// 2
```

shift()

`shift` 方法从集合中移除并返回第一个数据项：

```
$collection = collect([1, 2, 3, 4, 5]);
$collection->shift();
// 1
$collection->all();
// [2, 3, 4, 5]
```

shuffle()

`shuffle` 方法随机打乱集合中的数据项：

```
$collection = collect([1, 2, 3, 4, 5]);
$shuffled = $collection->shuffle();
$shuffled->all();
// [3, 2, 5, 1, 4] // (generated randomly)
```

slice()

`slice` 方法从给定索开始返回集合的一个切片：

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
$slice = $collection->slice(4);
$slice->all();
// [5, 6, 7, 8, 9, 10]
```

如果你想要限制返回切片的尺寸，将尺寸值作为第二个参数传递到该方法：

```
$slice = $collection->slice(4, 2);
$slice->all();
// [5, 6]
```

返回的切片有新的、数字化索引的键，如果你想要保持原有的键，可以传递第三个参数 `true` 到该方法。

sort()

`sort` 方法对集合进行排序：

```
$collection = collect([5, 3, 1, 2, 4]);
$sorted = $collection->sort();
$sorted->values()->all();
// [1, 2, 3, 4, 5]
```

排序后的集合保持原来的数组键，在本例中我们使用 `values` 方法重置键为连续编号索引。
要为嵌套集合和对象排序，查看 sortBy 和 sortByDesc 方法。
如果你需要更加高级的排序，你可以使用自己的算法传递一个回调给 `sort` 方法。参考 PHP 官方文档关于 `usort` 的说明，`sort` 方法底层正是调用了该方法。

sortBy()

`sortBy` 方法通过给定键对集合进行排序：

```
$collection = collect([
    ['name' => 'Desk', 'price' => 200],
    ['name' => 'Chair', 'price' => 100],
    ['name' => 'Bookcase', 'price' => 150],
]);

$sorted = $collection->sortBy('price');
```

```
$sorted->values()->all();

/*
    [
        ['name' => 'Chair', 'price' => 100],
        ['name' => 'Bookcase', 'price' => 150],
        ['name' => 'Desk', 'price' => 200],
    ]
*/
```

排序后的集合保持原有数组索引，在本例中，使用 values 方法重置键为连续索引。
你还可以传递自己的回调来判断如何排序集合的值：

```
$collection = collect([
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown
']],
]);

$sorted = $collection->sortBy(function ($product, $key) {
    return count($product['colors']);
});

$sorted->values()->all();

/*
    [
        ['name' => 'Chair', 'colors' => ['Black']],
        ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
        ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Bro
wn']],
    ]
*/
```

sortByDesc()

该方法和 sortBy 用法相同，不同之处在于按照相反顺序进行排序。

splice()

`splice` 方法在从给定位置开始移除并返回数据项切片：

```
$collection = collect([1, 2, 3, 4, 5]);
$chunk = $collection->splice(2);
```

```
$chunk->all();
// [3, 4, 5]
$collection->all();
// [1, 2]
```

你可以传递参数来限制返回组块的大小：

```
$collection = collect([1, 2, 3, 4, 5]);
$chunk = $collection->splice(2, 1);
$chunk->all();
// [3]
$collection->all();
// [1, 2, 4, 5]
```

此外，你可以传递第三个参数来包含新的数据项来替代从集合中移除的数据项：

```
$collection = collect([1, 2, 3, 4, 5]);
$chunk = $collection->splice(2, 1, [10, 11]);
$chunk->all();
// [3]
$collection->all();
// [1, 2, 10, 11, 4, 5]
```

sum()

`sum` 方法返回集合中所有数据项的和：

```
collect([1, 2, 3, 4, 5])->sum();
// 15
```

如果集合包含嵌套数组或对象，应该传递一个键用于判断对哪些值进行求和运算：

```
$collection = collect([
    ['name' => 'JavaScript: The Good Parts', 'pages' => 176],
    ['name' => 'JavaScript: The Definitive Guide', 'pages' => 10
96],
]);

$collection->sum('pages');
// 1272
```

此外，你还可以传递自己的回调来判断对哪些值进行求和：

```
$collection = collect([
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
```

```
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown
']],
]);

$collection->sum(function ($product) {
    return count($product['colors']);
});
// 6
```

take()

`take` 方法使用指定数目的数据项返回一个新的集合：

```
$collection = collect([0, 1, 2, 3, 4, 5]);
$chunk = $collection->take(3);
$chunk->all();
// [0, 1, 2]
```

你还可以传递负数从集合末尾开始获取指定数目的数据项：

```
$collection = collect([0, 1, 2, 3, 4, 5]);
$chunk = $collection->take(-2);
$chunk->all();
// [4, 5]
```

toArray()

`toArray` 方法将集合转化为一个原生的 PHP 数组。如果集合的值是 Eloquent 模型，该模型也会被转化为数组：

```
$collection = collect(['name' => 'Desk', 'price' => 200]);
$collection->toArray();

/*
    [
        ['name' => 'Desk', 'price' => 200],
    ]
*/
```

注意：`toArray` 还将所有嵌套对象转化为数组。如果你想要获取底层数组，使用 all 方法。

toJson()

`toJson` 方法将集合转化为 JSON：

```
$collection = collect(['name' => 'Desk', 'price' => 200]);
```

```
$collection->toJson();
// '{"name":"Desk","price":200}'
```

transform()

`transform` 方法迭代集合并对集合中每个数据项调用给定回调。集合中的数据项将会被替代成从回调中返回的值：

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->transform(function ($item, $key) {
    return $item * 2;
});

$collection->all();
// [2, 4, 6, 8, 10]
```

注意：不同于大多数其它集合方法，`transform` 修改集合本身，如果你想要创建一个新的集合，使用 map 方法。

unique()

`unique` 方法返回集合中所有的唯一数据项：

```
$collection = collect([1, 1, 2, 2, 3, 4, 2]);
$unique = $collection->unique();
$unique->values()->all();
// [1, 2, 3, 4]
```

返回的集合保持原来的数组键，在本例中我们使用 values 方法重置这些键为连续的数字索引。
处理嵌套数组或对象时，可以指定用于判断唯一的键：

```
$collection = collect([
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone
'],
    ['name' => 'iPhone 5', 'brand' => 'Apple', 'type' => 'phone
'],
    ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'wat
ch'],
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'pho
ne'],
    ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'w
atch'],
]);

$unique = $collection->unique('brand');
```

```
$unique->values()->all();

/*
    [
        ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'ph
one'],
        ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' =>
'phone'],
    ]
*/
```

你还可以指定自己的回调用于判断数据项唯一性：

```
$unique = $collection->unique(function ($item) {
    return $item['brand'].$item['type'];
});

$unique->values()->all();

/*
    [
        ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'ph
one'],
        ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' =>
'watch'],
        ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' =>
'phone'],
        ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type'
=> 'watch'],
    ]
*/
```

values()

`values` 方法使用重置为连续整型数字的键返回新的集合：

```
$collection = collect([
    10 => ['product' => 'Desk', 'price' => 200],
    11 => ['product' => 'Desk', 'price' => 200]
]);

$values = $collection->values();

$values->all();
```

```
/*
    [
        0 => ['product' => 'Desk', 'price' => 200],
        1 => ['product' => 'Desk', 'price' => 200],
    ]
*/
```

where()

`where` 方法通过给定键值对过滤集合：

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->where('price', 100);

$filtered->all();

/*
[
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Door', 'price' => 100],
]
*/
```

检查数据项值时 `where` 方法使用严格条件约束。使用 whereLoose 方法过滤松散约束。

whereLoose()

该方法和 `where` 使用方法相同，不同之处在于 `whereLoose` 在比较值的时候使用松散约束。

zip()

`zip` 方法在于集合的值相应的索引处合并给定数组的值：

```
$collection = collect(['Chair', 'Desk']);
$zipped = $collection->zip([100, 200]);
$zipped->all();
// [['Chair', 100], ['Desk', 200]]
```

集成前端资源：Laravel Elixir

1、简介

Laravel Elixir 提供了一套干净、平滑的 API 用于为 Laravel 应用定义基本的 Gulp 任务。Elixir 支持一些通用的 CSS 和 JavaScript 预处理器，甚至测试工具。使用方法链，Elixir 允许你平滑的定义资源管道。例如：

```
elixir(function(mix) {
    mix.sass('app.scss')
       .coffee('app.coffee');
});
```

如果你曾经对如何使用 Gulp 和编译前端资源感到困惑，那么你会爱上 Laravel Elixir。然而，并不是强制要求在开发期间使用它。你可以自由选择使用任何前端资源管道工具，或者压根不使用。

2、安装 & 设置

2.1 安装 Node

在开始 Elixir 之前，必须首先确保 Node.js 在机器上已经安装：

```
node -v
```

默认情况下，Laravel Homestead 包含你需要的一切；然而，如果你不使用 Vagrant，你也可以通过访问 Node 的下载页面轻松的安装 Node。

2.2 Gulp

接下来，需要安装 Gulp 作为全局 NPM 包：

```
npm install --global gulp
```

2.3 Laravel Elixir

最后，在新安装的 Laravel 根目录下，你会发现有一个 `package.json` 文件。该文件和 `composer.json` 一样，只不过是用来定义 Node 依赖而非 PHP，你可以通过运行如下命令来安装需要的依赖：

```
npm install
```

如果你正在 Windows 系统上开发，需要在运行 `npm install` 命令时带上 `--no-bin-links`：

```
npm install --no-bin-links
```

3、运行 Elixir

Elixir 基于 Gulp，所以要运行 Elixir 命令你只需要在终端中运行 `gulp` 命令即可。添加 `--production` 标识到命令将会最小化 CSS 和 JavaScript 文件：

```
// Run all tasks...
gulp

// Run all tasks and minify all CSS and JavaScript...
gulp --production
```

监控前端资源改变

由于每次修改前端资源后都要运行 `gulp` 很不方便，可以使用 `gulp watch` 命令。该命令将会一直在终端运行并监控前端文件的改动。当改变发生时，新文件将会自动被编译：

```
gulp watch
```

4、处理 CSS

项目根目录下的 `gulpfile.js` 文件包含了所有的 Elixir 任务。Elixir 任务可以使用方法链的方式链接起来用于定义前端资源如何被编译。

4.1 Less

要将 Less 编译成 CSS，可以使用 `less` 方法。`less` 方法假定你的 Less 文件都放在 `resources/assets/less`。默认情况下，本例中该任务会将编译后的 CSS 放到 `public/css/app.css`：

```
elixir(function(mix) {
    mix.less('app.less');
});
```

你还可以将多个 Less 文件编译成单个 CSS 文件。同样，该文件会被放到 `public/css/app.css`：

```
elixir(function(mix) {
    mix.less([
        'app.less',
        'controllers.less'
    ]);
});
```

如果你想要自定义编译后文件的输出位置，可以传递第二个参数到 `less` 方法：

```
elixir(function(mix) {
    mix.less('app.less', 'public/stylesheets');
});

// Specifying a specific output filename...
```

```
elixir(function(mix) {
    mix.less('app.less', 'public/stylesheets/style.css');
});
```

4.2 Sass

`sass` 方法允许你将 Sass 编译成 CSS。假定你的 Sass 文件存放在 `resources/assets/sass`，你可以像这样使用该方法：

```
elixir(function(mix) {
    mix.sass('app.scss');
});
```

同样，和 `less` 方法一样，你可以将多个脚本编译成单个 CSS 文件，甚至自定义结果 CSS 的输出路径：

```
elixir(function(mix) {
    mix.sass([
        'app.scss',
        'controllers.scss'
    ], 'public/assets/css');
});
```

4.3 原生 CSS

如果你只想要将多个原生 CSS 样式文件合并到一个文件，可以使用 `styles` 方法。传递给该方法的路径相对于 `resources/assets/css` 目录，结果 CSS 被存放在 `public/css/all.css`：

```
elixir(function(mix) {
    mix.styles([
        'normalize.css',
        'main.css'
    ]);
});
```

当然，你还可以通过传递第二个参数到 `styles` 方法来输出结果文件到一个自定义路径：

```
elixir(function(mix) {
    mix.styles([
        'normalize.css',
        'main.css'
    ], 'public/assets/css');
});
```

4.4 源地图

默认源地图被启用，所以，对于每一个你编译过的文件都可以在同一目录下找到一个对应的 `*.css.map` 文件。这种匹配允许你在浏览器中调试时将编译过的样式选择器回溯到原来的 Sass 或 Less。
如果你不想为 CSS 生成源地图，可以使用一个简单配置选项关闭它们：

```
elixir.config.sourcemaps = false;

elixir(function(mix) {
    mix.sass('app.scss');
});
```

5、处理 JavaScript

Elixir 还提供了多个函数帮助你处理 JavaScript 文件，例如编译 ECMAScript 6，CoffeeScript，Browserify，最小化以及简单连接原生 JavaScript 文件。

5.1 CoffeeScript

`coffee` 方法用于将 CoffeeScript 编译成原生 JavaScript。该方法接收关联到 `resources/assets/coffee` 目录的 CoffeeScript 文件的一个字符串或数组并在 `public/js` 目录下生成单个 `app.js` 文件：

```
elixir(function(mix) {
    mix.coffee(['app.coffee', 'controllers.coffee']);
});
```

5.2 Browserify

Elixir 还提供了 `browserify` 方法，从而让你可以在浏览器中引入模块并使用 EcmaScript 6。
该任务假定你的脚本都存放在 `resources/assets/js` 而且将结果文件存放到 `public/js/bundle.js`：

```
elixir(function(mix) {
    mix.browserify('main.js');
});
```

除了处理 Partialify 和 Babelify，还可以安装并添加更多：

```
npm install vueify --save-dev
elixir.config.js.browserify.transformers.push({
    name: 'vueify',
    options: {}
});
```

```
elixir(function(mix) {
    mix.browserify('main.js');
});
```

5.3 Babel

`babel` 方法可用于将 EcmaScript 6 和 7 编译成原生 JavaScript。该方法接收相对于 `resources/assets/js` 目录的文件数组，并在 `public/js` 目录下生成单个 `all.js`：

```
elixir(function(mix) {
    mix.babel([
        'order.js',
        'product.js'
    ]);
});
```

要选择不同的输出路径，只需将目标路径作为第二个参数传递给该方法。处了 Babel 编译之外，`babel` 和 `mix.scripts()`的使用方法和功能差不多。

5.4 脚本

如果你有多个 JavaScript 文件想要编译成单个文件，可以使用 `scripts` 方法。`scripts` 方法假定所有路径相对于 `resources/assets/js` 目录，而且所有结果 JavaScript 默认存放在 `public/js/all.js`：

```
elixir(function(mix) {
    mix.scripts([
        'jquery.js',
        'app.js'
    ]);
});
```

如果你需要将多个脚本集合合并到不同的文件，需要多次调用 `scripts` 方法。该方法的第二个参数决定每个合并的结果文件名：

```
elixir(function(mix) {
    mix.scripts(['app.js', 'controllers.js'], 'public/js/app.js
')
       .scripts(['forum.js', 'threads.js'], 'public/js/forum.js
');
});
```

如果你需要将多个脚本合并到给定目录，可以使用 `scriptsIn` 方法。结果 JavaScript 会被存放到 `public/js/all.js`：

```
elixir(function(mix) {
    mix.scriptsIn('public/js/some/directory');
});
```

6、拷贝文件/目录

你可以使用 `copy` 方法拷贝文件/目录到新路径，所有操作都相对于项目根目录：

```
elixir(function(mix) {
    mix.copy('vendor/foo/bar.css', 'public/css/bar.css');
});

elixir(function(mix) {
    mix.copy('vendor/package/views', 'resources/views');
});
```

7、版本号/缓存刷新

很多开发者会给编译的前端资源添加时间戳或者唯一令牌后缀以强制浏览器加载最新版本而不是代码的缓存副本。Elixir 可以使用 `version` 方法为你处理这种情况。`version` 方法接收相对于 `public` 目录的文件名，附加唯一 hash 到文件名，从而实现缓存刷新。例如，生成的文件名看上去是这样——`all-16d570a7.css`：

```
elixir(function(mix) {
    mix.version('css/all.css');
});
```

生成版本文件后，可以在视图中使用 Elixir 全局的 PHP 帮助函数 `elixir` 方法来加载相应的带 hash 值的前端资源，`elixir` 函数会自动判断 hash 文件名：

```
<link rel="stylesheet" href="{{ elixir('css/all.css') }}">
```

给多个文件加上版本号

你可以传递一个数组到 `version` 方法来为多个文件添加版本号：

```
elixir(function(mix) {
    mix.version(['css/all.css', 'js/app.js']);
});
```

一旦文件被加上版本号，就可以使用帮助函数 `elixir` 来生成指向该 hash 文件的链接。记住，你只需要传递没有 hash 值的文件名到 `elixir` 方法。该帮助函数使用未加 hash 值的文件名来判断文件当前的 hash 版本：

```
<link rel="stylesheet" href="{{ elixir('css/all.css') }}">

<script src="{{ elixir('js/app.js') }}"></script>
```

8、BrowserSync

BrowserSync 会在你修改前端资源后自动刷新浏览器，运行 `gulp watch` 命令时你可以使用 `browserSync` 方法告知 Elixir 启动一个 BrowserSync 服务器：

```
elixir(function(mix) {
    mix.browserSync();
});
```

运行 `gulp watch` 后，使用 `http://homestead.app:3000` 访问应用来开启浏览器同步。如果你在本地开发中使用 `homestead.app` 之外的其它域名，可以传递域名参数到 `browserSync` 方法：

```
elixir(function(mix) {
    mix.browserSync({
        proxy: 'project.app'
    });
});
```

9、调用存在的 Gulp 任务

如果你需要从 Elixir 调用已存在的 Gulp 任务，可以使用 `task` 方法。例如，假定你有一个调用时只是简单说几句话的 Gulp 任务：

```
gulp.task('speak', function() {
    var message = 'Tea...Earl Grey...Hot';
    gulp.src('').pipe(shell('say ' + message));
});
```

如果你想要从 Elixir 中调用该任务，使用 `mix.task` 方法并传递任务名作为该方法的唯一参数：

```
elixir(function(mix) {
    mix.task('speak');
});
```

自定义监控者

如果你需要注册一个监控器在每一次文件修改时都运行自定义任务，传递一个正则表达式作为 `task` 方法的第二个参数：

```
elixir(function(mix) {
    mix.task('speak', 'app/**/*.php');
});
```

10、编写 Elixir 扩展

如果你需要比 Elixir 的 `task` 方法所提供的更加灵活的功能，可以创建自定义的 Elixir 扩展。Elixir 扩展允许你传递参数到自定义任务，例如，你可以像这样编写一个扩展：

```
// File: elixir-extensions.js
var gulp = require('gulp');
var shell = require('gulp-shell');
var Elixir = require('laravel-elixir');

var Task = Elixir.Task;

Elixir.extend('speak', function(message) {
    new Task('speak', function() {
        return gulp.src('').pipe(shell('say ' + message));
    });
});

// mix.speak('Hello World');
```

就是这样简单！注意你的特定 Gulp 逻辑应该放到闭包函数里作为第二个参数传递给 `Task` 构造器。你可以将其放在 `Gulpfile` 顶端，或者将其解析到自定义的任务文件。例如，如果你将扩展放在 `elixir-extensions.js`，可以在主 `Gulpfile` 中像这样引入该文件：

```
// File: Gulpfile.js
var elixir = require('laravel-elixir');

require('./elixir-extensions')

elixir(function(mix) {
    mix.speak('Tea, Earl Grey, Hot');
});
```

自定义监控器

如果你想要自定义任务在运行 `gulp watch` 的时候被触发，可以注册一个监控器：

```
new Task('speak', function() {
    return gulp.src('').pipe(shell('say ' + message));
}).watch('./app/**');
```

加密

1、配置

在使用 Laravel 的加密器之前，应该在配置文件 `config/app.php` 中设置 `key` 选项为 32 位随机字符串。如果这个值没有被设置，所有 Laravel 加密过的值都是不安全的。

2、基本使用

2.1 加密

你可以使用 Crypt 门面对数据进行加密，所有加密值都使用 OpenSSL 和 `AES-256-CBC` 密码进行加密。此外，所有加密值都通过一个消息认证码（MAC）来检测对加密字符串的任何修改。
例如，我们可以使用 `encrypt` 方法加密 `secret` 属性并将其存储到 Eloquent 模型：

```
<?php

namespace App\Http\Controllers;

use Crypt;
use App\User;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserController extends Controller{
    /**
     * Store a secret message for the user.
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function storeSecret(Request $request, $id)
    {
        $user = User::findOrFail($id);

        $user->fill([
            'secret' => Crypt::encrypt($request->secret)
        ])->save();
    }
```

```
}
```

2.2 解密

当然，你可以使用 `Crypt` 门面上的 `decrypt` 方法进行解密。如果该值不能被解密，例如 MAC 无效，将会抛出一个 `Illuminate\Contracts\Encryption\DecryptException` 异常：

```
use Illuminate\Contracts\Encryption\DecryptException;

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

错误&日志

1、简介

Laravel 默认已经为我们配置好了错误和异常处理，此外，Laravel 还集成了 Monolog 日志库以便提供多种功能强大的日志处理器。

2、配置

错误详情显示

配置文件 `config/app.php` 中的 `debug` 配置选项控制浏览器显示的错误详情数量。默认情况下，该配置选项被设置在 `.env` 文件中的环境变量 `APP_DEBUG`。
对本地开发而言，你应该设置环境变量 `APP_DEBUG` 值为 `true`。在生产环境，该值应该被设置为 `false`。

日志模式

Laravel 支持日志方法 `single`, `daily`, `syslog` 和 `errorlog`。例如，如果你想要日志文件按日生成而不是生成单个文件，应该在配置文件 `config/app.php` 中设置 `log` 值如下：

```
'log' => 'daily'
```

自定义 Monolog 配置

如果你想要在应用中完全控制 Monolog 的配置，可以使用应用的 `configureMonologUsing` 方法。你应该在 `bootstrap/app.php` 文件返回`$app` 变量之前调用该方法：

```
$app->configureMonologUsing(function($monolog) {
    $monolog->pushHandler(...);
});
```

```
return $app;
```

3、异常处理器

所有异常都由类 `App\Exceptions\Handler` 处理，该类包含两个方法：`report` 和 `render`。下面我们详细阐述这两个方法。

3.1 report 方法

`report` 方法用于记录异常并将其发送给外部服务如 Bugsnag。默认情况下，`report` 方法只是将异常传递给异常被记录的基类，你可以随心所欲的记录异常。
例如，如果你需要以不同方式报告不同类型的异常，可使用 PHP 的 `instanceof` 比较操作符：

```
/**
 * 报告或记录异常
 *
 * This is a great spot to send exceptions to Sentry, Bugsnag, etc.
 *
 * @param  \Exception  $e
 * @return void
 */
public function report(Exception $e){
    if ($e instanceof CustomException) {
        //
    }

    return parent::report($e);
}
```

通过类型忽略异常

异常处理器的`$dontReport` 属性包含一个不会被记录的异常类型数组，默认情况下，`404` 错误异常不会被写到日志文件，如果需要的话你可以添加其他异常类型到这个数组。

3.2 render 方法

`render` 方法负责将给定异常转化为发送给浏览器的 HTTP 响应，默认情况下，异常被传递给为你生成响应的基类。然而，你可以随心所欲地检查异常类型或者返回自定义响应：

```
/**
 * 将异常渲染到 HTTP 响应中
 *
 * @param  \Illuminate\Http\Request  $request
 * @param  \Exception  $e
```

```
 * @return \Illuminate\Http\Response
 */
public function render($request, Exception $e){
    if ($e instanceof CustomException) {
        return response()->view('errors.custom', [], 500);
    }

    return parent::render($request, $e);
}
```

4、HTTP 异常

有些异常描述来自服务器的 HTTP 错误码，例如，这可能是一个“页面未找到”错误（`404`），“认证失败错误”（`401`）亦或是程序出错造成的 `500` 错误，为了在应用中生成这样的响应，使用如下方法：

```
abort(404);
```

`abort` 方法会立即引发一个会被异常处理器渲染的异常，此外，你还可以像这样提供响应描述：

```
abort(403, 'Unauthorized action.');
```

该方法可在请求生命周期的任何时间点使用。

自定义 HTTP 错误页面

Laravel 使得返回多种 HTTP 状态码的错误页面变得简单，例如，如果你想要自定义 `404` 错误页面，创建一个 `resources/views/errors/404.blade.php` 文件，给文件将会渲染程序生成的所有 `404` 错误。
改目录下的视图命名应该和相应的 HTTP 状态码相匹配。

5、日志

Laravel 日志工具基于强大的 Monolog 库，默认情况下，Laravel 被配置为在 `storage/logs` 目录下每日为应用生成日志文件，你可以使用 `Log` 门面编写日志信息到日志中：

```
<?php

namespace App\Http\Controllers;

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

```
class UserController extends Controller{
    /**
     * 显示指定用户的属性
     *
     * @param  int  $id
     * @return Response
     */
    public function showProfile($id)
    {
        Log::info('Showing user profile for user: '.$id);
        return view('user.profile', ['user' => User::findOrFail
($id)]);
    }
}
```

该日志记录器提供了 RFC 5424 中定义的八种日志级别：
emergency, **alert**, **critical**, **error**,**warning**, **notice**, **info** 和 **debug**。

```
Log::emergency($error);

Log::alert($error);

Log::critical($error);

Log::error($error);

Log::warning($error);

Log::notice($error);

Log::info($error);

Log::debug($error);
```

上下文信息

上下文数据数组也会被传递给日志方法。上下文数据将会和日志消息一起被格式化和显示：

```
Log::info('User failed to login.', ['id' => $user->id]);
```

访问底层 Monolog 实例

Monolog 有多个可用于日志的处理器，如果需要的话，你可以访问底层 Monolog 实例：

```
$monolog = Log::getMonolog();
```

事件

1、简介

Laravel 事件提供了简单的观察者模式实现，允许你订阅和监听应用中的事件。事件类通常存放在 `app/Events` 目录，监听器存放在 `app/Listeners`。

2、注册事件/监听器

Laravel 自带的 `EventServiceProvider` 为事件注册提供了方便之所。其中的 `listen` 属性包含了事件（键）和对应监听器（值）数组。如果应用需要，你可以添加多个事件到该数组。例如，让我们添加 `PodcastWasPurchased` 事件：

```
/**
 * 事件监听器映射
 *
 * @var array
 */
protected $listen = [
    'App\Events\PodcastWasPurchased' => [
        'App\Listeners\EmailPurchaseConfirmation',
    ],
];
```

2.1 生成事件/监听器类

当然，手动为每个事件和监听器创建文件是很笨重的，取而代之地，我们可见简单添加监听器和事件到 `EventServiceProvider` 然后使用 `event:generate` 命令。该命令将会生成罗列在 `EventServiceProvider` 中的所有事件和监听器。当然，已存在的事件和监听器不会被创建：

```
php artisan event:generate
```

2.2 手动注册事件

一般我们需要将事件注册到 `EventServiceProvider` 的 `$listen` 数组，此外，我们还可以使用 `Event` 门面或者 `Illuminate\Contracts\Events\Dispatcher` 契约的具体实现类作为事件分发器手动注册事件：

```

/**
 * Register any other events for your application.
 *
 * @param  \Illuminate\Contracts\Events\Dispatcher  $events
```

```
 * @return void
 */
public function boot(DispatcherContract $events)
{
    parent::boot($events);

    $events->listen('event.name', function ($foo, $bar) {
        //
    });
}
```

使用通配符作为事件监听器

你还可以使用通配符*来注册监听器，从而允许你通过同一个监听器捕获多个事件。通配符监听器接收整个事件数据数组作为参数：

```
$events->listen('event.*', function (array $data) {
    //
});
```

3、定义事件

事件类是一个处理与事件相关的简单数据容器，例如，假设我们生成的 `PodcastWasPurchased` 事件接收一个 Eloquent ORM 对象：

```
<?php

namespace App\Events;

use App\Podcast;
use App\Events\Event;
use Illuminate\Queue\SerializesModels;

class PodcastWasPurchased extends Event{
    use SerializesModels;

    public $podcast;

    /**
     * 创建新的事件实例
     *
     * @param  Podcast  $podcast
     * @return void
     */
```

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

正如你所看到的，该事件类不包含任何特定逻辑，只是一个存放被购买的 `Podcast` 对象的容器，如果事件对象被序列化的话，事件使用的 `SerializesModels` trait 将会使用 PHP 的 `serialize` 函数序列化所有 Eloquent 模型。

4、定义监听器

接下来，让我们看看我们的示例事件的监听器，事件监听器在 `handle` 方法中接收事件实例，`event:generate` 命令将会自动在 `handle` 方法中导入合适的事件类和类型提示事件。在 `handle` 方法内，你可以执行任何需要的逻辑以响应事件。

```
<?php

namespace App\Listeners;

use App\Events\PodcastWasPurchased;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class EmailPurchaseConfirmation{
    /**
     * 创建事件监听器
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * 处理事件
     *
     * @param  PodcastWasPurchased  $event
     * @return void
     */
    public function handle(PodcastWasPurchased $event)
    {
        // Access the podcast using $event->podcast...
    }
```

```
}
```

你的事件监听器还可以在构造器中类型提示任何需要的依赖，所有事件监听器通过服务容器解析，所以依赖会自动注入：

```
use Illuminate\Contracts\Mail\Mailer;

public function __construct(Mailer $mailer){
    $this->mailer = $mailer;
}
```

停止事件继续往下传播

有时候，你希望停止事件被传播到其它监听器，你可以通过从监听器的 `handle` 方法中返回 `false` 来实现。

事件监听器队列

需要将事件监听器放到队列？没有比这更简单的了，只需要让监听器类实现 `ShouldQueue` 接口即可，通过 Artisan 命令 `event:generate` 生成的监听器类已经将接口导入当前命名空间，所有你可以立即拿来使用：

```
<?php

namespace App\Listeners;

use App\Events\PodcastWasPurchased;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class EmailPurchaseConfirmation implements ShouldQueue{
    //
}
```

就是这么简单，当监听器被事件调用，将会使用 Laravel 的队列系统通过队列分发器自动队列化。如果通过队列执行监听器的时候没有抛出任何异常，队列任务在执行完成后被自动删除。

手动访问队列

如果你需要手动访问底层队列任务的 `delete` 和 `release` 方法，在生成的监听器中默认导入的 `Illuminate\Queue\InteractsWithQueue` trait 提供了访问这两个方法的权限：

```
<?php

namespace App\Listeners;

use App\Events\PodcastWasPurchased;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;
```

```
class EmailPurchaseConfirmation implements ShouldQueue{
    use InteractsWithQueue;

    public function handle(PodcastWasPurchased $event)
    {
        if (true) {
            $this->release(30);
        }
    }
}
```

5、触发事件

要触发一个事件，可以使用 Event 门面，传递一个事件实例到 `fire` 方法，`fire` 方法会分发事件到所有监听器：

```
<?php

namespace App\Http\Controllers;

use Event;
use App\Podcast;
use App\Events\PodcastWasPurchased;
use App\Http\Controllers\Controller;

class UserController extends Controller{
    /**
     * 显示指定用户属性
     *
     * @param  int  $userId
     * @param  int  $podcastId
     * @return Response
     */
    public function purchasePodcast($userId, $podcastId)
    {
        $podcast = Podcast::findOrFail($podcastId);

        // Purchase podcast logic...

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

此外，你还可以使用全局的辅助函数 `event` 来触发事件：

```
event(new PodcastWasPurchased($podcast));
```

6、广播事件

在很多现代 Web 应用中，Web 套接字被用于实现实时更新的用户接口。当一些数据在服务器上被更新，通常一条消息通过 websocket 连接被发送给客户端处理。

为帮助你构建这样的应用，Laravel 让通过 websocket 连接广播事件变得简单。广播 Laravel 事件允许你在服务端和客户端 JavaScript 框架之间共享同一事件名。

6.1 配置

所有的事件广播配置选项都存放在 `config/broadcasting.php` 配置文件中。Laravel 支持多种广播驱动：Pusher、Redis 以及一个服务于本地开发和调试的日志驱动。每一个驱动都有一个配置示例。

广播预备知识

事件广播需要以下两个依赖：

- Pusher: `pusher/pusher-php-server ~2.0`
- Redis: `predis/predis ~1.0`

队列预备知识

在开始介绍广播事件之前，还需要配置并运行一个队列监听器。所有事件广播都通过队列任务来完成以便应用的响应时间不受影响。

6.2 将事件标记为广播

要告诉 Laravel 给定事件应该被广播，需要在事件类上实现 `Illuminate\Contracts\Broadcasting\ShouldBroadcast` 接口。`ShouldBroadcast` 接口要求你实现一个方法：`broadcastOn`。该方法应该返回事件广播“频道”名称数组：

```
<?php

namespace App\Events;

use App\User;
use App\Events\Event;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class ServerCreated extends Event implements ShouldBroadcast{
    use SerializesModels;

    public $user;

    /**
```

```
     * 创建新的事件实例
     *
     * @return void
     */
    public function __construct(User $user)
    {
        $this->user = $user;
    }

    /**
     * 获取事件广播频道
     *
     * @return array
     */
    public function broadcastOn()
    {
        return ['user.'.$this->user->id];
    }
}
```

然后，你只需要和正常一样触发该事件，事件被触发后，一个队列任务将通过指定广播驱动自动广播该事件。

6.3 广播数据

如果某个事件被广播，其所有的 `public` 属性都会按照事件负载自动序列化和广播，从而允许你从 JavaScript 中访问所有 `public` 数据，因此，举个例子，如果你的事件有一个单独的包含 Eloquent 模型的 `$user` 属性，广播负载定义如下：

```
{
    "user": {
        "id": 1,
        "name": "Jonathan Banks"
        ...
    }
}
```

然而，如果你希望对广播负载有更加细粒度的控制，可以添加 `broadcastWith` 方法到事件，该方法应该返回你想要通过事件广播的数组数据：

```
/**
 * 获取广播数据
 *
 * @return array
 */
public function broadcastWith(){
    return ['user' => $this->user->id];
```

```
}
```

6.4 自定义事件广播

自定义事件名

默认情况下，广播事件名就是事件类名，因此，如果事件的类名是 `App\Events\ServerCreated`，对应的广播事件名就是 `App\Events\ServerCreated`，你可以通过事件类上的 `broadcastAs` 方法自定义广播事件名：

```
/**
 * 获取广播事件名称
 *
 * @return string
 */
public function broadcastAs()
{
    return 'app.server-created';
}
```

自定义队列

默认情况下，每个被广播的事件都位于配置文件 `queue.php` 中定义的默认队列连接中的默认队列中，你可以通过事件类的 `onQueue` 方法自定义广播事件的队列名称。该方法会返回你期望使用的队列名：

```
/**
 * 设置事件所在队列的名称
 *
 * @return string
 */
public function onQueue()
{
    return 'your-queue-name';
}
```

6.5 消费事件广播

Pusher

你可以通过 Pusher 的 JavaScript SDK 方便地使用 Pusher 驱动消费事件广播。例如，让我们从之前的例子中消费 `App\Events\ServerCreated` 事件：

```
this.pusher = new Pusher('pusher-key');

this.pusherChannel = this.pusher.subscribe('user.' + USER_ID);
```

```
this.pusherChannel.bind('App\\Events\\ServerCreated', function
(message) {
    console.log(message.user);
});
```

Redis

如果你在使用 Redis 广播，你将需要编写自己的 Redis pub/sub 消费者来接收消息并使用自己选择的 websocket 技术将其进行广播。例如，你可以选择使用 Node 编写的流行的 Socket.io 库。

使用 Node 库 `socket.io` 和 `ioredis`，你可以快速编写事件广播发布所有广播事件：

```
var app = require('http').createServer(handler);
var io = require('socket.io')(app);

var Redis = require('ioredis');
var redis = new Redis();

app.listen(6001, function() {
    console.log('Server is running!');});

function handler(req, res) {
    res.writeHead(200);
    res.end('');}

io.on('connection', function(socket) {
    //
});

redis.psubscribe('*', function(err, count) {
    //
});

redis.on('pmessage', function(subscribed, channel, message) {
    message = JSON.parse(message);
    io.emit(channel + ':' + message.event, message.data);
});
```

7、事件订阅者

事件订阅者是指那些在类本身中订阅到多个事件的类，从而允许你在单个类中定义一些事件处理器。订阅者应该定义一个 `subscribe` 方法，该方法中传入一个事件分发器实例：

```
<?php
```

```
namespace App\Listeners;

class UserEventListener{
    /**
     * 处理用户登录事件
     */
    public function onUserLogin($event) {}

    /**
     * 处理用户退出事件
     */
    public function onUserLogout($event) {}

    /**
     * 为订阅者注册监听器
     *
     * @param  Illuminate\Events\Dispatcher  $events
     * @return array
     */
    public function subscribe($events)
    {
        $events->listen(
            'App\Events\UserLoggedIn',
            'App\Listeners\UserEventListener@onUserLogin'
        );

        $events->listen(
            'App\Events\UserLoggedOut',
            'App\Listeners\UserEventListener@onUserLogout'
        );
    }

}
```

注册一个事件订阅者

订阅者被定义后，可以通过事件分发器进行注册，你可以使用 `EventServiceProvider` 上的 `$subcribe` 属性来注册订阅者。例如，让我们添加 `UserEventListener`：

```
<?php

namespace App\Providers;

use Illuminate\Contracts\Events\Dispatcher as DispatcherContrac
t;
```

```
use Illuminate\Foundation\Support\Providers\EventServiceProvide
r as ServiceProvider;

class EventServiceProvider extends ServiceProvider{
    /**
     * 事件监听器映射数组
     *
     * @var array
     */
    protected $listen = [
        //
    ];

    /**
     * 要注册的订阅者
     *
     * @var array
     */
    protected $subscribe = [
        'App\Listeners\UserEventListener',
    ];
}
```

文件系统/云存储

1、简介

Laravel 基于 Frank de Jonge 开发的 PHP 包 Flysystem 提供了强大的文件系统抽象。Laravel 文件系统集成提供了使用驱动处理本地文件系统的简单使用，这些驱动包括 Amazon S3，以及 Rackspace 云存储。此外在这些存储选项间切换非常简单，因为对每个系统而言，API 是一样的。

2、配置

文件系统配置文件位于 `config/filesystems.php`。在该文件中可以配置所有”硬盘“，每个硬盘描述了特定的存储驱动和存储位置。为每种支持的驱动的示例配置包含在该配置文件中，所以，简单编辑该配置来反映你的存储参数和认证信息。
当然，你想配置磁盘多少就配置多少，多个磁盘也可以共用同一个驱动。

本地驱动

使用 `local` 驱动的时候，注意所有文件操作相对于定义在配置文件中的 `root` 目录，默认情况下，该值设置为 `storage/app` 目录，因此，下面的方法将会存储文件到 `storage/app/file.txt`：

```
Storage::disk('local')->put('file.txt', 'Contents');
```

其它驱动预备知识

在使用 Amazon S3 或 Rackspace 驱动之前，需要通过 Composer 安装相应的包：

- Amazon S3: `league/flysystem-aws-s3-v3 ~1.0`
- Rackspace: `league/flysystem-rackspace ~1.0`

3、基本使用

3.1 获取硬盘实例

`Storage` 门面用于和你配置的所有磁盘进行交互，例如，你可以使用该门面上的 put 方法来存储头像到默认磁盘，如果你调用 `Storage` 门面上的方法却先调用 `disk` 方法，该方法调用自动传递到默认磁盘：

```
<?php

namespace App\Http\Controllers;

use Storage;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserController extends Controller{
    /**
     * 更新指定用户头像
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function updateAvatar(Request $request, $id)
    {
        $user = User::findOrFail($id);

        Storage::put(
            'avatars/'.$user->id,
            file_get_contents($request->file('avatar')->getRealP
ath())
```

```
        );
    }
}
```

使用多个磁盘时，可以使用 `Storage` 门面上的 `disk` 方法访问特定磁盘。当然，可以继续使用方法链执行该磁盘上的方法：

```
$disk = Storage::disk('s3');
$contents = Storage::disk('local')->get('file.jpg')
```

3.2 获取文件

`get` 方法用于获取给定文件的内容，该方法将会返回该文件的原生字符串内容：

```
$contents = Storage::get('file.jpg');
```

`exists` 方法用于判断给定文件是否存在于磁盘上：

```
$exists = Storage::disk('s3')->exists('file.jpg');
```

文件元信息

`size` 方法以字节方式返回文件大小：

```
$size = Storage::size('file1.jpg');
```

`lastModified` 方法以 UNIX 时间戳格式返回文件最后一次修改时间：

```
$time = Storage::lastModified('file1.jpg');
```

3.3 存储文件

`put` 方法用于存储文件到磁盘。可以传递一个 PHP 资源到 `put` 方法，该方法将会使用 Flysystem 底层的流支持。在处理大文件的时候推荐使用文件流：

```
Storage::put('file.jpg', $contents);
Storage::put('file.jpg', $resource);
```

`copy` 方法将磁盘中已存在的文件从一个地方拷贝到另一个地方：

```
Storage::copy('old/file1.jpg', 'new/file1.jpg');
```

`move` 方法将磁盘中已存在的文件从一定地方移到到另一个地方：

```
Storage::move('old/file1.jpg', 'new/file1.jpg');
```

添加内容到文件开头/结尾

`prepend` 和 `append` 方法允许你轻松插入内容到文件开头/结尾：

```
Storage::prepend('file.log', 'Prepended Text');
Storage::append('file.log', 'Appended Text');
```

3.4 删除文件

`delete` 方法接收单个文件名或多个文件数组并将其从磁盘移除：

```
Storage::delete('file.jpg');
Storage::delete(['file1.jpg', 'file2.jpg']);
```

3.5 目录

获取一个目录下的所有文件

`files` 方法返回给定目录下的所有文件数组，如果你想要获取给定目录下包含子目录的所有文件列表，可以使用 `allFiles` 方法：

```
$files = Storage::files($directory);
$files = Storage::allFiles($directory);
```

获取一个目录下的所有子目录

`directories` 方法返回给定目录下所有目录数组，此外，可以使用 `allDirectories` 方法获取嵌套的所有子目录数组：

```
$directories = Storage::directories($directory);
// 递归...
$directories = Storage::allDirectories($directory);
```

创建目录

`makeDirectory` 方法将会创建给定目录，包含子目录（递归）：

```
Storage::makeDirectory($directory);
```

删除目录

最后，`deleteDirectory` 方法用于移除目录，包括该目录下的所有文件：

```
Storage::deleteDirectory($directory);
```

4、自定义文件系统

Laravel 的 Flysystem 集成支持自定义驱动，为了设置自定义的文件系统你需要创建一个服务提供者如 `DropboxServiceProvider`。在该提供者的 `boot` 方法中，你可以使用 `Storage` 门面的 `extend` 方法定义自定义驱动：

```
<?php

namespace App\Providers;

use Storage;
use League\Flysystem\Filesystem;
```

```
use Dropbox\Client as DropboxClient;
use Illuminate\Support\ServiceProvider;
use League\Flysystem\Dropbox\DropboxAdapter;

class DropboxServiceProvider extends ServiceProvider{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Storage::extend('dropbox', function($app, $config) {
            $client = new DropboxClient(
                $config['accessToken'], $config['clientIdentifie
r']
            );

            return new Filesystem(new DropboxAdapter($client));
        });
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

`extend` 方法的第一个参数是驱动名称，第二个参数是获取`$app` 和`$config` 变量的闭包。该解析器闭包必须返回一个 `League\Flysystem\Filesystem` 实例。$config 变量包含了定义在配置文件 `config/filesystems.php` 中为特定磁盘定义的选项。

创建好注册扩展的服务提供者后，就可以使用配置文件 `config/filesystem.php` 中的 `dropbox` 驱动了。

哈希

1、简介

Laravel 的 Hash 门面为存储用户密码提供了安全的 Bcrypt 哈希算法。如果你正在使用 Laravel 应用自带的 `AuthController` 控制器，将会自动为注册和认证使用该 Bcrypt。Bcrypt 是散列密码的绝佳选择，因为其“工作因子”是可调整的，这意味着随着硬件功能的提升，生成哈希所花费的时间也会增加。

2、基本使用

可以调用 `Hash` 门面上的 `make` 方法散列存储密码：

```
<?php

namespace App\Http\Controllers;

use Hash;
use App\User;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserController extends Controller{
    /**
     * 更新用户密码
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function updatePassword(Request $request, $id)
    {
        $user = User::findOrFail($id);

        // 验证新密码长度...

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

此外，还可以使用全局的辅助函数 `bcrypt`：

```
bcrypt('plain-text');
```

验证哈希密码

`check` 方法允许你验证给定原生字符串和给定哈希是否相等，然而，如果你在使用 Laravel 自带的 `AuthController`（详见用户认证一节），就不需要再直接使用该方法，因为自带的认证控制器自动调用了该方法：

```
if (Hash::check('plain-text', $hashedPassword)) {
    // 密码匹配...
}
```

检查密码是否需要被重新哈希

`needsRehash` 方法允许你判断哈希计算器使用的工作因子在上次密码被哈希后是否发生改变：

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

辅助函数

1、简介

Laravel 自带了一系列 PHP 辅助函数，很多被框架自身使用，如果你觉得方便的话也可以在代码中使用它们。

2、 数组函数

array_add()

`array_add` 函数添加给定键值对到数组，如果给定键不存在的话：

```
$array = array_add(['name' => 'Desk'], 'price', 100);

// ['name' => 'Desk', 'price' => 100]
```

array_collapse()

`array_collapse` 函数将多个数组合并成一个：

```
$array = array_collapse([[1, 2, 3], [4, 5, 6], [7, 8, 9]]);
// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

array_divide()

`array_divide` 函数返回两个数组，一个包含原数组的所有键，另外一个包含原数组的所有值：

```
list($keys, $values) = array_divide(['name' => 'Desk']);

// $keys: ['name']

// $values: ['Desk']
```

array_dot()

`array_dot` 函数使用”.”号将将多维数组转化为一维数组：

```
$array = array_dot(['foo' => ['bar' => 'baz']]);

// ['foo.bar' => 'baz'];
```

array_except()

`array_except` 方法从数组中移除给定键值对：

```
$array = ['name' => 'Desk', 'price' => 100];


$array = array_except($array, ['price']);

// ['name' => 'Desk']
```

array_first()

`array_first` 方法返回通过测试数组的第一个元素：

```
$array = [100, 200, 300];


$value = array_first($array, function ($key, $value) {

    return $value >= 150;});

// 200
```

默认值可以作为第三个参数传递给该方法，如果没有值通过测试的话返回默认值：

```
$value = array_first($array, $callback, $default);
```

array_flatten()

`array_flatten` 方法将多维数组转化为一维数组：

```
$array = ['name' => 'Joe', 'languages' => ['PHP', 'Ruby']];


$array = array_flatten($array);

// ['Joe', 'PHP', 'Ruby'];
```

array_forget()

`array_forget` 方法使用”.”号从嵌套数组中移除给定键值对：

```
$array = ['products' => ['desk' => ['price' => 100]]];


array_forget($array, 'products.desk');

// ['products' => []]
```

array_get()

`array_get` 方法使用”.”号从嵌套数组中获取值：

```
$array = ['products' => ['desk' => ['price' => 100]]];


$value = array_get($array, 'products.desk');

// ['price' => 100]
```

`array_get` 函数还接收一个默认值，如果指定键不存在的话则返回该默认值：

```
$value = array_get($array, 'names.john', 'default');
```

array_only()

`array_only` 方法只从给定数组中返回指定键值对：

```
$array = ['name' => 'Desk', 'price' => 100, 'orders' => 10];


$array = array_only($array, ['name', 'price']);
```

```
// ['name' => 'Desk', 'price' => 100]
```

array_pluck()

`array_pluck` 方法从数组中返回给定键对应的键值对列表：

```
$array = [
    ['developer' => ['name' => 'Taylor']],
    ['developer' => ['name' => 'Abigail']]];

$array = array_pluck($array, 'developer.name');
// ['Taylor', 'Abigail'];
```

你还可以指定返回结果的键：

```
$array = array_pluck($array, 'developer.name', 'developer.id');
// [1 => 'Taylor', 2 => 'Abigail'];
```

array_prepend()

`array_prepend` 函数将数据项推入数组开头：

```
$array = ['one', 'two', 'three', 'four'];

$array = array_prepend($array, 'zero');
// $array: ['zero', 'one', 'two', 'three', 'four']
```

array_pull()

`array_pull` 方法从数组中返回并移除键值对：

```
$array = ['name' => 'Desk', 'price' => 100];

$name = array_pull($array, 'name');
// $name: Desk

// $array: ['price' => 100]
```

array_set()

`array_set` 方法在嵌套数组中使用”.”号设置值：

```
$array = ['products' => ['desk' => ['price' => 100]]];


array_set($array, 'products.desk.price', 200);

// ['products' => ['desk' => ['price' => 200]]]
```

array_sort()

`array_sort` 方法通过给定闭包的结果对数组进行排序：

```
$array = [
    ['name' => 'Desk'],
    ['name' => 'Chair'],
];


$array = array_values(array_sort($array, function ($value) {
    return $value['name'];
}));


/*
    [
        ['name' => 'Chair'],
        ['name' => 'Desk'],
    ]
*/
```

array_sort_recursive()

`array_sort_recursive` 函数使用 `sort` 函数对数组进行递归排序：

```
$array = [
    [
```

```
        'Roman',
        'Taylor',
        'Li',
    ],
    [
        'PHP',
        'Ruby',
        'JavaScript',
    ],
];

$array = array_sort_recursive($array);

/*
    [
        [
            'Li',
            'Roman',
            'Taylor',
        ],
        [
            'JavaScript',
            'PHP',
            'Ruby',
        ]
    ];
*/
```

array_where()

`array_where` 函数使用给定闭包对数组进行过滤：

```
$array = [100, '200', 300, '400', 500];


$array = array_where($array, function ($key, $value) {
    return is_string($value);
});
// [1 => 200, 3 => 400]
```

head()

`head` 函数只是简单返回给定数组的第一个元素：

```
$array = [100, 200, 300];


$first = head($array);
// 100
```

last()

`last` 函数返回给定数组的最后一个元素：

```
$array = [100, 200, 300];


$last = last($array);
// 300
```

3、路径函数

app_path()

`app_path` 函数返回 `app` 目录的绝对路径：

```
$path = app_path();
```

你还可以使用 `app_path` 函数为相对于 `app` 目录的给定文件生成绝对路径：

```
$path = app_path('Http/Controllers/Controller.php');
```

base_path()

`base_path` 函数返回项目根目录的绝对路径：

```
$path = base_path();
```

你还可以使用 `base_path` 函数为相对于应用目录的给定文件生成绝对路径：

```
$path = base_path('vendor/bin');
```

config_path()

`config_path` 函数返回应用配置目录的绝对路径：

```
$path = config_path();
```

database_path()

`database_path` 函数返回应用数据库目录的绝对路径：

```
$path = database_path();
```

elixir()

`elixir` 函数返回版本控制的 Elixir 文件所在路径：

```
elixir($file);
```

public_path()

`public_path` 函数返回 `public` 目录的绝对路径：

```
$path = public_path();
```

storage_path()

`storage_path` 函数返回 `storage` 目录的绝对路径：

```
$path = storage_path();
```

还可以使用 `storage_path` 函数生成相对于 `storage` 目录的给定文件的绝对路径：

```
$path = storage_path('app/file.txt');
```

4、字符串函数

camel_case()

`camel_case` 函数将给定字符串转化为按驼峰式命名规则的字符串：

```
$camel = camel_case('foo_bar');

// fooBar
```

class_basename()

`class_basename` 返回给定类移除命名空间后的类名：

```
$class = class_basename('Foo\Bar\Baz');

// Baz
```

e()

`e` 函数在给定字符串上运行 `htmlentities`：

```
echo e('<html>foo</html>');

// &lt;html&gt;foo&lt;/html&gt;
```

ends_with()

`ends_with` 函数判断给定字符串是否以给定值结尾：

```
$value = ends_with('This is my name', 'name');

// true
```

snake_case()

`snake_case` 函数将给定字符串转化为下划线分隔的字符串：

```
$snake = snake_case('fooBar');

// foo_bar
```

str_limit()

`str_limit` 函数限制输出字符串的数目，该方法接收一个字符串作为第一个参数以及该字符串最大输出字符数作为第二个参数：

```
$value = str_limit('The PHP framework for web artisans.', 7);

// The PHP...
```

starts_with()

`starts_with` 函数判断给定字符串是否以给定值开头：

```
$value = starts_with('This is my name', 'This');

// true
```

str_contains()

`str_contains` 函数判断给定字符串是否包含给定值：

```
$value = str_contains('This is my name', 'my');

// true
```

str_finish()

`str_finish` 函数添加字符到字符串结尾：

```
$string = str_finish('this/string', '/');

// this/string/
```

str_is()

`str_is` 函数判断给定字符串是否与给定模式匹配，星号可用于表示通配符：

```
$value = str_is('foo*', 'foobar');

// true

$value = str_is('baz*', 'foobar');

// false
```

str_plural()

`str_plural` 函数将字符串转化为复数形式，该函数当前只支持英文：

```
$plural = str_plural('car');

// cars

$plural = str_plural('child');
```

```
// children
```

还可以传递整型数据作为第二个参数到该函数以获取字符串的单数或复数形式：

```
$plural = str_plural('child', 2);
// children
$plural = str_plural('child', 1);
// child
```

str_random()

`str_random` 函数通过指定长度生成随机字符串：

```
$string = str_random(40);
```

str_singular()

`str_singular` 函数将字符串转化为单数形式，该函数目前只支持英文：

```
$singular = str_singular('cars');

// car
```

str_slug()

`str_slug` 函数将给定字符串生成 URL 友好的格式：

```
$title = str_slug("Laravel 5 Framework", "-");

// laravel-5-framework
```

studly_case()

`studly_case` 函数将给定字符串转化为单词开头字母大写的格式：

```
$value = studly_case('foo_bar');

// FooBar
```

trans()

`trans` 函数使用本地文件翻译给定语言行：

```
echo trans('validation.required'):
```

trans_choice()

`trans_choice` 函数翻译带拐点的给定语言行：

```
$value = trans_choice('foo.bar', $count);
```

5、URL 函数

action()

`action` 函数为给定控制器动作生成 URL，你不需要传递完整的命名空间到该控制器，传递相对于命名空间 `App\Http\Controllers` 的类名即可：

```
$url = action('HomeController@getIndex');
```

如果该方法接收路由参数，你可以将其作为第二个参数传递进来：

```
$url = action('UserController@profile', ['id' => 1]);
```

asset()

使用当前请求的 scheme（HTTP 或 HTTPS）为前端资源生成一个 URL：

```
$url = asset('img/photo.jpg');
```

secure_asset()

使用 HTTPS 为前端资源生成一个 URL：

```
echo secure_asset('foo/bar.zip', $title, $attributes = []);
```

route()

`route` 函数为给定命名路由生成一个 URL：

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

如果该路由接收参数，你可以将其作为第二个参数传递进来：

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

url()

`url` 函数为给定路径生成绝对路径：

```
echo url('user/profile');


echo url('user/profile', [1]);
```

如果没有提供路径，将会返回 `Illuminate\Routing\UrlGenerator` 实例：

```
echo url()->current();
echo url()->full();
echo url()->previous();
```

6、其它函数

auth()

`auth` 函数返回一个认证器实例，为方便起见你可以用其取代 `Auth` 门面：

```
$user = auth()->user();
```

back()

`back` 函数生成重定向响应到用户前一个位置：

```
return back();
```

bcrypt()

`bcrypt` 函数使用 Bcrypt 对给定值进行哈希，你可以用其替代 `Hash` 门面：

```
$password = bcrypt('my-secret-password');
```

collect()

`collect` 函数会根据提供的数据项创建一个集合：

```
$collection = collect(['taylor', 'abigail']);
```

config()

`config` 函数获取配置变量的值，配置值可以通过使用"."号访问，包含文件名以及你想要访问的选项。如果配置选项不存在的话默认值将会被指定并返回：

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

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

辅助函数 `config` 还可以用于在运行时通过传递键值对数组设置配置变量值：

```
config(['app.debug' => true]);
```

csrf_field()

`csrf_field` 函数生成一个包含 CSRF 令牌值的 HTML 隐藏域，例如，使用 Blade 语法：

```
{!! csrf_field() !!}
```

csrf_token()

`csrf_token` 函数获取当前 CSRF 令牌的值：

```
$token = csrf_token();
```

dd()

`dd` 函数输出给定变量值并终止脚本执行：

```
dd($value);
```

dispatch()

`dispatch` 函数推送一个新的任务到 Laravel 任务队列：

```
dispatch(new App\Jobs\SendEmails);
```

env()

`env` 函数获取环境变量值或返回默认值：

```
$env = env('APP_ENV');

// Return a default value if the variable doesn't exist...

$env = env('APP_ENV', 'production');
```

event()

`event` 函数分发给定事件到对应监听器：

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

factory()

`factory` 函数为给定类、名称和数量创建模型工厂构建器，可用于测试或数据填充：

```
$user = factory('App\User')->make();
```

method_field()

`method_field` 函数生成包含 HTTP 请求方法的 HTML 隐藏域，例如：

```
<form method="POST">
    {!! method_field('delete') !!}</form>
```

old()

`old` 函数获取一次性存放在 Session 中的值：

```
$value = old('value');

$value = old('value', 'default');
```

redirect()

`redirect` 函数返回重定向器实例进行重定向：

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

request()

`request` 函数返回当前请求实例或者获取一个输入项：

```
$request = request();$value = request('key', $default = null)
```

response()

`response` 函数创建一个响应实例或者获取响应工厂实例：

```
return response('Hello World', 200, $headers);

return response()->json(['foo' => 'bar'], 200, $headers)
```

session()

`session` 函数可以用于获取/设置 Session 值：

```
$value = session('key');
```

可以通过传递键值对数组到该函数的方式设置 Session 值：

```
session(['chairs' => 7, 'instruments' => 3]);
```

如果没有传入参数到 `session` 函数则返回 Session 存储器对象实例：

```
$value = session()->get('key');session()->put('key', $value);
```

value()

`value` 函数返回给定的值，然而，如果你传递一个闭包到该函数，该闭包将会被执行并返回执行结果：

```
$value = value(function() { return 'bar'; });
```

view()

`view` 函数获取一个视图实例：

```
return view('auth.login');
```

with()

`with` 函数返回给定的值，该函数在方法链中特别有用，别的地方就没什么用了：

```
$value = with(new Foo)->work();
```

本地化

1、简介

Laravel 的本地化特性允许你在应用中轻松实现多种语言支持。
语言字符串默认存放在 `resources/lang` 目录中，在该目录中应该包含应用支持的每种语言的子目录：

```
/resources
    /lang
        /en
```

```
            messages.php
        /es
            messages.php
```

所有语言文件都返回一个键值对数组，例如：

```
<?php

return [
    'welcome' => 'Welcome to our application'
];
```

配置 Locale 选项

应用默认语言存放在配置文件 `config/app.php` 中，当然，你可以修改该值来匹配应用需要。你还可以在运行时使用 `App` 门面上的 `setLocale` 方法改变当前语言：

```
Route::get('welcome/{locale}', function ($locale) {
    App::setLocale($locale);
    //
});
```

你还可以配置一个“备用语言”，当当前语言不包含给定语言行时备用语言被返回。和默认语言一样，备用语言也在配置文件 `config/app.php` 中配置：

```
'fallback_locale' => 'en',
```

2、基本使用

你可以使用帮助函数 `trans` 从语言文件中获取行，该方法接收文件和语言行的键作为第一个参数，例如，让我们在语言文件 `resources/lang/messages.php` 中获取语言行 `welcome`：

```
echo trans('messages.welcome');
```

当然如果你使用 Blade 模板引擎，可以使用`{{ }}`语法打印语言行：

```
{{ trans('messages.welcome') }}
```

如果指定的语言行不存在，`trans` 函数将返回语言行的键，所以，使用上面的例子，如果语言行不存在的话，`trans` 函数将返回 `messages.welcome`。

替换语言行中的参数

如果需要的话，你可以在语言行中定义占位符，所有的占位符都有一个:前缀，例如，你可以用占位符名称定义一个 `welcome` 消息：

```
'welcome' => 'Welcome, :name',
```

要在获取语言行的时候替换占位符，传递一个替换数组作为 `trans` 函数的第二个参数：

```
echo trans('messages.welcome', ['name' => 'Dayle']);
```

多元化

多元化是一个复杂的问题，因为不同语言对多元化有不同的规则，通过使用管道字符“|”，你可以区分一个字符串的单数和复数形式：

```
'apples' => 'There is one apple|There are many apples',
```

然后，你可以使用 `trans_choice` 函数获取给定行数的语言行，在本例中，由于行数大于 1，将会返回语言行的复数形式：

```
echo trans_choice('messages.apples', 10);
```

Laravel 翻译器由 Symfony 翻译组件提供，因此你可以创建更复杂的多元化规则：

```
'apples' => '{0} There are none|[1,19] There are some|[20,Inf] There are many',
```

3、覆盖 Vendor 包的语言文件

有些包可以处理自己的语言文件。你可以通过将自己的文件放在 `resources/lang/vendor/{package}/{locale}`目录下来覆盖它们而不是破坏这些包的核心文件来调整这些句子。
所以，举个例子，如果你需要覆盖名为 `skyrim/hearthfire` 的包中的 `messages.php` 文件里的英文句子，可以创建一个 `resources/lang/vendor/hearthfire/en/messages.php` 文件。在这个文件中只需要定义你想要覆盖的句子，你没有覆盖的句子仍然从该包原来的语言文件中加载。

邮件

1、简介

Laravel 基于 SwiftMailer 库提供了一套干净清爽的邮件 API。Laravel 为 SMTP、Mailgun、Mandrill、Amazon SES、PHP 的 `mail` 函数，以及 `sendmail` 提供了驱动，从而允许你快速通过本地或云服务发送邮件。

邮件驱动预备知识

基于驱动的 API 如 Mailgun 和 Mandrill 通常比 SMTP 服务器更简单、更快。所有的 API 驱动要求应用已经安装 Guzzle HTTP 库。你可以通过添加如下行到 `composer.json` 文件来安装 Guzzle 到项目：

```
"guzzlehttp/guzzle": "~5.3|~6.0"
```

Mailgun 驱动

要使用 Mailgun 驱动（Mailgun 前 10000 封邮件免费，后续收费），首先安装 Guzzle，然后在配置文件 `config/mail.php` 中设置 `driver` 选项为 `mailgun`。接下来，验证配置文件 `config/services.php` 包含如下选项：

```
'mailgun' => [
    'domain' => 'your-mailgun-domain',
    'secret' => 'your-mailgun-key',],
```

Mandrill 驱动

要使用 Mandrill 驱动（Mandrill 不支持中国区用户注册，汗！），首先安装 Guzzle，然后在配置文件 `config/mail.php` 中设置 `driver` 选项值为 `mandrill`。接下来，验证配置文件 `config/services.php` 包含如下选项：

```
'mandrill' => [
    'secret' => 'your-mandrill-key',],
```

SES 驱动

要使用 Amazon SES 驱动（收费），安装 Amazon AWS 的 PHP SDK，你可以通过添加如下行到 `composer.json` 文件的 `require` 部分来安装该库：

```
"aws/aws-sdk-php": "~3.0"
```

接下来，设置配置文件 `config/mail.php` 中的 `driver` 选项为 `ses`。然后，验证配置文件 `config/services.php` 包含如下选项：

```
'ses' => [
    'key' => 'your-ses-key',
    'secret' => 'your-ses-secret',
    'region' => 'ses-region',  // e.g. us-east-1
],
```

2、发送邮件

Laravel 允许你在视图中存储邮件信息，例如，要组织你的电子邮件，可以在 `resources/views` 目录下创建 `emails` 目录。

要发送一条信息，使用 `Mail` 门面上的 `send` 方法。`send` 方法接收三个参数。第一个参数是包含邮件信息的视图名称；第二个参数是你想要传递到该视图的数组数据；第三个参数是接收消息实例的闭包回调——允许你自定义收件人、主题以及邮件其他方面的信息：

```
<?php

namespace App\Http\Controllers;

use Mail;
use App\User;
```

```
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserController extends Controller{
    /**
     * 发送邮件给用户
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function sendEmailReminder(Request $request, $id)
    {
        $user = User::findOrFail($id);

        Mail::send('emails.reminder', ['user' => $user], functio
n ($m) use ($user) {
            $m->from('hello@app.com', 'Your Application');$m->to
($user->email, $user->name)->subject('Your Reminder!');
        });
    }
}
```

由于我们在上例中传递一个包含 `user` 键的数组，我们可以在邮件中使用如下方式显示用户名：

```
<?php echo $user->name; ?>
```

注意：`$message` 变量总是被传递到邮件视图，并允许嵌入附件，因此，你应该在视图负载中避免传入消息变量。

构造消息

正如前面所讨论的，传递给 `send` 方法的第三个参数是一个允许你指定邮件消息本身多个选项的闭包。使用这个闭包可以指定消息的其他属性，例如抄送、群发，等等：

```
Mail::send('emails.welcome', $data, function ($message) {
    $message->from('us@example.com', 'Laravel');
    $message->to('foo@example.com')->cc('bar@example.com');
});
```

下面试`$message` 消息构建器实例上的可用方法：

```
$message->from($address, $name = null);
$message->sender($address, $name = null);
$message->to($address, $name = null);
$message->cc($address, $name = null);
$message->bcc($address, $name = null);
$message->replyTo($address, $name = null);
```

```
$message->subject($subject);
$message->priority($level);
$message->attach($pathToFile, array $options = []);
// 从$data 字符串追加文件...
$message->attachData($data, $name, array $options = []);
// 获取底层 SwiftMailer 消息实例...
$message->getSwiftMessage();
```

注意：传递给 `Mail::send` 闭包的消息实例继承自 `SwiftMailer` 消息类，该实例允许你调用该类上的任何方法来构建自己的电子邮件消息。

纯文本邮件

默认情况下，传递给 `send` 方法的视图假定包含 HTML，然而，通过传递数组作为第一个参数到 `send` 方法，你可以指定发送除 HTML 视图之外的纯文本视图：

```
Mail::send(['html.view', 'text.view'], $data, $callback);
```

或者，如果你只需要发送纯文本邮件，可以指定在数组中使用 text 键：

```
Mail::send(['text' => 'view'], $data, $callback);
```

原生字符串邮件

如果你想要直接发送原生字符串邮件你可以使用 `raw` 方法：

```
Mail::raw('Text to e-mail', function ($message) {
    //
});
```

2.1 附件

要添加附件到邮件，使用传递给闭包的`$message` 对象上的 `attach` 方法。该方法接收文件的绝对路径作为第一个参数：

```
Mail::send('emails.welcome', $data, function ($message) {
    //
    $message->attach($pathToFile);
});
```

当添加文件到消息时，你还可以通过传递数组作为第二个参数到 `attach` 方法来指定文件显示名和 MIME 类型：

```
$message->attach($pathToFile, ['as' => $display, 'mime' => $mim
e]);
```

2.2 内联附件

在邮件视图中嵌入一张图片

嵌套内联图片到邮件中通常是很笨重的，然而，Laravel 提供了一个便捷的方式附加图片到邮件并获取相应的 CID，要嵌入内联图片，在邮件视图中使用`$message`变量上的`embed`方法。记住，Laravel 自动在所有邮件视图中传入`$message`变量使其有效：

```
<body>
    Here is an image:
    <img src="<?php echo $message->embed($pathToFile); ?>">
</body>
```

在邮件视图中嵌入原生数据

如果你想要在邮件消息中嵌入原生数据字符串，可以使用`$message`变量上的`embedData`方法：

```
<body>
    Here is an image from raw data:
    <img src="<?php echo $message->embedData($data, $name); ?>
">
</body>
```

2.3 邮件队列

邮件消息队列

发送邮件消息可能会大幅度延长应用的响应时间，许多开发者选择将邮件发送放到队列中再后台执行，Laravel 中可以使用内置的统一队列 API 来实现。要将邮件消息放到队列中，使用`Mail`门面上的`queue`方法：

```
Mail::queue('emails.welcome', $data, function ($message) {
    //
});
```

该方法自动将邮件任务推送到队列中以便在后台发送。当然，你需要在使用该特性前配置队列。

延迟消息队列

如果你想要延迟已经放到队列中邮件的发送，可以使用`later`方法。只需要传递你想要延迟发送的秒数作为第一个参数到该方法即可：

```
Mail::later(5, 'emails.welcome', $data, function ($message) {
    //
});
```

推入指定队列

如果你想要将邮件消息推送到指定队列，可以使用`queueOn`和`laterOn`方法：

```
Mail::queueOn('queue-name', 'emails.welcome', $data, function
($message) {
    //
});

Mail::laterOn('queue-name', 5, 'emails.welcome', $data, functio
n ($message) {
    //
});
```

3、邮件&本地开发

开发发送邮件的应用时，你可能不想要真的发送邮件到有效的电子邮件地址，而只是想要做下测试。Laravel 提供了几种方式“禁止”邮件的实际发送。

日志驱动

一种解决方案是在本地开发时使用 `log` 邮件驱动。该驱动将所有邮件信息写到日志文件中以备查看，想要了解更多关于每个环境的应用配置信息，查看配置文档。

通用配置

Laravel 提供的另一种解决方案是为框架发送的所有邮件设置通用收件人，这样的话，所有应用生成的邮件将会被发送到指定地址，而不是实际发送邮件指定的地址。这可以通过在配置文件 `config/mail.php` 中设置 `to` 选项来实现：

```
'to' => [
    'address' => 'dev@domain.com',
    'name' => 'Dev Example'
],
```

Mailtrap

最后，你可以使用 Mailtrap 服务和 `smtp` 驱动发送邮件信息到“虚拟”邮箱，这种方法允许你在 Mailtrap 的消息查看器中查看最终的邮件。

4、事件

Laravel 会发送邮件前触发一个事件，记住，这个事件是在邮件被发送时触发，而不是推送到队列时，你可以在 `EventServiceProvider` 中注册事件监听器：

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
```

```
    'Illuminate\Mail\Events\MessageSending' => [
        'App\Listeners\LogSentMessage',
    ],
];
```

包开发

1、简介

包是添加功能到 Laravel 的主要方式。包可以提供任何功能，小到处理日期如 Carbon，大到整个 BDD 测试框架如 Behat。
当然，有很多不同类型的包。有些包是独立的，意味着可以在任何框架中使用，而不仅是 Laravel。比如 Carbon 和 Behat 都是独立的包。所有这些包都可以通过在 `composer.json` 文件中请求以便被 Laravel 使用。
另一方面，其它包只能特定和 Laravel 一起使用，这些包可能有路由，控制器、视图和配置用于加强 Laravel 应用的功能，本章主要讨论只能在 Laravel 中使用的包。

2、服务提供者

服务提供者是包和 Laravel 之间的连接点。服务提供者负责绑定对象到 Laravel 的服务容器并告知 Laravel 从哪里加载包资源如视图、配置和本地化文件。
服务提供者继承自 `Illuminate\Support\ServiceProvider` 类并包含两个方法：`register` 和 `boot`。`ServiceProvider` 基类位于 Composer 包 `illuminate/support`。
要了解更多关于服务提供者的内容，查看其文档。

3、路由

要定义包的路由，只需要在包服务提供者中的 `boot` 方法中引入路由文件。在路由文件中，可以使用 `Route` 门面注册路由，和 Laravel 应用中注册路由一样：

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot(){
    if (! $this->app->routesAreCached()) {
        require __DIR__.'/../../routes.php';
    }
}
```

4、资源

4.1 视图

要在 Laravel 中注册包视图，需要告诉 Laravel 视图在哪，可以使用服务提供者的 `loadViewsFrom` 方法来实现。`loadViewsFrom` 方法接收两个参数：视图模板的路径和包名称。例如，如果你的包名称是“courier”，添加如下代码到服务提供者的 `boot` 方法：

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot(){
    $this->loadViewsFrom(__DIR__.'/path/to/views', 'courier');
}
```

包视图通过使用类似的 `package::view` 语法来引用。所以，你可以通过如下方式加载 `courier` 包上的 `admin` 视图：

```
Route::get('admin', function () {
    return view('courier::admin');
});
```

覆盖包视图

当你使用 `loadViewsFrom` 方法的时候，Laravel 实际上为视图注册了两个存放位置：一个是 `resources/views/vendor` 目录，另一个是你指定的目录。所以，以 `courier` 为例：当请求一个包视图时，Laravel 首先检查开发者是否在 `resources/views/vendor/courier` 提供了自定义版本的视图，如果该视图不存在，Laravel 才会搜索你调用 `loadViewsFrom` 方法时指定的目录。这种机制使得终端用户可以轻松地自定义/覆盖包视图。

发布视图

如果你想要视图能够发布到应用的 `resources/views/vendor` 目录，可以使用服务提供者的 `publishes` 方法。该方法接收包视图路径及其相应的发布路径数组作为参数：

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot(){
    $this->loadViewsFrom(__DIR__.'/path/to/views', 'courier');

    $this->publishes([
        __DIR__.'/path/to/views' => base_path('resources/views/
vendor/courier'),
```

```
    ]);
}
```

现在，当包用户执行 Laravel 的 Artisan 命令 `vendor:publish` 时，你的视图包将会被拷贝到指定路径。

4.2 翻译

如果你的包包含翻译文件，你可以使用 `loadTranslationsFrom` 方法告诉 Laravel 如何加载它们，例如，如果你的包命名为“courier”，你应该添加如下代码到服务提供者的 `boot` 方法：

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot(){
    $this->loadTranslationsFrom(__DIR__.'/path/to/translations
', 'courier');
}
```

包翻译使用形如 `package::file.line` 的语法进行引用。所以，你可以使用如下方式从 `messages` 文件中加载 `courier` 包的 `welcome` 行：

```
echo trans('courier::messages.welcome');
```

发布翻译文件

如果你想要发布包翻译到应用的 resources/lang/vendor 目录，你可以使用服务提供者的 publishes 方法，该方法接收一个包路径和相应发布路径数组参数，例如，要发布 courier 包的翻译文件，可以这么做：

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */public function boot(){
    $this->loadTranslationsFrom(__DIR__.'/path/to/translations
', 'courier');

    $this->publishes([
        __DIR__.'/path/to/translations' => resource_path('lang/
vendor/courier'),
    ]);}
```

这样，包用户可以执行 Artisan 命令 vendor:publish 将包翻译文件发布到应用的指定目录。

4.3 配置

通常，你想要发布包配置文件到应用根目录下的 `config` 目录，这将允许包用户轻松覆盖默认配置选项，要发布一个配置文件，只需在服务提供者的 `boot` 方法中使用 `publishes` 方法即可：

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot(){
    $this->publishes([
        __DIR__.'/path/to/config/courier.php' => config_path('c
ourier.php'),
    ]);
}
```

现在，当包用户执行 Laravel 的 Artisan 命令 `vendor:publish` 时，你的文件将会被拷贝到指定位置，当然，配置被发布后，可以通过和其它配置选项一样的方式进行访问：

```
$value = config('courier.option');
```

默认包配置

你还可以选择将自己的包配置文件合并到应用的拷贝版本，这允许用户只引入他们在应用配置文件中实际想要覆盖的配置选项。要合并两个配置，在服务提供者的 `register` 方法中使用 `mergeConfigFrom` 方法即可：

```
/**
 * Register bindings in the container.
 *
 * @return void
 */
public function register(){
    $this->mergeConfigFrom(
        __DIR__.'/path/to/config/courier.php', 'courier'
    );
}
```

5、前端资源

你的包可能包含 JavaScript、CSS 和图片，要发布这些前端资源到应用根目录下的 `public` 目录，使用服务提供者的 `publishes` 方法。在本例中，我们添加一个前端资源组标签 `public`，用于发布相关的前端资源组：

```
/**
```

```
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot(){
    $this->publishes([
        __DIR__.'/path/to/assets' => public_path('vendor/courie
r'),
    ], 'public');
}
```

现在，当包用户执行 `vendor:publish` 命令时，前端资源将会被拷贝到指定位置，由于你需要在每次包更新时重写前端资源，可以使用`--force` 标识：

```
php artisan vendor:publish --tag=public --force
```

如果你想要确保前端资源已经更新到最新版本，可添加该命令到 `composer.json` 文件的 `post-update-cmd` 列表。

6、发布文件组

你可能想要分开发布包前端资源组和资源，例如，你可能想要用户发布包配置的同时不发布包前端资源，可以通过在调用时给它们打上“标签”来实现分离。下面我们在包服务提供者的 `boot` 方法中定义两个公共组：

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot(){
    $this->publishes([
        __DIR__.'/../config/package.php' => config_path('packag
e.php')
    ], 'config');

    $this->publishes([
        __DIR__.'/../database/migrations/' => database_path('mi
grations')
    ], 'migrations');
}
```

现在用户可以在使用 Artisan 命令 `vendor:publish` 时通过引用标签名来分开发布这两个组：

```
php artisan vendor:publish --provider="Vendor\Providers\PackageServiceProvider" --tag="config"
```

分页

1、简介

在其他框架中，分页是件非常痛苦的事，Laravel 则使其变得轻而易举。Laravel 能够基于当前页智能生成一定范围的链接，且生成的 HTML 兼容 Bootstrap CSS 框架。

2、基本使用

2.1 基于查询构建器分页

有多种方式实现分页，最简单的方式就是使用查询构建器或 Eloquent 模型的 `paginate` 方法。该方法基于当前用户查看页自动设置合适的偏移（offset）和限制（limit）。默认情况下，当前页通过 HTTP 请求查询字符串参数 `?page` 的值判断。当然，该值由 Laravel 自动检测，然后自动插入分页器生成的链接中。
让我们先来看看如何在查询上调用 `paginate` 方法。在本例中，传递给 `paginate` 的唯一参数就是你每页想要显示的数目，这里我们指定每页显示 `15` 个：

```
<?php

namespace App\Http\Controllers;

use DB;
use App\Http\Controllers\Controller;

class UserController extends Controller{
    /**
     * 显示应用中的所有用户
     *
     * @return Response
     */
    public function index()
    {
        $users = DB::table('users')->paginate(15);
        return view('user.index', ['users' => $users]);
    }
}
```

注意：目前，使用 groupBy 的分页操作不能被 Laravel 有效执行，如果你需要在分页结果中使用 groupBy，推荐你手动查询数据库然后创建分页器。

简单分页

如果你只需要在分页视图中简单的显示“下一个”和“上一个”链接，可以使用 simplePaginate 方法来执行该查询。在渲染包含大数据集的视图且不需要显示每个页码时非常有用：

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

2.2 基于 Eloquent 模型分页

你还可以对 Eloquent 查询结果进行分页，在本例中，我们对 User 模型进行分页，每页显示 15 条记录。正如你所看到的，该语法和基于查询构建器的分页差不多：

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

当然，你可以在设置其它约束调价之后调用 paginate，比如 where 子句：

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

你也可以使用 simplePaginate 方法：

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

2.3 手动创建分页器

有时候你可能想要通过传递数组数据来手动创建分页实例，你可以基于自己的需求通过创建 Illuminate\Pagination\Paginator 或 Illuminate\Pagination\LengthAwarePaginator 实例来实现。

Paginator 类不需要知道结果集中数据项的总数；然而，正因如此，该类也没有提供获取最后一页索引的方法。

LengthAwarePaginator 接收参数和 Paginator 几乎一样，只是，它要求传入结果集的总数。

换句话说，Paginator 对应 simplePaginate 方法，而 LengthAwarePaginator 对应 paginate 方法。

当手动创建分页器实例的时候，应该手动对传递到分页器的结果集进行“切片”，如果你不确定怎么做，查看 PHP 函数 array_slice。

3、在视图中显示分页结果

当你调用查询构建器或 Eloquent 查询上的 paginate 或 simplePaginate 方法时，你将会获取一个分页器实例。当调用 paginate 方法时，你将获取 Illuminate\Pagination\LengthAwarePaginator，而调用方法 simplePaginate 时，将会获取 Illuminate\Pagination\Paginator 实例。这些对象提供相关方法描述这些结果集，除了这些帮助函数外，分页器实例本身就是迭代器，可以像数组一样对其进行循环调用。

所以，获取到结果后，可以按如下方式使用 Blade 显示这些结果并渲染页面链接：

```
<div class="container">
    @foreach ($users as $user)
```

```
        {{ $user->name }}
    @endforeach
</div>

{!! $users->links() !!}
```

links 方法将会将结果集中的其它页面链接渲染出来。每个链接已经包含了 ?page 查询字符串变量。记住，render 方法生成的 HTML 兼容 Bootstrap CSS 框架。
注意：我们从 Blade 模板调用 render 方法时，确保使用{!! !!}语法以便 HTML 链接不被过滤。

自定义分页链接

setPath 方法允许你生成分页链接时自定义分页器使用的 URI，例如，如果你想要分页器生成形如 http://example.com/custom/url?page=N 的链接，应该传递 custom/url 到 setPath 方法：

```
Route::get('users', function () {
    $users = App\User::paginate(15);
    $users->setPath('custom/url');
    //
});
```

添加参数到分页链接

你可以使用 appends 方法添加查询参数到分页链接查询字符串。例如，要添加&sort=votes 到每个分页链接，应该像如下方式调用 appends：

```
{!! $users->appends(['sort' => 'votes'])->links() !!}
```

如果你想要添加"哈希片段"到分页链接，可以使用 fragment 方法。例如，要添加#foo 到每个分页链接的末尾，像这样调用 fragment 方法：

```
{!! $users->fragment('foo')->links() !!}
```

更多辅助方法

你还可以通过如下分页器实例上的方法访问更多分页信息：

- $results->count()
- $results->currentPage()
- $results->firstItem()
- $results->hasMorePages()
- $results->lastItem()
- $results->lastPage() (使用 simplePaginate 时无效)
- $results->nextPageUrl()
- $results->perPage()
- $results->previousPageUrl()
- $results->total() (使用 simplePaginate 时无效)
- $results->url($page)

4、将结果转化为 JSON

Laravel 分页器结果类实现了 `Illuminate\Contracts\Support\JsonableInterface` 契约并实现 `toJson` 方法，所以将分页结果转化为 JSON 非常简单。
你还可以简单通过从路由或控制器动作返回分页器实例将转其化为 JSON：

```
Route::get('users', function () {
    return App\User::paginate();
});
```

从分页器转化来的 JSON 包含了元信息如 `total`, `current_page`,`last_page` 等等，实际的结果对象数据可以通过该 JSON 数组中的 `data` 键访问。下面是一个通过从路由返回的分页器实例创建的 JSON 例子：

```
{
   "total": 50,
   "per_page": 15,
   "current_page": 1,
   "last_page": 4,
   "next_page_url": "http://laravel.app?page=2",
   "prev_page_url": null,
   "from": 1,
   "to": 15,
   "data":[
        {
            // Result Object
        },
        {
            // Result Object
        }
   ]
}
```

队列

1、简介

Laravel 队列服务为各种不同的后台队列提供了统一的 API。队列允许你推迟耗时任务（例如发送邮件）的执行，从而大幅提高 web 请求速度。

1.1 配置

队列配置文件存放在 `config/queue.php`。在该文件中你将会找到框架自带的每一个队列驱动的连接配置，包括数据库、Beanstalkd、 IronMQ、 Amazon SQS、 Redis 以及同步（本地使用）驱动。其中还包含了一个 null 队列驱动以拒绝队列任务。

1.2 队列驱动预备知识

数据库

为了使用 `database` 队列驱动，需要一张数据库表来存放任务，要生成创建该表的迁移，运行 Artisan 命令 `queue:table`，迁移被创建好了之后，使用 `migrate` 命令运行迁移：

```
php artisan queue:table
php artisan migrate
```

其它队列依赖

下面是以上列出队列驱动需要安装的依赖：

- Amazon SQS: `aws/aws-sdk-php ~3.0`
- Beanstalkd: `pda/pheanstalk ~3.0`
- Redis: `predis/predis ~1.0`

2、编写任务类

2.1 生成任务类

默认情况下，应用的所有队列任务都存放在 `app/Jobs` 目录。你可以使用 Artisan CLI 生成新的队列任务：

```
php artisan make:job SendReminderEmail
```

该命令将会在 `app/Jobs` 目录下生成一个新的类，并且该类实现了 `Illuminate\Contracts\Queue\ShouldQueue` 接口，告诉 Laravel 该任务应该被推送到队列而不是同步运行。

2.2 任务类结构

任务类非常简单，正常情况下只包含一个当队列处理该任务时被执行的 `handle` 方法，让我们看一个任务类的例子：

```
<?php

namespace App\Jobs;

use App\User;
use App\Jobs\Job;
use Illuminate\Contracts\Mail\Mailer;
use Illuminate\Queue\SerializesModels;
```

```
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendReminderEmail extends Job implements ShouldQueue
{
    use InteractsWithQueue, SerializesModels;

    protected $user;

    /**
     * 创建一个新的任务实例
     *
     * @param  User  $user
     * @return void
     */
    public function __construct(User $user)
    {
        $this->user = $user;
    }

    /**
     * 执行任务
     *
     * @param  Mailer  $mailer
     * @return void
     */
    public function handle(Mailer $mailer)
    {
        $mailer->send('emails.reminder', ['user' => $this->use
r], function ($m) {
            //
        });

        $this->user->reminders()->create(...);
    }
}
```

在本例中，注意我们能够直接将 Eloquent 模型传递到对列任务的构造函数中。由于该任务使用了 `SerializesModels` trait，Eloquent 模型将会在任务被执行是优雅地序列化和反序列化。如果你的队列任务在构造函数中接收 Eloquent 模型，只有模型的主键会被序列化到队列，当任务真正被执行的时候，队列系统会自动从数据库中获取整个模型实例。这对应用而言是完全透明的，从而避免序列化整个 Eloquent 模型实例引起的问题。

`handle` 方法在任务被队列处理的时候被调用，注意我们可以在任务的 `handle` 方法中进行依赖注入。Laravel 服务容器会自动注入这些依赖。

出错

如果任务被处理的时候抛出异常，则该任务将会被自动释放回队列以便再次尝试执行。任务会持续被释放知道尝试次数达到应用允许的最大次数。最大尝试次数通过 Artisan 任务 `queue:listen` 或 `queue:work` 上的 `--tries` 开关来定义。关于运行队列监听器的更多信息可以在下面看到。

手动释放任务

如果你想要手动释放任务，生成的任务类中自带的 `InteractsWithQueue` trait 提供了释放队列任务的 `release` 方法，该方法接收一个参数——同一个任务两次运行之间的等待时间：

```
public function handle(Mailer $mailer){
    if (condition) {
        $this->release(10);
    }
}
```

检查尝试运行次数

正如上面提到的，如果在任务处理期间发生异常，任务会自动释放回队列中，你可以通过 `attempts` 方法来检查该任务已经尝试运行次数：

```
public function handle(Mailer $mailer){
    if ($this->attempts() > 3) {
        //
    }
}
```

3、推送任务到队列

默认的 Laravel 控制器位于 `app/Http/Controllers/Controller.php` 并使用了 `DispatchesJobs` trait。该 trait 提供了一些允许你方便推送任务到队列的方法，例如 `dispatch` 方法：

```
<?php

namespace App\Http\Controllers;

use App\User;
use Illuminate\Http\Request;
use App\Jobs\SendReminderEmail;
use App\Http\Controllers\Controller;

class UserController extends Controller{
    /**
     * 发送提醒邮件到指定用户
     *
```

```
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function sendReminderEmail(Request $request, $id)
    {
        $user = User::findOrFail($id);

        $this->dispatch(new SendReminderEmail($user));
    }
}
```

DispatchesJobs Trait

当然，有时候你想要从应用中路由或控制器之外的某些地方分发任务，因为这个原因，你可以在应用的任何类中包含 `DispatchesJobs` trait，从而获取对分发方法的访问，举个例子，下面是使用该 trait 的示例类：

```
<?php

namespace App;

use Illuminate\Foundation\Bus\DispatchesJobs;

class ExampleClass{
    use DispatchesJobs;
}
```

dispatch 方法

或者，你也可以使用全局的 dispatch 方法：

```
Route::get('/job', function () {
    dispatch(new App\Jobs\PerformTask);
    return 'Done!';
});
```

为任务指定队列

你还可以指定任务被发送到的队列。
根据任务被推送到的不同队列，你可以对队列任务进行“分类”，甚至优先考虑分配给多个队列的 worker 数目。这并不会如队列配置文件中定义的那样将任务推送到不同队列“连接”，而只是在单个连接中发送给特定队列。要指定该队列，使用任务实例上的 `onQueue` 方法，该方法由 Laravel 自带的基类 `App\Jobs\Job` 中的 `Illuminate\Bus\Queueable` trait 提供：

```
<?php
```

```
namespace App\Http\Controllers;

use App\User;
use Illuminate\Http\Request;
use App\Jobs\SendReminderEmail;
use App\Http\Controllers\Controller;

class UserController extends Controller{
    /**
     * 发送提醒邮件到指定用户
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function sendReminderEmail(Request $request, $id)
    {
        $user = User::findOrFail($id);
        $job = (new SendReminderEmail($user))->onQueue('emails
');
        $this->dispatch($job);
    }
}
```

3.1 延迟任务

有时候你可能想要延迟队列任务的执行。例如，你可能想要将一个注册 15 分钟后给消费者发送提醒邮件的任务放到队列中，可以通过使用任务类上的 `delay` 方法来实现，该方法由 `Illuminate\Bus\Queueable` trait 提供：

```
<?php

namespace App\Http\Controllers;

use App\User;
use Illuminate\Http\Request;
use App\Jobs\SendReminderEmail;
use App\Http\Controllers\Controller;

class UserController extends Controller{
    /**
     * 发送提醒邮件到指定用户
     *
     * @param  Request  $request
```

```
     * @param  int  $id
     * @return Response
     */
    public function sendReminderEmail(Request $request, $id)
    {
        $user = User::findOrFail($id);
        $job = (new SendReminderEmail($user))->delay(60);
        $this->dispatch($job);
    }
}
```

在本例中，我们指定任务在队列中开始执行前延迟 60 秒。

注意：Amazon SQS 服务最大延迟时间是 15 分钟。

3.2 任务事件

任务完成事件

`Queue::after` 方法允许你在队列任务执行成功后注册一个要执行的回调函数。在该回调中我们可以添加日志、统计数据。例如，我们可以在 Laravel 内置的 `AppServiceProvider` 中添加事件回调:

```
<?php


namespace App\Providers;


use Queue;

use Illuminate\Support\ServiceProvider;


class AppServiceProvider extends ServiceProvider

{

    /**

     * Bootstrap any application services.

     *

     * @return void

     */

    public function boot()

    {
```

```
        Queue::after(function ($connection, $job, $data) {

            //

        });

    }


    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

4、运行队列监听器

启动任务监听器

Laravel 包含了一个 Artisan 命令用来运行被推送到队列的新任务。你可以使用 `queue:listen` 命令运行监听器：

```
php artisan queue:listen
```

还可以指定监听器使用哪个队列连接：

```
php artisan queue:listen connection
```

注意一旦任务开始后，将会持续运行直到手动停止。你可以使用一个过程监视器如 Supervisor 来确保队列监听器没有停止运行。

队列优先级

你可以传递逗号分隔的队列连接列表到 `listen` 任务来设置队列优先级：

```
php artisan queue:listen --queue=high,low
```

在本例中，`high` 队列上的任务总是在从 `low` 队列移动任务之前被处理。

指定任务超时参数

你还可以设置每个任务允许运行的最大时间（以秒为单位）：

```
php artisan queue:listen --timeout=60
```

指定队列睡眠时间

此外，可以指定轮询新任务之前的等待时间（以秒为单位）：

```
php artisan queue:listen --sleep=5
```

需要注意的是队列只会在队列上没有任务时“睡眠”，如果存在多个有效任务，该队列会持续运行，从不睡眠。

4.1 Supervisor 配置

Supervisor 为 Linux 操作系统提供的进程监视器，将会在失败时自动重启 `queue:listen` 或 `queue:work` 命令，要在 Ubuntu 上安装 Supervisor，使用如下命令：

```
sudo apt-get install supervisor
```

Supervisor 配置文件通常存放在 `/etc/supervisor/conf.d` 目录，在该目录中，可以创建多个配置文件指示 Supervisor 如何监视进程，例如，让我们创建一个开启并监视 `queue:work` 进程的 `laravel-worker.conf` 文件：

```
[program:laravel-worker]
process_name=%(program_name)s_%(process_num)02d
command=php /home/forge/app.com/artisan queue:work sqs --sleep=
3 --tries=3 --daemon
autostart=true
autorestart=true
user=forge
numprocs=8
redirect_stderr=true
stdout_logfile=/home/forge/app.com/worker.log
```

在本例中，`numprocs` 指令让 Supervisor 运行 8 个 `queue:work` 进程并监视它们，如果失败的话自动重启。配置文件创建好了之后，可以使用如下命令更新 Supervisor 配置并开启进程：

```
sudo supervisord -c /etc/supervisord.conf
sudo supervisorctl -c /etc/supervisor/supervisord.conf
sudo supervisorctl reread
sudo supervisorctl update
sudo supervisorctl start laravel-worker:*
```

要了解更多关于 Supervisor 的使用和配置，查看 Supervisor 文档。此外，还可以使用 Laravel Forge 从 web 接口方便地自动配置和管理 Supervisor 配置。

4.2 后台队列监听器

Artisan 命令 `queue:work` 包含一个`--daemon` 选项来强制队列 worker 持续处理任务而不必重新启动框架。相较于 `queue:listen` 命令该命令对 CPU 的使用有明显降低：

```
php artisan queue:work connection --daemon
php artisan queue:work connection --daemon --sleep=3
php artisan queue:work connection --daemon --sleep=3 --tries=3
```

正如你所看到的，`queue:work` 任务支持大多数 `queue:listen` 中有效的选项。你可以使用 `php artisan help queue:work` 任务来查看所有有效选项。

后台队列监听器编码考虑

后台队列 worker 在处理每个任务时不重启框架，因此，你要在任务完成之前释放资源，举个例子，如果你在使用 GD 库操作图片，那么就在完成时使用 `imagedestroy` 释放内存。类似的，数据库连接应该在后台长时间运行完成后断开，你可以使用 `DB::reconnect` 方法确保获取了一个新的连接。

4.3 部署后台队列监听器

由于后台队列 worker 是常驻进程，不重启的话不会应用代码中的更改，所以，最简单的部署后台队列 worker 的方式是使用部署脚本重启所有 worker，你可以通过在部署脚本中包含如下命令重启所有 worker：

```
php artisan queue:restart
```

该命令会告诉所有队列 worker 在完成当前任务处理后重启以便没有任务被遗漏。

注意：这个命令依赖于缓存系统重启进度表，默认情况下，APC 在 CLI 任务中无法正常工作，如果你在使用 APC，需要在 APC 配置中添加 `apc.enable_cli=1`。

5、处理失败任务

由于事情并不总是按照计划发展，有时候你的队列任务会失败。别担心，它发生在我们大多数人身上！Laravel 包含了一个方便的方式来指定任务最大尝试执行次数，任务执行次数达到最大限制后，会被插入到 `failed_jobs` 表，失败任务的名字可以通过配置文件 `config/queue.php` 来配置。
要创建一个 `failed_jobs` 表的迁移，可以使用 `queue:failed-table` 命令：

```
php artisan queue:failed-table
```

运行队列监听器的时候，可以在 `queue:listen` 命令上使用`--tries` 开关来指定任务最大可尝试执行次数：

```
php artisan queue:listen connection-name --tries=3
```

5.1 失败任务事件

如果你想要注册一个队列任务失败时被调用的事件，可以使用 `Queue::failing` 方法，该事件通过邮件或 HipChat 通知团队。举个例子，我么可以在 Laravel 自带的 `AppServiceProvider` 中附件一个回调到该事件：

```
<?php

namespace App\Providers;

use Queue;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider{
    /**
     * 启动应用服务
     *
     * @return void
     */
    public function boot()
    {
        Queue::failing(function ($connection, $job, $data) {
            // Notify team of failing job...
        });
    }

    /**
     * 注册服务提供者
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

任务类的失败方法

想要更加细粒度的控制，可以在队列任务类上直接定义 `failed` 方法，从而允许你在失败发生时执行指定动作：

```
<?php

namespace App\Jobs;
```

```
use App\Jobs\Job;
use Illuminate\Contracts\Mail\Mailer;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Bus\SelfHandling;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendReminderEmail extends Job implements SelfHandling, Sh
ouldQueue
{
    use InteractsWithQueue, SerializesModels;

    /**
     * 执行任务
     *
     * @param  Mailer  $mailer
     * @return void
     */
    public function handle(Mailer $mailer)
    {
        //
    }

    /**
     * 处理失败任务
     *
     * @return void
     */
    public function failed()
    {
        // Called when the job is failing...
    }
}
```

5.2 重试失败任务

要查看已插入到 `failed_jobs` 数据表中的所有失败任务，可以使用 Artisan 命令 `queue:failed`：

```
php artisan queue:failed
```

该命令将会列出任务 ID，连接，对列和失败时间，任务 ID 可用于重试失败任务，例如，要重试一个 ID 为 5 的失败任务，要用到下面的命令：

```
php artisan queue:retry 5
```

要重试所有失败任务，使用如下命令即可：

```
php artisan queue:retry all
```

如果你要删除一个失败任务，可以使用 `queue:forget` 命令：

```
php artisan queue:forget 5
```

要删除所有失败任务，可以使用 `queue:flush` 命令：

```
php artisan queue:flush
```

Redis

1、简介

Redis 是一个开源的、高级的键值对存储系统，经常被用作数据结构服务器，因为其支持字符串、Hash、列表、集合和有序集合等数据结构。在 Laravel 中使用 Redis 之前，需要通过 Composer 安装 `predis/predis` 包（~1.0）。

配置

应用的 Redis 配置位于配置文件 `config/database.php`。在这个文件中，可以看到包含被应用使用的 Redis 服务器的 `redis` 数组：

```
'redis' => [

    'cluster' => false,

    'default' => [
        'host'     => '127.0.0.1',
        'port'     => 6379,
        'database' => 0,
    ],

],
```

默认服务器配置可以满足开发需要，然而，你可以基于环境随意修改该数组，只需要给每个 Redis 服务器一个名字并指定该 Redis 服务器使用的主机和接口。

`cluster` 选项告诉 Laravel Redis 客户端在多个 Redis 节点间执行客户端分片，从而形成节点池并创建大量有效的 RAM。然而，客户端分片并不处理故障转移，所以，非常适合从另一个主数据存储那里获取有效的缓存数据。
此外，你可以在 Redis 连接定义中定义 options 数组值，从而允许你指定一系列 Predis 客户端选项。
如果 Redis 服务器要求认证信息，你可以通过添加 `password` 配置项到 Redis 服务器配置数组来提供密码。

注意：如果你通过 PECL 安装 PHP 的 Redis 扩展，需要在 `config/app.php` 文件中修改 Redis 的别名。

2、基本使用

你可以通过调用 `Redis` 门面上的多个方法来与 Redis 进行交互，该门面支持动态方法，所以你可以任何 Redis 命令，该命令将会直接传递给 Redis，在本例中，我们通过调用 `Redis` 门面上的 `get` 方法来调用 Redis 上的 GET 命令：

```
<?php

namespace App\Http\Controllers;

use Redis;use App\Http\Controllers\Controller;

class UserController extends Controller{
    /**
     * 显示指定用户属性
     *
     * @param  int  $id
     * @return Response
     */
    public function showProfile($id)
    {
        $user = Redis::get('user:profile:'.$id);
        return view('user.profile', ['user' => $user]);
    }
}
```

当然，如上所述，可以在 `Redis` 门面上调用任何 Redis 命令。Laravel 使用魔术方法将命令传递给 Redis 服务器，所以只需简单传递参数和 Redis 命令如下：

```
Redis::set('name', 'Taylor');
$values = Redis::lrange('names', 5, 10);
```

此外还可以使用 `command` 方法传递命令到服务器，该方法接收命令名作为第一个参数，参数值数组作为第二个参数：

```
$values = Redis::command('lrange', ['name', 5, 10]);
```

使用多个 Redis 连接

你可以通过调用 `Redis::connection` 方法获取 Redis 实例：

```
$redis = Redis::connection();
```

这将会获取默认 Redis 服务器实例，如果你没有使用服务器集群，可以传递服务器名到 `connection` 方法来获取指定 Redis 配置中定义的指定服务器：

```
$redis = Redis::connection('other');
```

管道命令

当你需要在一次操作中发送多个命令到服务器的时候应该使用管道，`pipeline` 方法接收一个参数：接收 Redis 实例的闭包。你可以将所有 Redis 命令发送到这个 Redis 实例，然后这些命令会在一次操作中被执行：

```
Redis::pipeline(function ($pipe) {
    for ($i = 0; $i < 1000; $i++) {
        $pipe->set("key:$i", $i);
    }
});
```

3、发布/订阅

Redis 还提供了调用 Redis 的 `publish` 和 `subscribe` 命令的接口。这些 Redis 命令允许你在给定“频道”监听消息，你可以从另外一个应用发布消息到这个频道，甚至使用其它编程语言，从而允许你在不同的应用/进程之间轻松通信。
首先，让我们使用 `subscribe` 方法通过 Redis 在一个频道上设置监听器。由于调用 `subscribe` 方法会开启一个常驻进程，我们将在 Artisan 命令中调用该方法：

```
<?php

namespace App\Console\Commands;

use Redis;
use Illuminate\Console\Command;

class RedisSubscribe extends Command{
    /**
     * 控制台命令名称
     *
     * @var string
     */
    protected $signature = 'redis:subscribe';

    /**
     * 控制台命令描述
     *
     * @var string
     */
    protected $description = 'Subscribe to a Redis channel';

    /**
     * 执行控制台命令
```

```
     *
     * @return mixed
     */
    public function handle()
    {
        Redis::subscribe(['test-channel'], function($message) {
            echo $message;
        });
    }
}
```

现在，我们可以使用 `publish` 发布消息到该频道：

```
Route::get('publish', function () {
    // 路由逻辑...
    Redis::publish('test-channel', json_encode(['foo' => 'bar
']));
});
```

通配符订阅

使用 `psubscribe` 方法，你可以订阅到一个通配符定义的频道，这在所有相应频道上获取所有消息时很有用。`$channel` 名将会作为第二个参数传递给提供的回调闭包：

```
Redis::psubscribe(['*'], function($message, $channel) {
    echo $message;
});

Redis::psubscribe(['users.*'], function($message, $channel) {
    echo $message;
});
```

Session

1、简介

由于 HTTP 驱动的应用是无状态的，所以我们使用 Session 来存储用户请求信息。Laravel 通过干净、统一的 API 处理后端各种 Session 驱动，目前支持的流行后端驱动包括 Memcached、Redis 和数据库。

1.1 配置

Session 配置文件位于 `config/session.php`。默认情况下，Laravel 使用的 `session` 驱动为文件驱动，这对许多应用而言是没有什么问题的。在生产环境中，你可能考虑使用 `memcached` 或者 `redis` 驱动以便获取更快的 session 性能。
`session` 驱动定义请求的 Session 数据存放在哪里，Laravel 可以处理多种类型的驱动：

- `file` – session 数据存储在 `storage/framework/sessions` 目录下；
- `cookie` – session 数据存储在经过加密的安全的 cookie 中；
- `database` – session 数据存储在数据库中
- `memcached` / `redis` – session 数据存储在 memcached/redis 中；
- `array` – session 数据存储在简单 PHP 数组中，在多个请求之间是非持久化的。

注意：数组驱动通常用于运行测试以避免 session 数据持久化。

1.2 Session 驱动预备知识

数据库

当使用 `database`session 驱动时，需要设置表包含 `session` 项，下面是该数据表的表结构声明：

```
Schema::create('sessions', function ($table) {
    $table->string('id')->unique();
    $table->text('payload');
    $table->integer('last_activity');
});
```

你可以使用 Artisan 命令 `session:table` 来生成迁移：

```
php artisan session:table
composer dump-autoload
php artisan migrate
```

Redis

在 Laravel 中使用 Redis `session` 驱动前，需要通过 Composer 安装 `predis/predis` 包。

1.3 其它 Session 相关问题

Laravel 框架内部使用 `flash` session 键，所以你不应该通过该名称添加数据项到 session。

如果你需要所有存储的 session 数据经过加密，在配置文件中设置 `encrypt` 配置为 `true`。

2、基本使用

访问 session

首先，我们来访问 session，我们可以通过 HTTP 请求访问 session 实例，可以在控制器方法中通过类型提示引入请求实例，记住，控制器方法依赖通过 Laravel 服务容器注入：

```
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller{
```

```
    /**
     * 显示指定用户的属性
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function showProfile(Request $request, $id)
    {
        $value = $request->session()->get('key');

        //
    }
}
```

从 session 中获取数据的时候，还可以传递默认值作为第二个参数到 `get` 方法，默认值在指定键在 session 中不存在时返回。如果你传递一个闭包作为默认值到 `get` 方法，该闭包会执行并返回执行结果：

```
$value = $request->session()->get('key', 'default');

$value = $request->session()->get('key', function() {
    return 'default';
});
```

如果你想要从 session 中获取所有数据，可以使用 `all` 方法：

```
$data = $request->session()->all();
```

还可以使用全局的 PHP 函数 `session` 来获取和存储 session 中的数据：

```
Route::get('home', function () {
    // 从 session 中获取数据...
    $value = session('key');

    // 存储数据到 session...
    session(['key' => 'value']);
});
```

判断 session 中是否存在指定项

`has` 方法可用于检查数据项在 session 中是否存在。如果存在的话返回 `true`：

```
if ($request->session()->has('users')) {
    //
}
```

在 session 中存储数据

获取到 session 实例后，就可以调用多个方法来与底层数据进行交互，例如，`put` 方法存储新的数据到 session 中：

```
$request->session()->put('key', 'value');
```

推送数据到数组 session

`push` 方法可用于推送数据到值为数组的 session，例如，如果 `user.teams` 键包含团队名数组，可以像这样推送新值到该数组：

```
$request->session()->push('user.teams', 'developers');
```

获取并删除数据

`pull` 方法将会从 session 获取并删除数据：

```
$value = $request->session()->pull('key', 'default');
```

从 session 中删除数据项

`forget` 方法从 session 中移除指定数据，如果你想要从 session 中移除所有数据，可以使用 `flush` 方法：

```
$request->session()->forget('key');
$request->session()->flush();
```

重新生成 Session ID

如果你需要重新生成 session ID，可以使用 `regenerate` 方法：

```
$request->session()->regenerate();
```

2.1 一次性数据

有时候你可能想要在 session 中存储只在下个请求中有效的数据，可以通过 `flash` 方法来实现。使用该方法存储的 session 数据只在随后的 HTTP 请求中有效，然后将会被删除：

```
$request->session()->flash('status', 'Task was successful!');
```

如果你需要在更多请求中保持该一次性数据，可以使用 `reflash` 方法，该方法将所有一次性数据保留到下一个请求，如果你只是想要保存特定一次性数据，可以使用 `keep` 方法：

```
$request->session()->reflash();
$request->session()->keep(['username', 'email']);
```

3、添加自定义 Session 驱动

要为 Laravel 后端 session 添加更多驱动，可以使用 Session 门面上的 `extend` 方法。可以在服务提供者的 `boot` 方法中调用该方法：

```
<?php
```

```
namespace App\Providers;

use Session;
use App\Extensions\MongoSessionStore;
use Illuminate\Support\ServiceProvider;

class SessionServiceProvider extends ServiceProvider{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Session::extend('mongo', function($app) {
            // Return implementation of SessionHandlerInterfac
e...
            return new MongoSessionStore;
        });
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

需要注意的是自定义 session 驱动需要实现 `SessionHandlerInterface` 接口，该接口包含少许我们需要实现的方法，一个 MongoDB 的实现如下：

```
<?php

namespace App\Extensions;

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) {}
}
```

由于这些方法并不像缓存的 `StoreInterface` 接口方法那样容易理解，我们接下来快速过一遍每一个方法：

- `open` 方法用于基于文件的 session 存储系统，由于 Laravel 已经有了一个 `file` session 驱动，所以在该方法中不需要放置任何代码，可以将其置为空方法。
- `close` 方法和 `open` 方法一样，也可以被忽略，对大多数驱动而言都用不到该方法。
- `read` 方法应该返回与给定$sessionId 相匹配的 session 数据的字符串版本，从驱动中获取或存储 session 数据不需要做任何序列化或其它编码，因为 Laravel 已经为我们做了序列化。
- `write` 方法应该讲给定`$data` 写到持久化存储系统相应的`$sessionId`，例如 MongoDB, Dynamo 等等。
- `destroy` 方法从持久化存储中移除 `$sessionId` 对应的数据。
- `gc` 方法销毁大于给定$lifetime 的所有 session 数据，对本身拥有过期机制的系统如 Memcached 和 Redis 而言，该方法可以留空。

session 驱动被注册之后，就可以在配置文件 `config/session.php` 中使用 `mongo` 驱动了。

Envoy Task Runner

1、简介

Laravel Envoy 为定义运行在远程主机上的通用任务提供了一套干净、最简化的语法。使用 Blade 样式语法，你可以轻松为开发设置任务，Artisan 命令，以及更多，目前，Envoy 只支持 Mac 和 Linux 操作系统。

1.1 安装

首先，使用 Composer 的 `global` 命令安装 Envoy：

```
composer global require "laravel/envoy=~1.0"
```

确保`~/.composer/vendor/bin` 目录在系统路径 PATH 中否则在终端中由于找不到 `envoy` 而无法执行该命令。

更新 Envoy

还可以使用 Composer 保持安装的 Envoy 是最新版本：

```
composer global update
```

2、编写任务

所有的 Envoy 任务都定义在项目根目录下的 `Envoy.blade.php` 文件中，下面是一个让你开始的示例：

```
@servers(['web' => 'user@192.168.1.1'])

@task('foo', ['on' => 'web'])
    ls -la
@endtask
```

正如你所看到的，`@servers` 数组定义在文件顶部，从而允许你在任务声明中使用 `on` 选项引用这些服务器，在 `@task` 声明中，应该放置将要在服务器上运行的 Bash 代码。

启动

有时候，你需要在评估 Envoy 任务之前执行一些 PHP 代码，可以在 Envoy 文件中使用 `@setup` 指令来声明变量和要执行的 PHP 代码：

```
@setup
    $now = new DateTime();
    $environment = isset($env) ? $env : "testing";
@endsetup
```

还可以使用`@include` 来引入外部 PHP 文件：

```
@include('vendor/autoload.php');
```

确认任务

如果你想要在服务器上运行给定任务之前弹出弹出提示进行确认，可以在任务声明中使用 `confirm` 指令：

```
@task('deploy', ['on' => 'web', 'confirm' => true])
    cd site
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

2.1 任务变量

如果需要的话，你可以使用命令行开关传递变量到 Envoy 文件，从而允许你自定义任务：

```
envoy run deploy --branch=master
```

你可以在任务中通过 Blade 的“echo”语法使用该选项：

```
@servers(['web' => '192.168.1.1'])

@task('deploy', ['on' => 'web'])
    cd site
```

```
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

2.2 多个服务器

你可以轻松地在多主机上运行同一个任务，首先，添加额外服务器到`@servers`声明，每个服务器应该被指配一个唯一的名字。定义好服务器后，在任务声明中简单列出所有服务器即可：

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2']])
    cd site
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

默认情况下，该任务将会依次在每个服务器上执行，这意味着，该任务在第一台服务器上运行完成后才会开始在第二台服务器运行。

平行运行

如果你想要在多个服务器上平行运行，添加`parallel`选项到任务声明：

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2'], 'parallel' => tru
e])
    cd site
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

2.3 任务宏

宏允许你使用单个命令中定义多个依次运行的任务。例如，`deploy`宏会运行 git 和 composer 任务：

```
@servers(['web' => '192.168.1.1'])

@macro('deploy')
    git
    composer
@endmacro

@task('git')
    git pull origin master
```

```
@endtask

@task('composer')
    composer install
@endtask
```

宏被定义好了之后，你就可以通过如下单个命令运行它：

```
envoy run deploy
```

3、运行任务

要从 `Envoy.blade.php` 文件中运行一个任务，需要执行 Envoy 的 `run` 命令，然后传递你要执行的任务的命令名或宏。Envoy 将会运行命令并从服务打印输出：

```
envoy run task
```

4、通知

4.1 HipChat

运行完一个任务后，可以使用 Envoy 的`@hipchat` 指令发送通知到团队的 HipChat 房间，该指令接收一个 API 令牌、房间名称、和用户名：

```
@servers(['web' => '192.168.1.1'])

@task('foo', ['on' => 'web'])
    ls -la
@endtask

@after
    @hipchat('token', 'room', 'Envoy')
@endafter
```

需要的话，你还可以传递自定义发送给 HipChat 房间的消息，所有在 Envoy 任务中有效的变量在构建消息时也有效：

```
@after
    @hipchat('token', 'room', 'Envoy', "{$task} ran in the {$en
v} environment.")
@endafter
```

4.2 Slack

除了 HipChat 之外，Envoy 还支持发送通知到 Slack。`@slack` 指令接收一个 Slack 钩子 URL、频道名称、和你要发送给该频道的消息：

```
@after
    @slack('hook', 'channel', 'message')
@endafter
```

你可以通过创建集成到 Slack 网站的 Incoming WebHooks 来获取钩子 URL，该 hook 参数是由 Incoming Webhooks Slack 集成提供的完整 webhook URL，例如：

```
https://hooks.slack.com/services/ZZZZZZZZZ/YYYYYYYYY/XXXXXXXXXXXXXXX
```

你可以提供下面两种其中之一作为频道参数：

- 发送消息到频道: `#channel`
- 发送消息到用户: `@user`

任务调度

1、简介

在以前，开发者需要为每一个需要调度的任务编写一个 Cron 条目，这是很让人头疼的事。你的任务调度不在源码控制中，你必须使用 SSH 登录到服务器然后添加这些 Cron 条目。Laravel 命令调度器允许你平滑而又富有表现力地在 Laravel 中定义命令调度，并且服务器上只需要一个 Cron 条目即可。
任务调度定义在 `app/Console/Kernel.php` 文件的 `schedule` 方法中，该方法中已经包含了一个示例。你可以自由地添加你需要的调度任务到 `Schedule` 对象。

开启调度

下面是你唯一需要添加到服务器的 Cron 条目：

```
* * * * * php /path/to/artisan schedule:run 1>> /dev/null 2>&1
```

该 Cron 将会每分钟调用 Laravel 命令调度，然后，Laravel 评估你的调度任务并运行到期的任务。

2、定义调度

你可以在 `App\Console\Kernel` 类的 `schedule` 方法中定义所有调度任务。开始之前，让我们看一个调度任务的例子，在这个例子中，我们将会在每天午夜调度一个被调用的闭包。在这个闭包中我们将会执行一个数据库查询来清空表：

```
<?php

namespace App\Console;
```

```
use DB;
use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;

class Kernel extends ConsoleKernel{
    /**
     * 应用提供的 Artisan 命令
     *
     * @var array
     */
    protected $commands = [
        'App\Console\Commands\Inspire',
    ];

    /**
     * 定义应用的命令调度
     *
     * @param  \Illuminate\Console\Scheduling\Schedule  $schedu
le
     * @return void
     */
    protected function schedule(Schedule $schedule)
    {
        $schedule->call(function () {
            DB::table('recent_users')->delete();
        })->daily();
    }
}
```

除了调度闭包调用外，还可以调度 Artisan 命令和操作系统命令。例如，可以使用 `command` 方法来调度一个 Artisan 命令：

```
$schedule->command('emails:send --force')->daily();
```

`exec` 命令可用于发送命令到操作系统：

```
$schedule->exec('node /home/forge/script.js')->daily();
```

2.1 调度常用选项

当然，你可以分配多种调度到任务：

方法	描述
`->cron('* * * * *');`	在自定义 Cron 调度上运行任务

方法	描述
`->everyMinute();`	每分钟运行一次任务
`->everyFiveMinutes();`	每五分钟运行一次任务
`->everyTenMinutes();`	每十分钟运行一次任务
`->everyThirtyMinutes();`	每三十分钟运行一次任务
`->hourly();`	每小时运行一次任务
`->daily();`	每天凌晨零点运行任务
`->dailyAt('13:00');`	每天 13:00 运行任务
`->twiceDaily(1, 13);`	每天 1:00 & 13:00 运行任务
`->weekly();`	每周运行一次任务
`->monthly();`	每月运行一次任务
`->quarterly();`	每个季度运行一次
`->yearly();`	每年运行一次

这些方法可以和额外的约束一起联合起来创建一周特定时间运行的更加细粒度的调度，例如，要每周一调度一个命令：

```
$schedule->call(function () {
    // 每周星期一 13:00 运行一次...
})->weekly()->mondays()->at('13:00');
```

下面是额外的调度约束列表：

方法	描述
`->weekdays();`	只在工作日运行任务
`->sundays();`	每个星期天运行任务
`->mondays();`	每个星期一运行任务
`->tuesdays();`	每个星期二运行任务

方法	描述
`->wednesdays();`	每个星期三运行任务
`->thursdays();`	每个星期四运行任务
`->fridays();`	每个星期五运行任务
`->saturdays();`	每个星期六运行任务
`->when(Closure);`	基于特定测试运行任务

基于测试的约束条件

`when` 方法用于限制任务在通过给定测试之后运行。换句话说，如果给定闭包返回 `true`，只要没有其它约束条件阻止任务运行，该任务就会执行：

```
$schedule->command('emails:send')->daily()->when(function () {
    return true;
});
```

reject 方法和 when 相反，如果 reject 方法返回 true，调度任务将不会执行：

```
$schedule->command('emails:send')->daily()->reject(function ()
{
    return true;
});
```

使用 when 方法链的时候，调度命令将只会执行返回 true 的 when 方法。

2.2 避免任务重叠

默认情况下，即使前一个任务仍然在运行调度任务也会运行，要避免这样的情况，可使用 `withoutOverlapping` 方法：

```
$schedule->command('emails:send')->withoutOverlapping();
```

在本例中，Artisan 命令 `emails:send` 每分钟都会运行，如果该命令没有在运行的话。如果你的任务在执行时经常大幅度的变化，那么 `withoutOverlapping` 方法就非常有用，你不必再去预测给定任务到底要消耗多长时间。

3、任务输出

Laravel 调度器为处理调度任务输出提供了多个方便的方法。首先，使用 `sendOutputTo` 方法，你可以发送输出到文件以便稍后检查：

```
$schedule->command('emails:send')
        ->daily()
        ->sendOutputTo($filePath);
```

如果你想要追加输出到给定文件，可以使用 appendOutputTo 方法：

```
$schedule->command('emails:send')
        ->daily()
        ->appendOutputTo($filePath);
```

使用 `emailOutputTo` 方法，你可以将输出发送到电子邮件，注意输出必须首先通过 `sendOutputTo` 方法发送到文件。还有，使用电子邮件发送任务输出之前，应该配置 Laravel 的电子邮件服务：

```
$schedule->command('foo')
        ->daily()
        ->sendOutputTo($filePath)
        ->emailOutputTo('foo@example.com');
```

注意：`emailOutputTo` 和 `sendOutputTo` 方法只对 `command` 方法有效，不支持 `call` 方法。

4、任务钩子

使用 `before` 和 `after` 方法，你可以指定在调度任务完成之前和之后要执行的代码：

```
$schedule->command('emails:send')
        ->daily()
        ->before(function () {
            // Task is about to start...
        })
        ->after(function () {
            // Task is complete...
        });
```

ping URL

使用 `pingBefore` 和 `thenPing` 方法，调度器可以在任务完成之前和之后自动 ping 给定的 URL。该方法在通知外部服务时很有用，例如 Laravel Envoyer，在调度任务开始或完成的时候：

```
$schedule->command('emails:send')
        ->daily()
        ->pingBefore($url)
        ->thenPing($url);
```

使用 `pingBefore($url)`或 `thenPing($url)`特性需要安装 HTTP 库 Guzzle，可以在 `composer.json` 文件中添加如下行来安装 Guzzle 到项目：

```
"guzzlehttp/guzzle": "~5.3|~6.0"
```

测试

1、简介

Laravel 植根于测试，实际上，内置使用 PHPUnit 对测试提供支持是即开即用的，并且 `phpunit.xml` 文件已经为应用设置好了。框架还提供了方便的辅助方法允许你对应用进行富有表现力的测试。
`tests` 目录中提供了一个 `ExampleTest.php` 文件，安装完新的 Laravel 应用后，只需简单在命令行运行 `phpunit` 来运行测试。

1.1 测试环境

运行测试的时候，Laravel 自动设置配置环境为 `testing`。Laravel 在测试时自动配置 session 和 cache 驱动为数组驱动，这意味着测试时不会持久化存储 session 和 cache。
如果需要的话你也可以创建其它测试环境配置。`testing` 环境变量可以在 `phpunit.xml` 文件中配置。

1.2 定义&运行测试

要创建一个新的测试用例，可以使用如下 Artisan 命令：

```
php artisan make:test UserTest
```

该命令将会在 `tests` 目录下生成一个新的 `UserTest` 类。然后你可以使用 PHPUnit 定义测试方法。要运行测试，简单从终端执行 `phpunit` 命令即可：

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class UserTest extends TestCase{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function testExample()
    {
        $this->assertTrue(true);
    }
}
```

注意：如果你在测试类中定义自己的 `setUp` 方法，确保在其中调用 `parent::setUp`。

2、应用测试

Laravel 为生成 HTTP 请求、测试输出、以及填充表单提供了平滑的 API。举个例子，我们看下 `tests` 目录下包含的 `ExampleTest.php` 文件：

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase{
    /**
     * 基本功能测试示例
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->visit('/')
             ->see('Laravel 5')
             ->dontSee('Rails');
    }
}
```

`visit` 方法生成了一个 GET 请求，`see` 方法对我们从应用返回响应中应该看到的给定文本进行断言。`dontSee` 方法对给定文本没有从应用响应中返回进行断言。在 Laravel 中这是最基本的有效应用测试。

2.1 与应用交互

当然，除了对响应文本进行断言之外还有做更多测试，让我们看一些点击链接和填充表单的例子：

点击链接

在本测试中，我们将为应用生成请求，在返回的响应中“点击”链接，然后对访问 URI 进行断言。例如，假定响应中有一个“关于我们”的链接：

```
<a href="/about-us">About Us</a>
```

现在，让我们编写一个测试点击链接并断言用户访问页面是否正确：

```
public function testBasicExample(){
    $this->visit('/')
         ->click('About Us')
```

```
        ->seePageIs('/about-us');
}
```

处理表单

Laravel 还为处理表单提供了多个方法。`type`, `select`, `check`, `attach`, 和 `press` 方法允许你与所有表单输入进行交互。例如，我们假设这个表单存在于应用注册页面：

```
<form action="/register" method="POST">
    {!! csrf_field() !!}

    <div>
        Name: <input type="text" name="name">
    </div>

    <div>
        <input type="checkbox" value="yes" name="terms"> Accept
Terms
    </div>

    <div>
        <input type="submit" value="Register">
    </div>
</form>
```

我们可以编写测试完成表单并检查结果：

```
public function testNewUserRegistration(){
    $this->visit('/register')
         ->type('Taylor', 'name')
         ->check('terms')
         ->press('Register')
         ->seePageIs('/dashboard');
}
```

当然，如果你的表单包含其他输入比如单选按钮或下拉列表，也可以轻松填写这些字段类型。这里是所有表单操作方法列表：

方法	描述
`$this->type($text, $elementName)`	“Type” 文本到给定字段
`$this->select($value, $elementName)`	“Select” 单选框或下拉列表
`$this->check($elementName)`	“Check” 复选框

方法	描述
`$this->attach($pathToFile, $elementName)`	“Attach” 文件到表单
`$this->press($buttonTextOrElementName)`	“Press” 给定文本或 name 的按钮
`$this->uncheck($elementName)`	“Uncheck”复选框

处理附件

如果表单包含 `file` 输入类型，可以使用 `attach` 方法添加文件到表单：

```
public function testPhotoCanBeUploaded(){
    $this->visit('/upload')
         ->name('File Name', 'name')
         ->attach($absolutePathToFile, 'photo')
         ->press('Upload')
         ->see('Upload Successful!');
}
```

2.2 测试 JSON API

Laravel 还提供多个帮助函数用于测试 JSON API 及其响应。例如，`get`, `post`, `put`, `patch`, 和 `delete` 方法用于通过多种 HTTP 请求方式发出请求。你还可以轻松传递数据和头到这些方法。作为开始，我们编写测试来生成 POST 请求到 `/user` 并断言返回的数据是否是 JSON 格式：

```
<?php

class ExampleTest extends TestCase{
    /**
     * 基本功能测试示例
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->post('/user', ['name' => 'Sally'])
             ->seeJson([
                 'created' => true,
             ]);
    }
}
```

`seeJson` 方法将给定数组转化为 JSON，然后验证应用返回的整个 JSON 响应中的 JSON 片段。因此，如果在 JSON 响应中有其他属性，只要给定片段存在的话测试依然会通过。

验证 JSON 值匹配

如果你想要验证给定数组和应用返回的 JSON 能够精确匹配，使用 `seeJsonEquals` 方法：

```
<?php

class ExampleTest extends TestCase{
    /**
     * 基本功能测试示例
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->post('/user', ['name' => 'Sally'])
             ->seeJsonEquals([
                 'created' => true,
             ]);
    }
}
```

验证 JSON 数据结构匹配

还可以验证 JSON 响应是否与指定数据结构匹配，我们使用 `seeJsonStructure` 方法来实现这一功能：

```
<?php

class ExampleTest extends TestCase{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->get('/user/1')
             ->seeJsonStructure([
                 'name',
                 'pet' => [
                     'name', 'age'
                 ]
             ]);
    }
}
```

上面的例子演示了期望获取一个包含 `name` 和嵌套 `pet` 对象（该对象包含 `name` 和 `age` 属性）的 JSON 数据。如果 JSON 响应中包含其它额外键 `seeJsonStructure` 也不会失败，例如，如果 `pet` 对象包含 `weight` 属性测试仍将通过。
你可以使用*来断言返回 JSON 结构包含一个列表，该列表中的每个数据项都包含至少如下示例中列出的属性：

```
<?php

class ExampleTest extends TestCase{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        // Assert that each user in the list has at least an id,
 name and email attribute.
        $this->get('/users')
             ->seeJsonStructure([
                 '*' => [
                     'id', 'name', 'email'
                 ]
             ]);
    }
}
```

你还可以使用嵌套的*，在这种场景中，我们可以断言 JSON 响应中的每个用户都包含一个给定属性集合，而且每个用户的每个 `pet` 都包含给定属性集合：

```
$this->get('/users')
     ->seeJsonStructure([
         '*' => [
             'id', 'name', 'email', `pets` => [
                 '*' => [
                     'name', 'age'
                 ]
             ]
         ]
     ]);
```

2.3 Session/认证

Laravel 提供了多个辅助函数用于在测试期间处理 Session，首先，可以使用 `withSession` 方法设置 session 值到给定数组。这在测试请求前获取 session 数据时很有用：

```
<?php

class ExampleTest extends TestCase{
    public function testApplication()
    {
        $this->withSession(['foo' => 'bar'])
             ->visit('/');
    }
}
```

当然，session 的通常用于操作用户状态，例如认证用户。辅助函数 `actingAs` 提供了认证给定用户为当前用户的简单方法，例如，我们使用模型工厂生成和认证用户：

```
<?php

class ExampleTest extends TestCase{
    public function testApplication()
    {
        $user = factory('App\User')->create();

        $this->actingAs($user)
             ->withSession(['foo' => 'bar'])
             ->visit('/')
             ->see('Hello, '.$user->name);
    }
}
```

还可以通过传递 `guard` 名称作为 `actingAs` 函数的第二个参数的方式来指定使用哪个 `guard` 来认证给定用户：

```
$this->actingAs($user, 'backend')
```

2.4 禁止中间件

测试应用时，为某些测试禁止中间件很方便。这种机制允许你将路由和控制器与中间件孤立开来做测试，Laravel 包含了一个简单的 `WithoutMiddleware` trait，可以使用该 trait 自动在测试类中禁止所有中间件：

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase{
    use WithoutMiddleware;
    //
```

```
}
```

如果你只想在某些方法中禁止中间件，可以在测试方法中调用 `withoutMiddleware` 方法：

```
<?php

class ExampleTest extends TestCase{
    /**
     * 基本功能测试示例
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->withoutMiddleware();

        $this->visit('/')
             ->see('Laravel 5');
    }
}
```

2.5 自定义 HTTP 请求

如果你想要在应用中生成自定义 HTTP 请求并获取完整的 `Illuminate\Http\Response` 对象，可以使用 `call` 方法：

```
public function testApplication(){
    $response = $this->call('GET', '/');
    $this->assertEquals(200, $response->status());
}
```

如果你要生成 `POST`, `PUT`, 或者 `PATCH` 请求可以在请求中传入输入数据数组，在路由或控制器中可以通过 Request 实例访问请求数据：

```
$response = $this->call('POST', '/user', ['name' => 'Taylor']);
```

2.6 PHPUnit 断言方法

Laravel 为 PHPUnit 测试提供了额外的断言方法：

方法	描述
`->assertResponseOk();`	断言客户端响应状态码是否为 200
`->assertResponseStatus($code);`	断言客户端响应状态码是否是给定
`->assertViewHas($key, $value = null);`	断言响应视图是否包含给定的绑定

方法	描述
	段
`->assertViewHasAll(array $bindings);`	断言视图是否包含给定绑定数据列
`->assertViewMissing($key);`	断言响应视图缺失绑定数据片段
`->assertRedirectedTo($uri, $with = []);`	断言客户端是否重定向到给定 URI
`->assertRedirectedToRoute($name, $parameters = [], $with = []);`	断言客户端是否重定向到给定路由
`->assertRedirectedToAction($name, $parameters = [], $with = []);`	断言客户端是否重定向到给定 actio
`->assertSessionHas($key, $value = null);`	断言 session 是否包含给定值
`->assertSessionHasAll(array $bindings);`	断言 session 是否保护眼给定值列表
`->assertSessionHasErrors($bindings = [], $format = null);`	断言 session 是否包含错误绑定
`->assertHasOldInput();`	断言 sessio 包含上次输入数据

3、处理数据库

Laravel 还提供了多种有用的工具让测试数据库驱动的应用更加简单。首先，你可以使用帮助函数 `seeInDatabase` 来断言数据库中的数据是否和给定数据集合匹配。例如，如果你想要通过 email 值为 `sally@example.com` 的条件去数据表 `users` 查询是否存在该记录 ，我们可以这样做：

```
public function testDatabase(){
    // 调用应用...
    $this->seeInDatabase('users', ['email' => 'sally@foo.com
']);
}
```

当然，`seeInDatabase` 方法和其它类似辅助方法都是为了方便起见进行的封装，你也可以使用其它 PHPUnit 内置的断言方法来进行测试。

3.1 每次测试后重置数据库

每次测试后重置数据库通常很有用，这样的话上次测试的数据不会影响下一次测试。

使用迁移

一种方式是每次测试后回滚数据库并在下次测试前重新迁移。Laravel 提供了一个简单的 `DatabaseMigrations` trait 来自动为你处理。在测试类上简单使用该 trait 如下：

```
<?php
```

```
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase{
    use DatabaseMigrations;

    /**
     * 基本功能测试示例
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->visit('/')
             ->see('Laravel 5');
    }
}
```

使用事务

另一种方式是将每一个测试用例包裹到一个数据库事务中，Laravel 提供了方便的 `DatabaseTransactions` trait 自动为你处理：

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase{
    use DatabaseTransactions;

    /**
     * 基本功能测试示例
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->visit('/')
             ->see('Laravel 5');
    }
}
```

注意：该 trait 只在事务中封装默认数据库连接。

3.2 模型工厂

测试时，通常需要在执行测试前插入新数据到数据库。在创建测试数据时，Laravel 允许你使用“factories”为每个 Eloquent 模型定义默认的属性值集合，而不用手动为每一列指定值。作为开始，我们看一下 `database/factories/ModelFactory.php` 文件，该文件包含了一个工厂定义：

```
$factory->define(App\User::class, function (Faker\Generator $faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->email,
        'password' => bcrypt(str_random(10)),
        'remember_token' => str_random(10),
    ];
});
```

在闭包中，作为工厂定义，我们返回该模型上所有属性默认测试值。该闭包接收 PHP 库 Faker 实例，从而允许你方便地为测试生成多种类型的随机数据。
当然，你可以添加更多工厂到 `ModelFactory.php` 文件。

多个工厂类型

有时候你可能想要为同一个 Eloquent 模型类生成多个工厂，例如，除了正常用户外可能你想要为“管理员”用户生成一个工厂，你可以使用 `defineAs` 方法定义这些工厂：

```
$factory->defineAs(App\User::class, 'admin', function ($faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->email,
        'password' => str_random(10),
        'remember_token' => str_random(10),
        'admin' => true,
    ];
});
```

你可以使用 `raw` 方法获取基本属性而不用重复基本用户工厂中的所有属性，获取这些属性后，只需将你要求的额外值增补进去即可：

```
$factory->defineAs(App\User::class, 'admin', function ($faker) use ($factory) {
    $user = $factory->raw(App\User::class);
    return array_merge($user, ['admin' => true]);
});
```

在测试中使用工厂

定义好工厂后，可以在测试或数据库填充文件中通过全局的 `factory` 方法使用它们来生成模型实例，所以，让我们看一些生成模型的例子，首先，我们使用 `make` 方法，该方法创建模型但不将其保存到数据库：

```
public function testDatabase(){
    $user = factory(App\User::class)->make();
    // 用户模型测试...
}
```

如果你想要覆盖模型的一些默认值，可以传递数组值到 `make` 方法。只有指定值被替换，其他数据保持不变：

```
$user = factory(App\User::class)->make([
    'name' => 'Abigail',
]);
```

还可以创建多个模型集合或者创建给定类型的集合：

```
// 创建 3 个 App\User 实例...
$users = factory(App\User::class, 3)->make();
// 创建 1 个 App\User "admin" 实例...
$user = factory(App\User::class, 'admin')->make();
// 创建 3 个 App\User "admin" 实例...
$users = factory(App\User::class, 'admin', 3)->make();
```

持久化工厂模型

`create` 方法不仅能创建模型实例，还可以使用 Eloquent 的 `save` 方法将它们保存到数据库：

```
public function testDatabase(){
    $user = factory(App\User::class)->create();
    //用户模型测试...
}
```

你仍然可以通过传递数组到 `create` 方法覆盖模型上的属性：

```
$user = factory(App\User::class)->create([
    'name' => 'Abigail',
]);
```

添加关联关系到模型

你甚至可以持久化多个模型到数据库。在本例中，我们添加一个关联到创建的模型，使用 `create` 方法创建多个模型的时候，会返回一个 Eloquent 集合实例，从而允许你使用集合提供的所有便利方法，例如 `each`：

```
$users = factory(App\User::class, 3)
           ->create()
```

```
            ->each(function($u) {
                $u->posts()->save(factory(App\Post::class)->make
());
            });
```

4、模拟

4.1 模拟事件

如果你在重度使用 Laravel 的时间系统，可能想要在测试时模拟特定事件。例如，如果你在测试用户注册，你可能不想所有 `UserRegistered` 的时间处理器都被触发，因为这可能会发送欢迎邮件，等等。

Laravel 提供可一个方便的 `expectsEvents` 方法来验证期望的事件被触发，但同时阻止该事件的其它处理器运行：

```
<?php

class ExampleTest extends TestCase{
    public function testUserRegistration()
    {
        $this->expectsEvents(App\Events\UserRegistered::class);
        // 测试用户注册代码...
    }
}
```

可以使用 `doesntExpectEvents` 方法来验证给定事件没有被触发：

```
<?php

class ExampleTest extends TestCase{
    public function testPodcastPurchase()
    {
        $this->expectsEvents(App\Events\PodcastWasPurchased::cl
ass);

        $this->doesntExpectEvents(App\Events\PaymentWasDecline
d::class);

        // Test purchasing podcast...
    }
}
```

如果你想要阻止所有事件运行，可以使用 `withoutEvents` 方法：

```
<?php

class ExampleTest extends TestCase{
```

```
    public function testUserRegistration()
    {
        $this->withoutEvents();
        // 测试用户注册代码...
    }
}
```

4.2 模拟队列任务

有时候，你可能想要在请求时简单测试控制器分发的指定任务，这允许你孤立的测试路由/控制器——将其从任务逻辑中分离出去，当然，接下来你可以在一个独立测试类中测试任务本身。

Laravel 提供了一个方便的 `expectsJobs` 方法来验证期望的任务被分发，但该任务本身不会被测试：

```
<?php

class ExampleTest extends TestCase{
    public function testPurchasePodcast()
    {
        $this->expectsJobs(App\Jobs\PurchasePodcast::class);
        // 测试购买播客代码...
    }
}
```

注意：这个方法只检查通过 `DispatchesJobs` trait 分发方法分发的任务，并不检查直接通过 `Queue::push` 分发的任务。

4.3 模拟门面

测试的时候，你可能经常想要模拟 Laravel 门面的调用，例如，看看下面的控制器动作：

```
<?php

namespace App\Http\Controllers;

use Cache;
use Illuminate\Routing\Controller;

class UserController extends Controller{
    /**
     * 显示应用用户列表
     *
     * @return Response
     */
    public function index()
    {
```

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

        //
    }
}
```

我们可以通过使用 `shouldReceive` 方法模拟 `Cache` 门面的调用，该方法返回一个 Mockery 模拟的实例，由于门面通过 Laravel 服务容器解析和管理，它们比通常的静态类更具有可测试性。例如，我们来模拟 `Cache` 门面的调用：

```
<?php

class FooTest extends TestCase{
    public function testGetIndex()
    {
        Cache::shouldReceive('get')
                    ->once()
                    ->with('key')
                    ->andReturn('value');

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

注意：不要模拟 `Request` 门面，取而代之地，在测试时传递输入到 HTTP 帮助函数如 `call` 和 `post`。

验证

1、简介

Laravel 提供了多种方法来验证应用输入数据。默认情况下，Laravel 的控制器基类使用 `ValidatesRequests` trait，该 trait 提供了便利的方法通过各种功能强大的验证规则来验证输入的 HTTP 请求。

2、快速入门

要掌握 Laravel 强大的验证特性，让我们先看一个完整的验证表单并返回错误信息给用户的例子。

2.1 定义路由

首先，我们假定在 `app/Http/routes.php` 文件中包含如下路由：

```
// 显示创建博客文章表单...
```

```
Route::get('post/create', 'PostController@create');
// 存储新的博客文章...
Route::post('post', 'PostController@store');
```

当然，GET 路由为用户显示了一个创建新的博客文章的表单，POST 路由将新的博客文章存储到数据库。

2.2 创建控制器

接下来，让我们看一个处理这些路由的简单控制器示例。我们先将 `store` 方法留空：

```
<?php

namespace App\Http\Controllers;

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

class PostController extends Controller{
    /**
     * 显示创建新的博客文章的表单
     *
     * @return Response
     */
    public function create()
    {
        return view('post.create');
    }

    /**
     * 存储新的博客文章
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        // 验证并存储博客文章...
    }
}
```

2.3 编写验证逻辑

现在我们准备用验证新博客文章输入的逻辑填充 `store` 方法。如果你检查应用的控制器基类（`App\Http\Controllers\Controller`），你会发现该类使用了 `ValidatesRequests` trait，这个 trait 在所有控制器中提供了一个便利的 `validate` 方法。

validate 方法接收一个 HTTP 请求输入数据和验证规则，如果验证规则通过，代码将会继续往下执行；然而，如果验证失败，将会抛出一个异常，相应的错误响应也会自动发送给用户。在一个传统的 HTTP 请求案例中，将会生成一个重定向响应，如果是 AJAX 请求则会返回一个 JSON 响应。

要更好的理解 validate 方法，让我们回到 store 方法：

```
/**
 * 存储博客文章
 *
 * @param  Request  $request
 * @return Response
 */
public function store(Request $request){
    $this->validate($request, [
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ]);

    // 验证通过，存储到数据库...
}
```

正如你所看到的，我们只是传递输入的 HTTP 请求和期望的验证规则到 validate 方法，在强调一次，如果验证失败，相应的响应会自动生成。如果验证通过，控制器将会继续正常执行。

首次验证失败后中止后续规则验证

有时候你可能想要在首次验证失败后停止检查属性其它验证规则，要实现这个功能，可以在属性中分配 bail 规则：

```
$this->validate($request, [
    'title' => 'bail|required|unique:posts|max:255',
    'body' => 'required',]
);
```

在这个例子中，如果 title 属性上的 required 规则验证失败，则不会检查 unique 规则，规则会按照分配顺序依次进行验证。

嵌套属性注意事项

如果 HTTP 请求中包含“嵌套”参数，可以使用“.”在验证规则中指定它们：

```
$this->validate($request, [
    'title' => 'required|unique:posts|max:255',
    'author.name' => 'required',
    'author.description' => 'required',
]);
```

2.4 显示验证错误信息

那么，如果请求输入参数没有通过给定验证规则怎么办？正如前面所提到的，Laravel 将会自动将用户重定向回上一个位置。此外，所有验证错误信息会自动一次性存放到 session。注意我们并没有在 GET 路由中明确绑定错误信息到视图。这是因为 Laravel 总是从 session 数据中检查错误信息，而且如果有的话会自动将其绑定到视图。所以，值得注意的是每次请求的所有视图中总是存在一个`$errors`变量，从而允许你在视图中方便而又安全地使用。`$errors`变量是的一个`Illuminate\Support\MessageBag`实例。想要了解更多关于该对象的信息，查看其文档。

注意：`$errors`变量会通过`web`中间件组中的`Illuminate\View\Middleware\ShareErrorsFromSession`中间件绑定到视图，如果使用了该中间件，那么`$errors`变量在视图中总是有效，从而方便你随时使用。

所以，在我们的例子中，验证失败的话用户将会被重定向到控制器的`create`方法，从而允许我们在视图中显示错误信息：

```
<!-- /resources/views/post/create.blade.php -->

<h1>Create Post</h1>

@if (count($errors) > 0)
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

<!-- Create Post Form -->
```

自定义错误格式

如果你想要自定义保存在 session 中的验证错误信息的格式，需要在控制器基类中重写`formatValidationErrors`方法（不要忘了在该控制器类的顶部导入`Illuminate\Contracts\Validation\Validator`类）：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Foundation\Bus\DispatchesJobs;
use Illuminate\Contracts\Validation\Validator;
use Illuminate\Routing\Controller as BaseController;
use Illuminate\Foundation\Validation\ValidatesRequests;
```

```
abstract class Controller extends BaseController{
    use DispatchesJobs, ValidatesRequests;

    /**
     * {@inheritdoc}
     */
    protected function formatValidationErrors(Validator $valida
tor)
    {
        return $validator->errors()->all();
    }
}
```

2.5 AJAX 请求&验证

在这个例子中，我们使用传统的表单来发送数据到应用。然而，很多应用使用 AJAX 请求。在 AJAX 请求中使用 `validate` 方法时，Laravel 不会生成重定向响应。取而代之的，Laravel 生成一个包含验证错误信息的 JSON 响应。该 JSON 响应会带上一个 HTTP 状态码 `422`。

2.6 验证数组输入

验证表单数组输入字段在 Laravel 5.2 中不再是件痛苦的事情，例如，要验证给定数组输入中每个 email 是否是唯一的，可以这么做：

```
$validator = Validator::make($request->all(), [
    'person.*.email' => 'email|unique:users']
);
```

类似地，在语言文件中你也可以使用`*`字符指定验证消息，从而可以使用单个消息定义基于数组字段的验证规则：

```
'custom' => [
    'person.*.email' => [
        'unique' => 'Each person must have a unique e-mail addre
ss',
    ]
],
```

3、其它验证方法

3.1 手动创建验证器

如果你不想使用 `ValidatesRequests` trait 的 `validate` 方法，可以使用 `Validator` 门面手动创建一个验证器实例，该门面上的 `make` 方法用于生成一个新的验证器实例：

```
<?php

namespace App\Http\Controllers;

use Validator;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PostController extends Controller{
    /**
     * 存储新的博客文章
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        $validator = Validator::make($request->all(), [
            'title' => 'required|unique:posts|max:255',
            'body' => 'required',
        ]);

        if ($validator->fails()) {
            return redirect('post/create')
                        ->withErrors($validator)
                        ->withInput();
        }

        // 存储博客文章...
    }
}
```

传递给 `make` 方法的第一个参数是需要验证的数据，第二个参数是要应用到数据上的验证规则。

检查请求是够通过验证后，可以使用 `withErrors` 方法将错误数据一次性存放到 session，使用该方法时，`$errors` 变量重定向后自动在视图间共享，从而允许你轻松将其显示给用户，`withErrors` 方法接收一个验证器、或者一个 MessageBag，又或者一个 PHP 数组。

命名错误包

如果你在单个页面上有多个表单，可能需要命名 MessageBag，从而允许你为指定表单获取错误信息。只需要传递名称作为第二个参数给 `withErrors` 即可：

```
return redirect('register')
            ->withErrors($validator, 'login');
```

然后你就可以从`$errors` 变量中访问命名的 MessageBag 实例：

```
{{ $errors->login->first('email') }}
```

验证钩子之后

验证器允许你在验证完成后添加回调，这种机制允许你轻松执行更多验证，甚至添加更多错误信息到消息集合。使用验证器实例上的 `after` 方法即可：

```
$validator = Validator::make(...);

$validator->after(function($validator) {
    if ($this->somethingElseIsInvalid()) {
        $validator->errors()->add('field', 'Something is wrong w
ith this field!');
    }
});

if ($validator->fails()) {
    //
}
```

3.2 表单请求验证

对于更复杂的验证场景，你可能想要创建一个“表单请求”。表单请求是包含验证逻辑的自定义请求类，要创建表单验证类，可以使用 Artisan 命令 `make:request`：

```
php artisan make:request StoreBlogPostRequest
```

生成的类位于 `app/Http/Requests` 目录下，接下来我们添加少许验证规则到 `rules` 方法：

```
/**
 * 获取应用到请求的验证规则
 *
 * @return array
 */
public function rules(){
    return [
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ];
```

```
}
```

那么，验证规则如何生效呢？你所要做的就是在控制器方法中类型提示该请求。表单输入请求会在控制器方法被调用之前被验证，这就是说你不需要将控制器和验证逻辑杂糅在一起：

```
/**
 * 存储输入的博客文章
 *
 * @param  StoreBlogPostRequest  $request
 * @return Response
 */
public function store(StoreBlogPostRequest $request){
    // The incoming request is valid...
}
```

如果验证失败，重定向响应会被生成并将用户退回上一个位置，错误信息也会被一次性存储到 session 以便在视图中显示。如果是 AJAX 请求，带 `422` 状态码的 HTTP 响应将会返回给用户，该响应数据中还包含了 JSON 格式的验证错误信息。

认证表单请求

表单请求类还包含了一个 `authorize` 方法，你可以检查认证用户是否有资格更新指定资源。例如，如果用户尝试更新一个博客评论，那么他是否是评论的所有者呢？举个例子：

```
/**
 * 判断请求用户是否经过认证
 *
 * @return bool
 */
public function authorize(){
    $commentId = $this->route('comment');
    return Comment::where('id', $commentId)
                ->where('user_id', Auth::id())->exists();
}
```

注意上面这个例子中对 `route` 方法的调用。该方法赋予用户访问被调用路由 URI 参数的权限，比如下面这个例子中的`{comment}`参数：

```
Route::post('comment/{comment}');
```

如果 `authorize` 方法返回 `false`，一个包含 `403` 状态码的 HTTP 响应会自动返回而且控制器方法将不会被执行。

如果你计划在应用的其他部分包含认证逻辑，只需在 `authorize` 方法中简单返回 `true` 即可：

```
/**
 * 判断请求用户是否经过认证
```

```
     *
     * @return bool
     */
    public function authorize(){
        return true;
    }
```

自定义错误格式

如果你想要自定义验证失败时一次性存储到 session 中验证错误信息的格式，重写请求基类（`App\Http\Requests\Request`）中的 `formatErrors` 方法即可。不要忘记在文件顶部导入 `Illuminate\Contracts\Validation\Validator` 类：

```
    /**
     * {@inheritdoc}
     */
    protected function formatErrors(Validator $validator){
        return $validator->errors()->all();
    }
```

自定义错误消息

你可以通过重写 messages 方法自定义表单请求使用的错误消息，该方法应该返回属性/规则对数组及其对应错误消息：

```
    /**
     * Get the error messages for the defined validation rules.
     *
     * @return array
     */
    public function messages(){
        return [
            'title.required' => 'A title is required',
            'body.required'  => 'A message is required',
        ];
    }
```

4、处理错误信息

调用 Validator 实例上的 `errors` 方法之后，将会获取一个 `Illuminate\Support\MessageBag` 实例，该实例中包含了多种处理错误信息的便利方法。

获取某字段的第一条错误信息

要获取指定字段的第一条错误信息，可以使用 `first` 方法：

```
$messages = $validator->errors();
```

```
echo $messages->first('email');
```

获取指定字段的所有错误信息

如果你想要简单获取指定字段的所有错误信息数组，使用 `get` 方法：

```
foreach ($messages->get('email') as $message) {
    //
}
```

获取所有字段的所有错误信息

要获取所有字段的所有错误信息，可以使用 `all` 方法：

```
foreach ($messages->all() as $message) {
    //
}
```

判断消息中是否存在某字段的错误信息

```
if ($messages->has('email')) {
    //
}
```

获取指定格式的错误信息

```
echo $messages->first('email', '<p>:message</p>');
```

获取指定格式的所有错误信息

```
foreach ($messages->all('<li>:message</li>') as $message) {
    //
}
```

4.1 自定义错误信息

如果需要的话，你可以使用自定义错误信息替代默认的，有多种方法来指定自定义信息。首先，你可以传递自定义信息作为第三方参数给 `Validator::make` 方法：

```
$messages = [
    'required' => 'The :attribute field is required.',
];

$validator = Validator::make($input, $rules, $messages);
```

在本例中，`:attribute` 占位符将会被验证时实际的字段名替换，你还可以在验证消息中使用其他占位符，例如：

```
$messages = [
    'same'    => 'The :attribute and :other must match.',
```

```
    'size'    => 'The :attribute must be exactly :size.',
    'between' => 'The :attribute must be between :min - :max.',
    'in'      => 'The :attribute must be one of the following ty
pes: :values',
];
```

为给定属性指定自定义信息

有时候你可能只想为特定字段指定自定义错误信息，可以通过”.”来实现，首先指定属性名，然后是规则：

```
$messages = [
    'email.required' => 'We need to know your e-mail address!',
];
```

在语言文件中指定自定义消息

在很多案例中，你可能想要在语言文件中指定属性特定自定义消息而不是将它们直接传递给 `Validator`。要实现这个，添加消息到 `resources/lang/xx/validation.php` 语言文件的 custom 数组：

```
'custom' => [
    'email' => [
        'required' => 'We need to know your e-mail address!',
    ],
],
```

5、验证规则大全

下面是有效规则及其函数列表：

- 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
- JSON
- Max
- MIME Types (File)
- Min
- Not In
- Numeric
- Regular Expression
- Required
- Required If
- Required Unless
- Required With
- Required With All
- Required Without
- Required Without All
- Same
- Size
- String
- Timezone
- Unique (Database)
- URL

accepted

在验证中该字段的值必须是 `yes`、`on`、`1` 或 `true`，这在“同意服务协议”时很有用。

active_url

该字段必须是一个基于 PHP 函数 `checkdnsrr` 的有效 URL

after:date

该字段必须是给定日期后的一个值，日期将会通过 PHP 函数 `strtotime` 传递：

```
'start_date' => 'required|date|after:tomorrow'
```

你可以指定另外一个比较字段而不是使用 strtotime 验证传递的日期字符串：

```
'finish_date' => 'required|date|after:start_date'
```

alpha

该字段必须是字母

alpha_dash

该字段可以包含字母和数字，以及破折号和下划线

alpha_num

该字段必须是字母或数字

array

该字段必须是 PHP 数组

before:date

验证字段必须是指定日期之前的一个数值，该日期将会传递给 PHP `strtotime` 函数。

between:min,max

验证字段尺寸在给定的最小值和最大值之间，字符串、数值和文件都可以使用该规则

boolean

验证字段必须可以被转化为 `boolean`，接收 `true`, `false`, `1`,`0`, `"1"`，和 `"0"`等输入。

confirmed

验证字段必须有一个匹配字段 `foo_confirmation`，例如，如果验证字段是 `password`，必须输入一个与之匹配的 `password_confirmation` 字段

date

验证字段必须是一个基于 PHP `strtotime` 函数的有效日期

date_format:format

验证字段必须匹配指定格式，该格式将使用 PHP 函数 `date_parse_from_format` 进行验证。你应该在验证字段时使用 `date` 或 `date_format`

different:field

验证字段必须是一个和指定字段不同的值

digits:value

验证字段必须是数字且长度为 `value` 指定的值

digits_between:min,max

验证字段数值长度必须介于最小值和最大值之间

email

验证字段必须是格式化的电子邮件地址

exists:table.column

验证字段必须存在于指定数据表

基本使用：

```
'state' => 'exists:states'
```

指定自定义列名：

```
'state' => 'exists:states,abbreviation'
```

还可以添加更多查询条件到 `where` 查询子句：

```
'email' => 'exists:staff,email,account_id,1'
```

传递 NULL 作为 `where` 子句的值将会判断数据库值是否为 NULL：

```
'email' => 'exists:staff,email,deleted_at,NULL'
```

image

验证文件必须是图片（jpeg、png、bmp、gif 或者 svg）

in:foo,bar…

验证字段值必须在给定的列表中

integer

验证字段必须是整型

ip

验证字段必须是 IP 地址

JSON

验证字段必须是有效的 JSON 字符串

max:value

验证字段必须小于等于最大值，和字符串、数值、文件字段的 size 规则一起使用

mimes:foo,bar,…

验证文件的 MIMIE 类型必须是该规则列出的扩展类型中的一个

MIMIE 规则的基本使用：

```
'photo' => 'mimes:jpeg,bmp,png'
```

尽管你只需要指定扩展，该规则实际上验证的是通过读取文件内容获取到的文件 MIME 类型。
完整的 MIME 类型列表及其相应的扩展可以在这里找到：
http://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types

min:value

验证字段的最小值，和字符串、数值、文件字段的 size 规则一起使用

not_in:foo,bar,…

验证字段值不在给定列表中

numeric

验证字段必须是数值

regex:pattern

验证字段必须匹配给定正则表达式

注意：使用 `regex` 模式时，规则必须放在数组中，而不能使用管道分隔符，尤其是正则表达式中使用管道符号时。

required

输入字段值不能为空，以下情况字段值都为空：

- 值为 null
- 值是空字符串
- 值是空数组或者空的 Coutable 对象
- 值是上传文件但路径为空

required_if:anotherfield,value,…

验证字段在另一个字段等于指定值 value 时是必须的

required_unless:anotherfield,value,…

除了 anotherfield 字段等于 value，验证字段不能空

required_with:foo,bar,...

验证字段只有在任一其它指定字段存在的话才是必须的

required_with_all:foo,bar,...

验证字段只有在所有指定字段存在的情况下才是必须的

required_without:foo,bar,...

验证字段只有当任一指定字段不存在的情况下才是必须的

required_without_all:foo,bar,...

验证字段只有当所有指定字段不存在的情况下才是必须的

same:field

给定字段和验证字段必须匹配

size:value

验证字段必须有和给定值相 value 匹配的尺寸，对字符串而言，`value` 是相应的字符数目；对数值而言，`value` 是给定整型值；对文件而言，`value` 是相应的文件字节数

string

验证字段必须是字符串

timezone

验证字符必须是基于 PHP 函数 `timezone_identifiers_list` 的有效时区标识

unique:table,column,except,idColumn

验证字段在给定数据表上必须是唯一的，如果不指定 `column` 选项，字段名将作为默认 `column`。

指定自定义列名：

```
'email' => 'unique:users,email_address'
```

自定义数据库连接

有时候，你可能需要自定义验证器生成的数据库连接，正如上面所看到的，设置 `unique:users` 作为验证规则将会使用默认数据库连接来查询数据库。要覆盖默认连接，在数据表名后使用”.“指定连接：

```
'email' => 'unique:connection.users,email_address'
```

强制一个唯一规则来忽略给定 ID：

有时候，你可能希望在唯一检查时忽略给定 ID，例如，考虑一个包含用户名、邮箱地址和位置的"更新属性"界面，当然，你将会验证邮箱地址是唯一的，然而，如果用户只改变用户名字段而并没有改变邮箱字段，你不想要因为用户已经拥有该邮箱地址而抛出验证错误，你只想要在用户提供的邮箱已经被别人使用的情况下才抛出验证错误，要告诉唯一规则忽略用户 ID，可以传递 ID 作为第三个参数：

```
'email' => 'unique:users,email_address,'.$user->id
```

添加额外的 `where` 子句：

还可以指定更多条件给 `where` 子句：

```
'email' => 'unique:users,email_address,NULL,id,account_id,1'
```

url

验证字段必须是基于 PHP 函数 `filter_var` 过滤的的有效 URL

6、添加条件规则

在某些场景下，你可能想要只有某个字段存在的情况下运行验证检查，要快速完成这个，添加 `sometimes` 规则到规则列表：

```
$v = Validator::make($data, [
    'email' => 'sometimes|required|email',
]);
```

在上例中，email 字段只有存在于`$data` 数组时才会被验证。

复杂条件验证

有时候你可能想要基于更复杂的条件逻辑添加验证规则。例如，你可能想要只有在另一个字段值大于 `100` 时才要求一个给定字段是必须的，或者，你可能需要只有当另一个字段存在时两个字段才都有给定值。添加这个验证规则并不是一件头疼的事。首先，创建一个永远不会改变的静态规则到 `Validator` 实例：

```
$v = Validator::make($data, [
    'email' => 'required|email',
    'games' => 'required|numeric',
]);
```

让我们假定我们的 web 应用服务于游戏收集者。如果一个游戏收集者注册了我们的应用并拥有超过 `100` 个游戏，我们想要他们解释为什么他们会有这么多游戏，例如，也许他们在运营一个游戏二手店，又或者他们只是喜欢收集。要添加这种条件，我们可以使用 Validator 实例上的 `sometimes` 方法：

```
$v->sometimes('reason', 'required|max:500', function($input) {
    return $input->games >= 100;
```

```
});
```

传递给 `sometimes` 方法的第一个参数是我们需要有条件验证的名称字段，第二个参数是我们想要添加的规则，如果作为第三个参数的闭包返回 `true`，规则被添加。该方法让构建复杂条件验证变得简单，你甚至可以一次为多个字段添加条件验证：

```
$v->sometimes(['reason', 'cost'], 'required', function($input)
{
    return $input->games >= 100;
});
```

注意：传递给闭包的`$input` 参数是 `Illuminate\Support\Fluent` 的一个实例，可用于访问输入和文件。

7、自定义验证规则

Laravel 提供了多种有用的验证规则；然而，你可能还是想要指定一些自己的验证规则。注册验证规则的一种方法是使用 `Validator` 门面的 extend 方法。让我们在服务提供者中使用这种方法来注册一个自定义的验证规则：

```
<?php

namespace App\Providers;

use Validator;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider{
    /**
     * 启动应用服务
     *
     * @return void
     */
    public function boot()
    {
        Validator::extend('foo', function($attribute, $value,
$parameters) {
            return $value == 'foo';
        });
    }

    /**
     * 注册服务提供者
     *
     * @return void
     */
```

```
    public function register()
    {
        //
    }
}
```

自定义验证器闭包接收三个参数：要验证的属性名称，属性值和传递给规则的参数数组。

你还可以传递类和方法到 `extend` 方法而不是闭包：

```
Validator::extend('foo', 'FooValidator@validate');
```

定义错误信息

你还需要为自定义规则定义错误信息。你可以使用内联自定义消息数组或者在验证语言文件中添加条目来实现这一目的。消息应该被放到数组的第一维，而不是在只用于存放属性指定错误信息的 `custom` 数组内：

```
"foo" => "Your input was invalid!",
"accepted" => "The :attribute must be accepted.",
// 验证错误信息其它部分...
```

当创建一个自定义验证规则时，你可能有时候需要为错误信息定义自定义占位符，可以通过创建自定义验证器然后调用 `Validator` 门面上的 `replacer` 方法来实现。可以在服务提供者的 `boot` 方法中编写代码：

```
/**
 * 启动应用服务
 *
 * @return void
 */
public function boot(){
    Validator::extend(...);
    Validator::replacer('foo', function($message, $attribute,
$rule, $parameters) {
        return str_replace(...);
    });
}
```

隐式扩展

默认情况下，被验证的属性如果没有提供或者验证规则为 `required` 而值为空，那么正常的验证规则，包括自定义扩展将不会执行。例如，`unique` 规则将不会检验 null 值：

```

$rules = ['name' => 'unique'];
$input = ['name' => null];
Validator::make($input, $rules)->passes(); // true
```

如果要求即使为空时也要验证属性，则必须要暗示属性是必须的，要创建一个隐式扩展，可以使用`Validator::extendImplicit()`方法：

```
Validator::extendImplicit('foo', function($attribute, $value, $parameters, $validator) {
    return $value == 'foo';
});
```

注意：一个隐式扩展仅仅暗示属性是必须的，至于它到底是缺失的还是空值这取决于你。

八、新手入门指南

简单任务管理系统

1、简介

快速入门指南会对 Laravel 框架做一个基本介绍，包括数据库迁移、Eloquent ORM、路由、验证、视图以及 Blade 模板等等。如果你是个 Laravel 新手甚至之前对 PHP 框架也很陌生，那么这里将会成为你的良好起点。如果你已经使用过 Laravel 获取其它 PHP 框架，可以考虑跳转到进阶指南（翻译中）。
为了演示 Laravel 特性的基本使用，我们将将会构建一个简单的、用于追踪所有要完成任务的任务列表（To-Do List），本教程完整的代码已经公开在 Github 上：https://github.com/laravel/quickstart-basic。

2、安装

安装 Laravel

当然，开始之前你首先要做的是安装一个新的 Laravel 应用。你可以使用 Homestead 虚拟机或者本地 PHP 开发环境来运行应用。设置好开发环境后，可以使用如下 Composer 命令安装应用：

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

安装 Quickstart 项目

当然你还可以通过克隆 GitHub 仓库到本地来安装：

```
git clone https://github.com/laravel/quickstart-basic quickstart

cd quickstart

composer install
```

```
php artisan migrate
```

如果你还不了解如何构建本地开发环境，可参考 Homestead 和安装文档。

3、准备好数据库

3.1 数据库迁移

首先，让我们使用迁移来定义数据表用于处理所有任务。Laravel 的数据库迁移特性提供了一个简单的方式来对数据表结构进行定义和修改：不需要让团队的每个成员添加列到本地数据库，只需要简单运行你提交到源码控制中的迁移即可实现数据表创建及修改。

那么，让我们来创建这个处理所有任务的数据表吧。Artisan 命令可以用来生成多种类从而节省重复的劳动，在本例中，我们使用 `make:migration` 命令生成 `tasks` 对应的数据表迁移：

```
php artisan make:migration create_tasks_table --create=tasks
```

该命令生成的迁移文件位于项目根目录下的 `database/migrations` 目录，可能你已经注意到了，`make:migration` 命令已经在迁移文件中为我们添加了自增 ID 和时间戳，接下来我们要编辑该文件添加更多的列到数据表 `tasks`：

```
<?php


use Illuminate\Database\Schema\Blueprint;

use Illuminate\Database\Migrations\Migration;


class CreateTasksTable extends Migration{

    /**

     * Run the migrations.

     *

     * @return void

     */

    public function up()

    {

        Schema::create('tasks', function (Blueprint $table) {

            $table->increments('id');

            $table->string('name');
```

```
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('tasks');
    }
}
```

要运行迁移，可以使用 Artisan 命令 `migrate`。如果你使用的是 Homestead，应该在虚拟机中运行该命令：

```
php artisan migrate
```

该命令会为我们创建迁移文件中定义的所有数据表，如果你使用数据库客户端软件查看数据库，可以看到已经创建了一个新的 `tasks` 表，其中包含了我们在迁移中定义的列。接下来，我们准备为这个数据表定义一个 Eloquent ORM 模型。

3.2 Eloquent 模型

Laravel 使用的默认 ORM 是 Eloquent，Eloquent 使用模型让数据存取变得简单和轻松，通常，每一个 Eloquent 模型都有一个与之对应的数据表。
所以我们要定义一个与刚刚创建的 `tasks` 表对应的 `Task` 模型，同样我们使用 Artisan 命令来生成这个模型：

```
 php artisan make:model Task
```

该模型类位于 `app` 目录下，默认情况下，模型类是空的，我们不需要告诉该 Eloquent 模型对应哪张数据表，这一点我们在 Eloquent 文档中提及过，这里默认对应的数据表是 `tasks`，下面是这个空的模型类：

```
<?php
```

```
namespace App;


use Illuminate\Database\Eloquent\Model;


class Task extends Model{

    //

}
```

了解更多关于 Eloquent 模型类的细节，可查看完整的 Eloquent 文档。

4、路由

4.1 路由存根

下面我们需要为应用定义一些路由，路由的作用是在用户访问指定页面时将页面 URL 匹配到被执行的控制器或匿名函数。默认情况下，所有的 Laravel 路由都定义在 `app/Http/routes.php`。
在本应用中，我们需要至少三个路由：显示所有任务的路由，添加新任务的路由，以及删除已存在任务的路由。接下来，让我们在 `app/Http/routes.php` 文件中创建这三个路由：

```
<?php


use App\Task;

use Illuminate\Http\Request;


/**

 * Display All Tasks

 */

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

   //

});


/**

 * Add A New Task

 */
```

```
Route::post('/task', function (Request $request) {
    //
});


/**
 * Delete An Existing Task
 */
Route::delete('/task/{id}', function ($id) {
    //
});
```

4.2 显示视图

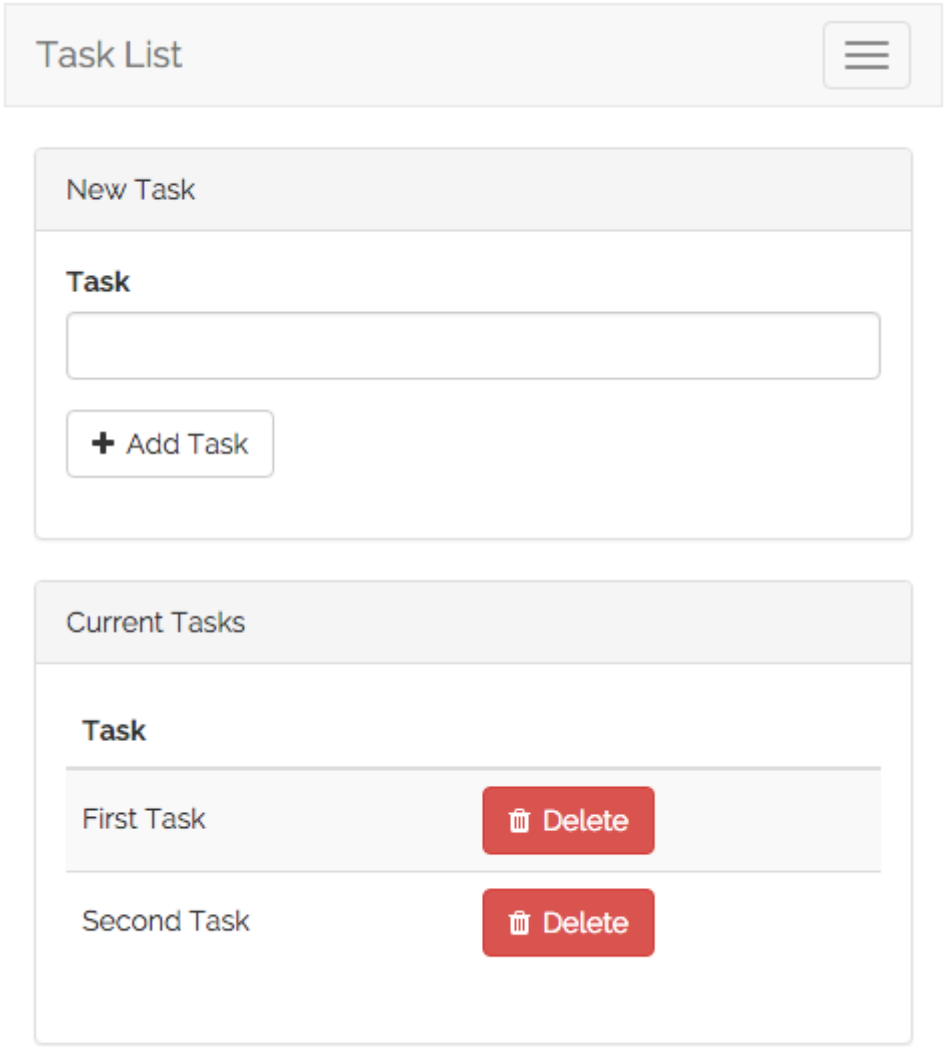
接下来，我们来填充/路由，在这个路由中，我们要渲染一个 HTML 模板，该模板包含添加新任务的表单，以及显示任务列表。
在 Laravel 中，所有的 HTML 模板都存放在 `resources/views` 目录下，我们可以使用 `view` 函数从路由中返回其中一个模板：

```
Route::get('/', function () {
    return view('tasks');
});
```

传递 tasks 到 view 函数将会创建一个对应视图模板为 resources/views/tasks.blade.php 的 View 对象。接下来我们需要去创建这个视图文件。

5、创建布局&视图

本应用为了简单处理只包含一个视图，其中包含了添加新任务的表单和所有任务的列表。为了让大家有一个直观的视觉效果，我们贴出该视图的截图，可以看到我们在视图中使用了基本的 Bootstrap CSS 样式：



5.1 定义布局

几乎所有的 web 应用都是在不同页面中共享同一个布局，例如，本应用在视图顶部有一个导航条，该导航条在每个页面都会出现。Laravel 通过在每个页面中使用 Blade 布局让共享这些公共特性变得简单。

正如我们之前讨论的，所有 Laravel 视图都存放在 `resources/views` 中，因此，我们在 `resources/views/layouts/app.blade.php` 中定义一个新的布局视图，`.blade.php` 扩展表明框架使用 Blade 模板引擎来渲染视图，当然，你可以使用原生的 PHP 模板，然而，Blade 提供了的标签语法可以帮助我们编写更加清爽、简短的模板。
编辑 `app.blade.php` 内容如下：

```
// resources/views/layouts/app.blade.php

<!DOCTYPE html><html lang="en">

    <head>

        <title>Laravel Quickstart - Basic</title>
```

```
        <!-- CSS And JavaScript -->
    </head>

    <body>
        <div class="container">
            <nav class="navbar navbar-default">
                <!-- Navbar Contents -->
            </nav>
        </div>

        @yield('content')
    </body>
</html>
```

注意布局中的`@yield('content')`部分，这是一个 Blade 指令，用于指定继承布局的子页面在这里可以注入自己的内容。接下来，我们来定义使用该布局的子视图来提供主体内容。

5.2 定义子视图

好了，我们已经创建了应用的布局视图，下面我们需要定义一个包含创建新任务的表单和已存在任务列表的视图，该视图文件存放在`resources/views/tasks.blade.php`。
我们将跳过 Bootstrap CSS 的样板文件而只专注在我们所关注的事情上，不要忘了，你可以从 GitHub 下载本应用的所有资源：

```
//resources/views/tasks.blade.php

@extends('layouts.app')

@section('content')

    <!-- Bootstrap Boilerplate... -->

    <div class="panel-body">
        <!-- Display Validation Errors -->
        @include('common.errors')

        <!-- New Task Form -->
        <form action="/task" method="POST" class="form-horizont
al">
```

```
            {{ csrf_field() }}

            <!-- Task Name -->
            <div class="form-group">
                <label for="task" class="col-sm-3 control-label
">Task</label>

                <div class="col-sm-6">
                    <input type="text" name="name" id="task-name
" class="form-control">
                </div>
            </div>

            <!-- Add Task Button -->
            <div class="form-group">
                <div class="col-sm-offset-3 col-sm-6">
                    <button type="submit" class="btn btn-default
">
                        <i class="fa fa-plus"></i> Add Task
                    </button>
                </div>
            </div>
        </form>
    </div>

    <!-- TODO: Current Tasks -->
@endsection
```

一些需要注意的事项

在继续往下之前，让我们简单谈谈这个模板。首先，我们使用`@extends`指令告诉 Blade 我们要使用定义在`resources/views/layouts/app.blade.php`的布局，所有`@section('content')`和`@endsection`之间的内容将会被注入到`app.blade.php`布局的`@yield('contents')`指令位置。
现在，我们已经为应用定义了基本的布局和视图，接下来，我们准备添加代码到`POST /task`路由来处理添加新任务到数据库。
`@include('common.errors')`指令将会加载`resources/views/common/errors.blade.php`模板中的内容，我们还没有定义这个模板，但很快就会了！
现在我们为应用定义了基本布局和视图文件，现在我们回到/路由：

```
Route::get('/', function () {
    return view('tasks');
});
```

接下来，我们准备添加代码到 `POST /task` 路由以便处理输入并添加新任务到数据库。

6、添加任务

6.1 验证

现在我们已经在视图中定义了表单，接下来需要在 POST /task 路由中编写代码处理表单请求，我们需要验证表单输入，然后才能创建一个新任务。

对这个表单而言，我们将 `name` 字段设置为必填项，而且长度不能超过 255 个字符。如果表单验证失败，将会跳转到前一个页面，并且将错误信息存放到一次性 Session 中：

```
Route::post('/task', function (Request $request) {
    $validator = Validator::make($request->all(), [
        'name' => 'required|max:255',
    ]);

    if ($validator->fails()) {
        return redirect('/')
            ->withInput()
            ->withErrors($validator);
    }

    // Create The Task...
});
```

$errors 变量

让我们停下来讨论下上述代码中的`->withErrors($validator)`部分，`->withErrors($validator)`会将验证错误信息存放到一次性 session 中，以便在视图中可以通过`$errors`变量访问。
我们在视图中使用了`@include('common.errors')`指令来渲染表单验证错误信息，`common.errors`允许我们在所有页面以统一格式显示错误信息。我们定义`common.errors`内容如下：

```
// resources/views/common/errors.blade.php

@if (count($errors) > 0)
    <!-- Form Error List -->
```

```
    <div class="alert alert-danger">
        <strong>Whoops! Something went wrong!</strong>

        <br><br>

        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```

注：`$errors` 变量在每个 Laravel 视图中都可以访问，如果没有错误信息的话它就是一个空的 `ViewErrorBag` 实例。

6.2 创建任务

现在输入验证已经做好了，接下来正式开始创建一个新任务。一旦新任务创建成功，页面会跳转到`/`。要创建任务，可以使用 Eloquent 模型提供的 `save` 方法：

```
Route::post('/task', function (Request $request) {
    $validator = Validator::make($request->all(), [
        'name' => 'required|max:255',
    ]);

    if ($validator->fails()) {
        return redirect('/')
            ->withInput()
            ->withErrors($validator);
    }

    $task = new Task;
    $task->name = $request->name;
```

```
    $task->save();


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

好了，到了这里，我们已经可以成功创建任务，接下来，我们继续添加代码到视图来显示所有任务列表。

6.3 显示已存在的任务

首先，我们需要编辑/路由传递所有已存在任务到视图。view 函数接收一个数组作为第二个参数，我们可以将数据通过该数组传递到视图中：

```
Route::get('/', function () {
    $tasks = Task::orderBy('created_at', 'asc')->get();


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

数据被传递到视图后，我们可以在 tasks.blade.php 中以表格形式显示所有任务。Blade 中使用@foreach 处理循环数据：

```
@extends('layouts.app')


@section('content')
    <!-- Create Task Form... -->


    <!-- Current Tasks -->
    @if (count($tasks) > 0)
        <div class="panel panel-default">
            <div class="panel-heading">
                Current Tasks
            </div>
```

```
            <div class="panel-body">
                <table class="table table-striped task-table">

                    <!-- Table Headings -->
                    <thead>
                        <th>Task</th>
                        <th> </th>
                    </thead>

                    <!-- Table Body -->
                    <tbody>
                    @foreach ($tasks as $task)
                        <tr>
                            <!-- Task Name -->
                            <td class="table-text">
                                <div>{{ $task->name }}</div>
                            </td>

                            <td>
                                <!-- TODO: Delete Button -->
                            </td>
                        </tr>
                    @endforeach
                    </tbody>
                </table>
            </div>
        </div>
    @endif
@endsection
```

至此，本应用基本完成。但是，当任务完成时我们还没有途径删除该任务，接下来我们就来处理这件事。

7、删除任务

7.1 添加删除按钮

我们在 `tasks.blade.php` 视图中留了一个“TODO”注释用于放置删除按钮。当删除按钮被点击时，`DELETE /task` 请求被发送到应用后台：

```
<tr>
    <!-- Task Name -->
    <td class="table-text">
        <div>{{ $task->name }}</div>
    </td>

    <!-- Delete Button -->
    <td>
        <form action="/task/{{ $task->id }}" method="POST">
            {{ csrf_field() }}
            {{ method_field('DELETE') }}

            <button>Delete Task</button>
        </form>
    </td>
</tr>
```

关于方法伪造

尽管我们使用的路由是 `Route::delete`，但我们在删除按钮表单中使用的请求方法为 `POST`，HTML 表单只支持 `GET` 和 `POST` 两种请求方式，因此我们需要使用某种方式来伪造 `DELETE` 请求。

我们可以在表单中通过输出 `method_field('DELETE')` 来伪造 `DELETE` 请求，该函数生成一个隐藏的表单输入框，然后 Laravel 识别出该输入并使用其值覆盖实际的 HTTP 请求方法。生成的输入框如下：

```
<input type="hidden" name="_method" value="DELETE">
```

7.2 删除任务

最后，让我们添加业务逻辑到路由中执行删除操作，我们可以使用 Eloquent 提供的 `findOrFail` 方法从数据库通过 ID 获取模型实例，如果不存在则抛出 404 异常。获取到模型后，我们使用模型的 `delete` 方法删除该模型在数据库中对应的记录。记录被删除后，跳转到`/`页面：

```
Route::delete('/task/{id}', function ($id) {
    Task::findOrFail($id)->delete();
    return redirect('/');
});
```

带用户功能的任务管理系统

1、简介

本进阶指南提供了对 Laravel 框架更深入的介绍，包括数据库迁移、Eloquent ORM、路由、认证、授权、依赖注入、验证、视图以及 Blade 模板。如果你对 Laravel 框架或其他 PHP 框架已经有了基本的认识，本章节将是你新的起点，如果你完全还是新手，请从新手入门指南开始。
本节的示例仍然是构建一个任务系统，但是在上一节基础上，本任务系统将允许用户注册登录，同样完整的代码已经放到 GitHub 上：https://github.com/laravel/quickstart-intermediate。

2、安装

安装 Laravel

首先你需要安装一个新的 Laravel 应用。你可以使用 Homestead 虚拟机或者本地 PHP 开发环境来运行应用。开发环境准备完毕后，可以使用 Composer 来安装应用：

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

安装 Quickstart 项目

你可以继续往下阅读，也可以选择去 GitHub 下载项目源码并在本地运行：

```
git clone https://github.com/laravel/quickstart-intermediate quickstart
cd quickstart
composer install
```

```
php artisan migrate
```

关于构建本地开发环境的详细文档可查看 Homestead 和安装文档。

3、准备好数据库

3.1 数据库迁移

首先，我们使用迁移来定于处理所有任务的数据库。Laravel 的数据库迁移使用平滑、优雅的 PHP 代码来提供一个简单的方式定义和修改数据表结构。团队成员们无需在本地数据库手动添加或删除列，只需要简单运行你提交到源码控制系统中的迁移即可。

users 表

由于我们允许用户注册，所以需要一张用来存储用户的表。幸运的是 Laravel 已经自带了这个迁移用于创建基本的 users 表，我们不需要手动生成。该迁移文件默认位于 `database/migrations` 目录下。

tasks 表

接下来，让我们来创建用于处理所有任务的数据表 `tasks`。我们使用 Artisan 命令 `make:migration` 来为 `tasks` 生成一个新的数据库迁移：

```
php artisan make:migration create_tasks_table --create=tasks
```

生成的新迁移文件位于 `database/migrations` 目录下。你可能已经注意到了，`make:migration` 命令已经在迁移文件中添加了自增 ID 和时间戳。我们将编辑该文件添加更多的字段到任务表 `tasks`：

```
<?php


use Illuminate\Database\Schema\Blueprint;

use Illuminate\Database\Migrations\Migration;


class CreateTasksTable extends Migration{

    /**

     * Run the migrations.

     *

     * @return void

     */

    public function up()

    {
```

```
        Schema::create('tasks', function (Blueprint $table) {
            $table->increments('id');
            $table->integer('user_id')->index();
            $table->string('name');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('tasks');
    }
}
```

其中，`user_id` 用于建立 `tasks` 表与 `users` 表之间的关联。
要运行迁移，可以使用 `migrate` 命令。如果你使用的是 Homestead，需要在虚拟机中运行该命令，因为你的主机不能直接访问 Homestead 上的数据库：

```
php artisan migrate
```

该命令将会创建迁移中定义的尚未创建的所有数据表。如果你使用 MySQL 客户端（如 Navicat For MySQL）查看数据表，你将会看到新的 `users` 表和 `tasks` 表。下一步，我们将要定义 Eloquent ORM 模型。

3.2 Eloquent 模型

Eloquent 是 Laravel 默认的 ORM，Eloquent 使用“模型”这一概念使得从数据库存取数据变得轻松。通常，每个 Eloquent 模型都对应一张数据表。

User 模型

首先，我们一个与 `users` 表相对应的模型 `User`。Laravel 已经自带了这一模型 `app/User`，所以我们不需要重复创建了。

Task 模型

接下来，我们来定义与 `tasks` 表相对应的模型 Task。同样，我们使用 Artisan 命令来生成模型类，在本例中，我们使用 `make:model` 命令：

```
php artisan make:model Task
```

该模型位于应用的 `app` 目录下，默认情况下，该模型类是空的。我们不需要明确告诉 Eloquent 模型对应哪张表，Laravel 底层会有一个映射规则，这一点在之前 Eloquent 文档已有说明，按照规则，这里 `Task` 模型默认对应 `tasks` 表。
接下来，让我们在 `Task` 模型类中加一些代码。首先，我们声明模型上的 name 属性支持“批量赋值”（关于批量赋值说明可查看这篇文章）：

```
<?php


namespace App;


use Illuminate\Database\Eloquent\Model;


class Task extends Model{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['name'];
}
```

我们将在后续添加路由到应用中学习更多如何使用 Eloquent 模型。当然，你也可以先去查看完整的 Eloquent 文档了解更多。

3.3 Eloquent 关联关系

现在，模型已经定义好了，我们需要将它们关联起来。例如，一个 `User` 实例对应多个 `Task` 实例，而一个 `Task` 实例从属于某个 `User`。定义关联关系后将允许我们更方便的获取关联模型：

```
$user = App\User::find(1);
```

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

tasks 关联关系

首先，我们在 `User` 模型中定义 `tasks` 关联关系。Eloquent 关联关系被定义成模型的方法，并且支持多种不同的关联关系类型（查看完整的 Eloquent 关联关系文档了解更多）。在本例中，我们将会在 `User` 模型中定义 `tasks` 方法并在其中调用 Eloquent 提供的 `hasMany` 方法：

```
<?php

namespace App;
// Namespace Imports...
class User extends Model implements AuthenticatableContract,
AuthorizableContract,CanResetPasswordContract
{
    use Authenticatable, Authorizable, CanResetPassword;

    // Other Eloquent Properties...

    /**
     * Get all of the tasks for the user.
     */
    public function tasks()
    {
        return $this->hasMany(Task::class);
    }
}
```

user 关联关系

接下来，我们会在 `Task` 模型中定义 `user` 关联关系。同样，我们将其定义为模型的方法。在本例中，我们使用 Eloquent 提供的 `belongsTo` 方法来定义该关联关系：

```
<?php


namespace App;


use App\User;
use Illuminate\Database\Eloquent\Model;


class Task extends Model{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['name'];


    /**
     * Get the user that owns the task.
     */
    public function user()
    {
        return $this->belongsTo(User::class);
    }
}
```

好极了！现在我们已经定义好了关联关系，接下来可以正式开始创建控制器了！

4、路由

在新手入门指南创建的任务管理系统中，我们在 `routes.php` 中使用闭包定义所有的业务逻辑。而实际上，大部分应用都会使用控制器来组织路由。

4.1 显示视图

我们还保留一个路由使用闭包：/路由，该路由是用于展示给游客的引导页，我们将在该路由中渲染欢迎页面。

在 Laravel 中，所有的 HTML 模板都位于 `resources/views` 目录，并且我们使用 `view` 函数从路由中返回其中一个模板：

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

    return view('welcome');

});
```

当然，我们需要创建这个视图，稍后就会。

4.2 用户认证

此外，我们还要让用户注册并登录到应用。通常，在 web 应用中构建整个登录认证层是一件相当冗长乏味的工作，然而，由于它是一个如此通用的需求，Laravel 试图将这一过程变得简单而轻松。

首先，注意到新安装的 Laravel 应用中已经包含了 `app/Http/Controllers/AuthController` 这个控制器，该控制器中使用了一个特殊的 `AuthenticatesAndRegistersUsers` trait，而这个 trait 中包含了用户注册登录的所必须的相关逻辑。

认证路由&视图

那么接下来我们该怎么做呢？我们仍然需要创建注册和登录模板并定义指向认证控制器 `AuthController` 的路由。我们可以通过 Artisan 命令 make:auth 来完成所有工作：

`php artisan make:auth --views`

注：如果你想要查看这些视图的完整示例，可以去下载相应的 GitHub 项目：

https://github.com/laravel/quickstart-intermediate

接下来，我们还要添加认证路由到路由文件，我们可以通过使用 Route 门面上的 auth 方法来实现这一目的，该方法会注册我们所需的所有认证路由，包括注册、登录和密码重置：

`// Authentication Routes...`

Route::auth();

4.3 任务控制器

由于我们需要获取和保存任务，所以还需要使用 Artisan 命令创建一个 `TaskController`，生成的控制器位于 `app/Http/Controllers` 目录：

```
php artisan make:controller TaskController
```

现在这个控制器已经生成了，让我们去 `app/Http/routes.php` 中定义一些指向该控制器的路由吧：

```
Route::get('/tasks', 'TaskController@index');

Route::post('/task', 'TaskController@store');

Route::delete('/task/{task}', 'TaskController@destroy');
```

设置所有任务路由需要登录才能访问

对本应用而言，我们想要所有任务需要登录用户才能访问，换句话说，用户必须登录到系统才能创建新任务。所以，我们需要限制访问任务路由的用户为登录用户。Laravel 使用中间件来处理这种限制。

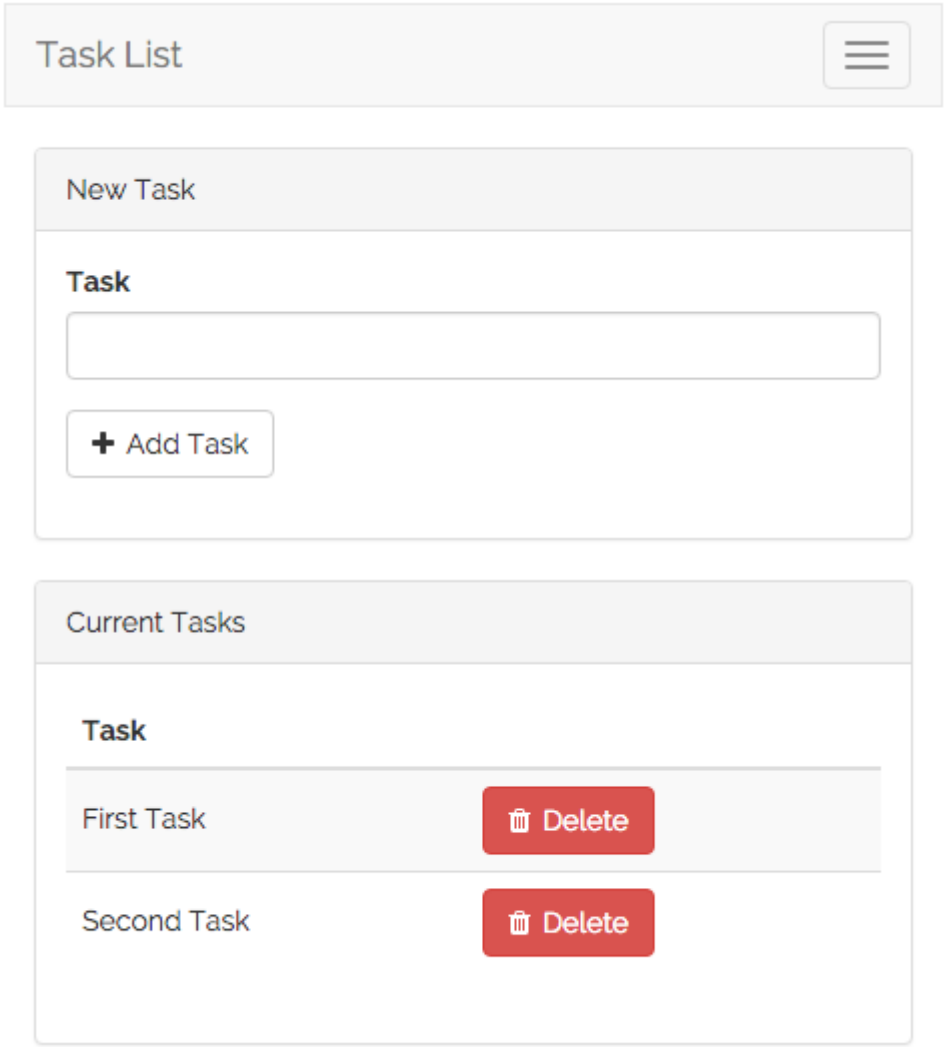
如果要限制登录用户才能访问该控制器的所有动作，可以在控制器的构造函数中添加对 `middleware` 方法的调用。所有有效的路由中间件都定义在 app/Http/Kernel.php 文件中。

在本例中，我们想要定义一个 `auth` 中间件到 `TaskController` 上的所有动作：

```
<?php


namespace App\Http\Controllers;


use App\Http\Requests;

use Illuminate\Http\Request;

use App\Http\Controllers\Controller;


class TaskController extends Controller{

    /**

     * Create a new controller instance.

     *

     * @return void

     */

    public function __construct()

    {

        $this->middleware('auth');

    }

}
```

5、创建布局&视图

本应用仍然只有一个视图，该视图包含了用于添加新任务的表单和显示已存在任务的列表。为了让你更直观的查看该视图，我们将已完成的应用截图如下：



5.1 定义布局

几乎所有的 web 应用都会在不同界面共享同一布局，例如，本应用顶部的导航条将会在每个页面显示。Laravel 使用 Blade 让不同页面共享这些公共特性变得简单。

正如我们之前讨论的，所有 Laravel 视图都存放在 `resources/views` 中，因此，我们在 `resources/views/layouts/app.blade.php` 中定义一个新的布局视图，`.blade.php` 扩展表明框架使用 Blade 模板引擎来渲染视图，当然，你可以使用原生的 PHP 模板，然而，Blade 提供了的标签语法可以帮助我们编写更加清爽、简短的模板。
编辑 app.blade.php 内容如下：

```
// resources/views/layouts/app.blade.php

<!DOCTYPE html><html lang="en">

    <head>

        <title>Laravel Quickstart - Advanced</title>
```

```
        <!-- CSS And JavaScript -->
    </head>

    <body>
        <div class="container">
            <nav class="navbar navbar-default">
                <!-- Navbar Contents -->
            </nav>
        </div>

        @yield('content')
    </body>
</html>
```

注意布局中的`@yield('content')`部分，这是一个 Blade 指令，用于指定继承布局的子页面在这里可以注入自己的内容。接下来，我们来定义使用该布局的子视图来提供主体内容。

5.2 定义子视图

好了，我们已经创建了应用的布局视图，下面我们需要定义一个包含创建新任务的表单和已存在任务列表的视图，该视图文件存放在`resources/views/tasks.blade.php`，对应`TaskController`中的`index`方法。
我们将跳过 Bootstrap CSS 的样板文件而只专注在我们所关注的事情上，不要忘了，你可以从 GitHub 下载本应用的所有资源：

```
// resources/views/tasks.blade.php

@extends('layouts.app')

@section('content')

    <!-- Bootstrap Boilerplate... -->

    <div class="panel-body">
        <!-- Display Validation Errors -->
```

```
        @include('common.errors')

        <!-- New Task Form -->
        <form action="/task" method="POST" class="form-horizontal">
            {{ csrf_field() }}

            <!-- Task Name -->
            <div class="form-group">
                <label for="task" class="col-sm-3 control-label">Task</label>

                <div class="col-sm-6">
                    <input type="text" name="name" id="task-name" class="form
-control">
                </div>
            </div>

            <!-- Add Task Button -->
            <div class="form-group">
                <div class="col-sm-offset-3 col-sm-6">
                    <button type="submit" class="btn btn-default">
                        <i class="fa fa-plus"></i> Add Task
                    </button>
                </div>
            </div>
        </form>
    </div>

    <!-- TODO: Current Tasks -->
@endsection
```

一些需要解释的地方

在继续往下之前，让我们简单谈谈这个模板。首先，我们使用`@extends`指令告诉 Blade 我们要使用定义在`resources/views/layouts/app.blade.php`的布局，所有`@section('content')`和`@endsection`之间的内容将会被注入到`app.blade.php`布局的`@yield('contents')`指令位置。
@include('common.errors')指令会加载 resources/views/common/errors.blade.php 模板，我们后续会定义这个模板。

现在，我们已经为应用定义了基本的布局和视图，然后我们回到`TaskController`的`index`方法：

```
/**
 * Display a list of all of the user's task.
 *
 * @param Request $request
 * @return Response
 */
public function index(Request $request){
    return view('tasks.index');
}
```

接下来，让我们继续添加代码到`POST /task`路由的控制器方法来处理表单输入并添加新任务到数据库。

6、添加任务

6.1 验证表单输入

现在我们已经在视图中定义了表单，接下来需要编写代码到`TaskController@store`方法来处理表单请求并创建一个新任务。
对这个表单而言，我们将`name`字段设置为必填项，而且长度不能超过 255 个字符。如果表单验证失败，将会跳转到`/tasks`页面，并且将错误信息存放到一次性 session 中：

```
/**
 * Create a new task.
 *
 * @param Request $request
 * @return Response
 */
```

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

    // Create The Task...
}
```

如果你已经看过新手入门教程，那么你可能会注意到这里的验证代码与之前大不相同，这是因为我们现在在控制器中，可以方便地调用 `ValidatesRequests` trait（包含在 Laravel 控制器基类中）提供的 `validate` 方法。
我们甚至不需要手动判读是否验证失败然后重定向。如果验证失败，用户会自动被重定向到来源页面，而且错误信息也会被存放到一次性 Session 中。简直太棒了，有木有！

$errors 变量

我们在视图中使用了`@include('common.errors')`指令来渲染表单验证错误信息，`common.errors` 允许我们在所有页面以统一格式显示错误信息。我们定义 `common.errors` 内容如下：

```
// resources/views/common/errors.blade.php

@if (count($errors) > 0)
    <!-- Form Error List -->
    <div class="alert alert-danger">
        <strong>Whoops! Something went wrong!</strong>

        <br><br>

        <ul>
        @foreach ($errors->all() as $error)
            <li>{{ $error }}</li>
        @endforeach
        </ul>
    </div>
```

```
@endif
```

注：`$errors` 变量在每个 Laravel 视图中都可以访问，如果没有错误信息的话它就是一个空的 `ViewErrorBag` 实例。

6.2 创建任务

现在输入验证已经做好了，接下来正式开始创建一个新任务。一旦新任务创建成功，页面会跳转到`/tasks`。要创建任务，可以借助 Eloquent 模型的关联关系。
大部分 Laravel 的关联关系提供了 `save` 方法，该方法接收一个关联模型实例并且会在保存到数据库之前自动设置外键值到关联模型上。在本例中，`save` 方法会自动将当前用户登录认证用户的 ID 赋给给给定任务的 `user_id` 属性。我们通过`$request->user()`获取当前登录用户实例：

```
/**
 * Create a new task.
 *
 * @param Request $request
 * @return Response
 */
public function store(Request $request){
    $this->validate($request, [
        'name' => 'required|max:255',
    ]);

    $request->user()->tasks()->create([
        'name' => $request->name,
    ]);

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

很好，到了这里，我们已经可以成功创建任务，接下来，我们继续添加代码到视图来显示所有任务列表。

7、显示已存在的任务

首先，我们需要编辑 `TaskController@index` 传递所有已存在任务到视图。`view` 函数接收一个数组作为第二个参数，我们可以将数据通过该数组传递到视图中：

```
/**
 * Display a list of all of the user's task.
 *
 * @param Request $request
 * @return Response
 */
public function index(Request $request){
    $tasks = Task::where('user_id', $request->user()->id)->get();

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

这里，我们还要讲讲 Laravel 的依赖注入，这里我们将 `TaskRepository` 注入到 `TaskController`，以方便对 `Task` 模型所有数据的访问和使用。

7.1 依赖注入

Laravel 的服务容器是整个框架中最重要的特性，在看完快速入门教程后，建议去研习下服务容器的文档。

创建 Repository

正如我们之前提到的，我们想要定义一个 `TaskRepository` 来处理所有对 `Task` 模型的数据访问，随着应用的增长当你需要在应用中共享一些 Eloquent 查询时这就变得特别有用。因此，我们创建一个 `app/Repositories` 目录并在其中创建一个 `TaskRepository` 类。记住，Laravel 项目的 `app` 文件夹下的所有目录都使用 PSR-4 自动加载标准被自动加载，所以你可以在其中随心所欲地创建需要的目录：

```
<?php


namespace App\Repositories;
```

```
use App\User;
use App\Task;


class TaskRepository{
    /**
     * Get all of the tasks for a given user.
     *
     * @param User $user
     * @return Collection
     */
    public function forUser(User $user)
    {
        return Task::where('user_id', $user->id)
            ->orderBy('created_at', 'asc')
            ->get();
    }
}
```

注入 Repository

Repository 创建好了之后，可以简单通过在 `TaskController` 的构造函数中以类型提示的方式注入该 Repository，然后就可以在 `index` 方法中使用 —— 由于 Laravel 使用容器来解析所有控制器，所以我们的依赖会被自动注入到控制器实例：

```
<?php


namespace App\Http\Controllers;


use App\Task;use App\Http\Requests;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;
use App\Repositories\TaskRepository;
```

```
class TaskController extends Controller{
    /**
     * The task repository instance.
     *
     * @var TaskRepository
     */
    protected $tasks;

    /**
     * Create a new controller instance.
     *
     * @param TaskRepository $tasks
     * @return void
     */
    public function __construct(TaskRepository $tasks)
    {
        $this->middleware('auth');
        $this->tasks = $tasks;
    }

    /**
     * Display a list of all of the user's task.
     *
     * @param Request $request
     * @return Response
     */
    public function index(Request $request)
    {
        return view('tasks.index', [
            'tasks' => $this->tasks->forUser($request->user()),
```

```
        ]);
    }
}
```

7.2 显示任务

数据被传递到视图后，我们可以在 `tasks/index.blade.php` 中以表格形式显示所有任务。Blade 中使用 `@foreach` 处理循环数据：

```
@extends('layouts.app')


@section('content')
     <!-- Create Task Form... -->


    <!-- Current Tasks -->
    @if (count($tasks) > 0)
        <div class="panel panel-default">
            <div class="panel-heading">
                Current Tasks
            </div>


            <div class="panel-body">
                <table class="table table-striped task-table">


                <!-- Table Headings -->
                <thead>
                    <th>Task</th>
                    <th> </th>
                </thead>


                <!-- Table Body -->
                <tbody>
```

```
                @foreach ($tasks as $task)
                    <tr>
                        <!-- Task Name -->
                        <td class="table-text">
                            <div>{{ $task->name }}</div>
                        </td>

                        <td>
                            <!-- TODO: Delete Button -->
                        </td>
                    </tr>
                @endforeach
                </tbody>
                </table>
            </div>
        </div>
    @endif
@endsection
```

至此，本应用基本完成。但是，当任务完成时我们还没有途径删除该任务，接下来我们就来处理这件事。

8、删除任务

8.1 添加删除按钮

我们在 `tasks/index.blade.php` 视图中留了一个“TODO”注释用于放置删除按钮。当删除按钮被点击时，`DELETE /task` 请求被发送到应用后台并触发 `TaskController@destroy` 方法：

```
<tr>
    <!-- Task Name -->
    <td class="table-text">
        <div>{{ $task->name }}</div>
    </td>
```

```
    <!-- Delete Button -->
    <td>
        <form action="/task/{{ $task->id }}" method="POST">
            {{ csrf_field() }}
            {{ method_field('DELETE') }}

            <button>Delete Task</button>
        </form>
    </td>
</tr>
```

关于方法伪造

尽管我们使用的路由是 `Route::delete`，但我们在删除按钮表单中使用的请求方法为 `POST`，HTML 表单只支持 GET 和 POST 两种请求方式，因此我们需要使用某种方式来伪造 `DELETE` 请求。
我们可以在表单中通过输出 `method_field('DELETE')`来伪造 `DELETE` 请求，该函数生成一个隐藏的表单输入框，然后 Laravel 识别出该输入并使用其值覆盖实际的 HTTP 请求方法。生成的输入框如下：

```
<input type="hidden" name="_method" value="DELETE">
```

8.2 路由模型绑定

现在，我们准备在 `TaskController` 中定义 `destroy` 方法，但是，在此之前，让我们回顾下路由中对删除任务的定义：

```
Route::delete('/task/{task}', 'TaskController@destroy');
```

对应控制器 TaskController 中删除方法 destroy 定义如下：

```
/**
 * Destroy the given task.
 *
 * @param  Request  $request
 * @param  Task  $task
 * @return Response
 */
```

```
public function destroy(Request $request, Task $task){
    //
}
```

由于路由中的{task}变量与控制器方法中的$task变量相匹配，Laravel 的隐式模型绑定特性将会自动注入相应的 Task 模型实例到 destroy 方法中。

8.3 用户授权

现在，我们已经将 Task 实例注入到 destroy 方法；然而，我们并不能保证当前登录认证用户是给定任务的实际拥有人。例如，一些恶意请求可能尝试通过传递随机任务 ID 到 /tasks/{task}链接删除另一个用户的任务。因此，我们需要使用 Laravel 的授权功能来确保当前登录用户拥有对注入到路由中的 Task 实例进行删除的权限。

创建 Policy

Laravel 使用“策略”来将授权逻辑组织到单个类中，通常，每个策略都对应一个模型，因此，让我们使用 Artisan 命令创建一个 TaskPolicy，生成的文件位于 app/Policies/TaskPolicy.php：

```
php artisan make:policy TaskPolicy
```

接下来，让我们添加 destroy 方法到策略中，该方法会获取一个 User 实例和一个 Task 实例。该方法简单检查用户 ID 和任务的 user_id 值是否相等。实际上，所有的策略方法都会返回 true 或 false：

```
<?php


namespace App\Policies;


use App\User;

use App\Task;

use Illuminate\Auth\Access\HandlesAuthorization;


class TaskPolicy{

    use HandlesAuthorization;


    /**

     * Determine if the given user can delete the given task.

     *

     * @param User $user
```

```
     * @param Task $task
     * @return bool
     */
    public function destroy(User $user, Task $task)
    {
        return $user->id === $task->user_id;
    }
}
```

最后，我们需要关联 `Task` 模型和 `TaskPolicy`，这可以通过在 `app/Providers/AuthServiceProvider.php` 文件的 `policies` 属性中添加注册来实现，注册后会告知 Laravel 无论何时我们尝试授权动作到 `Task` 实例时该使用哪个策略类进行判断：

```
/**
 * The policy mappings for the application.
 *
 * @var array
 */
protected $policies = [
    'App\Task' => 'App\Policies\TaskPolicy',
];
```

授权动作

现在我们编写好了策略，让我们在 `destroy` 方法中使用它。所有的 Laravel 控制器中都可以调用 `authorize` 方法，该方法由 `AuthorizesRequest` trait 提供：

```
/**
 * Destroy the given task.
 *
 * @param Request $request
 * @param Task $task
 * @return Response
 */
public function destroy(Request $request, Task $task){
```

```
    $this->authorize('destroy', $task);

    // Delete The Task...

}
```

我们可以检查下该方法调用：传递给 `authorize` 方法的第一个参数是我们想要调用的策略方法名，第二个参数是当前操作的模型实例。由于我们在之前告诉过 Laravel，`Task` 模型对应的策略类是 `TaskPolicy`，所以框架知道触发哪个策略类上的 `destroy` 方法。当前用户实例会被自动传递到策略方法，所以我们不需要手动传递。
如果授权成功，代码会继续执行。如果授权失败，会抛出一个 403 异常并显示一个错误页面给用户。

注：除此之外，Laravel 还提供了其它授权服务实现方式，可以查看授权文档了解更多。

8.4 删除任务

最后，让我们添加业务逻辑到路由中执行删除操作，我们可以使用 Eloquent 提供的 `delete` 方法从数据库中删除给定的模型实例。记录被删除后，跳转到 `/tasks` 页面：

```
/**
 * Destroy the given task.
 *
 * @param Request $request
 * @param Task $task
 * @return Response
 */
public function destroy(Request $request, Task $task){

    $this->authorize('destroy', $task);

    $task->delete();

    return redirect('/tasks');

}
```