

Table of Contents

Introduction	1.1
1 前言	1.2
1-1 翻译说明	1.2.1
1-2 发行说明	1.2.2
1-3 升级说明	1.2.3
1-4 贡献引导	1.2.4
2 入门指南	1.3
2-1 安装	1.3.1
2-2 配置信息	1.3.2
2-3 文件夹结构	1.3.3
2-4 请求周期	1.3.4
3 开发环境部署	1.4
3-1 Homestead	1.4.1
3-2 Valet	1.4.2
4 核心概念	1.5
4-1 服务容器	1.5.1
4-2 服务提供者	1.5.2
4-3 Facades	1.5.3
4-4 Contracts	1.5.4
5 HTTP层	1.6
5-1 路由	1.6.1
5-2 中间件	1.6.2
5-3 CSRF保护	1.6.3
5-4 控制器	1.6.4
5-5 请求	1.6.5
5-6 响应	1.6.6
5-7 视图	1.6.7
5-8 Session	1.6.8
5-9 表单验证	1.6.9
6 前端	1.7
6-1 Blade模版	1.7.1
6-2 本地化	1.7.2
6-3 前端指南	1.7.3

6-4 编辑资源Mix	1.7.4
7 安全	1.8
7-1 用户认证	1.8.1
7-2 Passport OAuth 认证	1.8.2
7-3 用户授权	1.8.3
7-4 加密解密	1.8.4
7-5 哈希	1.8.5
7-6 重置密码	1.8.6
8 综合话题	1.9
8-1 Artisan命令行	1.9.1
8-2 广播系统	1.9.2
8-3 缓存系统	1.9.3
8-4 集合	1.9.4
8-5 错误与日志	1.9.5
8-6 事件系统	1.9.6
8-7 文件存储	1.9.7
8-8 辅助函数	1.9.8
8-9 邮件发送	1.9.9
8-10 消息通知	1.9.10
8-11 扩展包开发	1.9.11
8-12 队列	1.9.12
8-13 任务调度	1.9.13
9 数据库	1.10
9-1 快速入门	1.10.1
9-2 查询构造器	1.10.2
9-3 分页	1.10.3
9-4 数据库迁移	1.10.4
9-5 数据填充	1.10.5
9-6 Redis	1.10.6
10 Eloquent ORM	1.11
10-1 快速入门	1.11.1
10-2 模型关联	1.11.2
10-3 Eloquent 集合	1.11.3
10-4 修改器	1.11.4
10-5 序列化	1.11.5
11 测试	1.12

11-1 快速入门	1.12.1
11-2 HTTP 测试	1.12.2
11-3 浏览器测试 Dusk	1.12.3
11-4 数据库测试	1.12.4
11-5 测试模拟器	1.12.5
12 官方扩展包	1.13
12-1 Cashier 交易工具包	1.13.1
12-2 Envoy 部署工具	1.13.2
12-3 Scout 全文搜索	1.13.3
12-4 Socialite 社会化登录	1.13.4

Introduction

1 前言

Laravel 5.4 中文文档

说明

为了更好的阅读文档, 请查看 [Laravel China 文档导读](#)。

此文档由 [Laravel China 社区](#) 用户协同翻译产生, 本次 5.4 版本的翻译参与人员总共 29 人, 完整的参与人员列表请见 : [Laravel 5.4 译者](#) 。

感谢这些可爱的译者, 感谢他们的热情。

相关讨论

- [Laravel 5.4 文档翻译召集](#)
- [Laravel 5.4 中文文档翻译完成](#)

排版规范

此文档遵循 [中文排版指南](#) 规范, 并在此之上遵守以下约定:

- 英文的左右保持一个空白, 避免中英文字黏在一起;
- 使用全角标点符号;
- 严格遵循 Markdown 语法;
- 原文中的双引号 (" ") 请代换成中文的引号 (「」) 符号怎么打出来见 [这里](#) ;
- 「加亮」和「加粗」和「[链接](#)」都需要在左右保持一个空格。

翻译对照列表

按照英文字母排序。

A

- Aggregate 聚合
- Array 数组
- Artisan (命令, 不翻)
- Argument 参数
- Assets 资源文件
- Authorization 用户授权
- Adapter 接口

B

- Blade (专为 Laravel 发明的 PHP 模板引擎, 不翻)

- Bundle (上一代的 package 名称, 不翻)
- Binding 绑定

C

- Cache 缓存
- Call 调用
- Callback 回调
- Classes 类
- CLI 命令行接口
- Command 命令
- Command Line 命令行
- Component 组件
- Console 终端
- Context 情境
- Controller 控制器
- Controller Action 控制器行为
- constructor 类构造器
- Cookie (不翻)
- Composer (开源项目名称, 不翻)
- Credentials 凭证
- Closure 闭包
- Configuration 配置信息
- Chain 链式
- Chain Methods 链式调用
- Contracts 契约

D

- Database-Transactions 数据库事务
- Deferred Providers 延迟提供者
- Driver 驱动
- Dependency Injection 依赖注入

E

- Event 事件
- Extend 扩展
- Extension 扩展
- Eloquent (不翻)
- Exception 异常
- Echo (不翻)
- Elixir (不翻)

F

- Facades (不翻)
- Framework 框架
- Filter 过滤器
- Form 表单
- Function 函数

G

- Guide 指南
- Guard 保卫

H

- Helper 辅助函数
- Hash 哈希 (可不翻)
- Homestead (不翻)
- Header 标头
- Hook 钩子

I

- Instance 实例
- IoC (不翻)
- inheritance 继承
- implements 实现

J

- Job 任务

K

- Key 键

L

- Laravel (不翻)
- Listener 监听器
- Library 函数库

M

- Method 方法
- Migration 迁移
- Model 模型
- Middleware 中间件

N

- Namespace 命名空间

O

- Object 对象

P

- Package 扩展包
- Packagist (开源项目名称, 不翻)
- Provider 提供者
- Prefix 前缀
- Presenter (不翻)
- Pipeline 管道
- Policies 策略
- Passport (不翻)

Q

- Queue 队列
- Query Builder 查询语句构造器

R

- Route / Routing 路由
- Router 路由器
- Request 请求
- Response 响应
- Resolved 解析
- Repository (不翻)
- Redirect 重定向
- (Database's) Rollback 还原

S

- Schema 数据库结构
- Service 服务

- Service Container 服务容器
- Service Providers 服务提供者
- Session (不翻)
- Seed 数据填充
- Scheduler 计划器
- Scout (不翻)

T

- Tag 标签
- Table 数据表
- Templates 模板
- Terminal 终端
- Token 令牌
- Timestamps 时间戳
- Type-hint 类型提示
- Trait (不翻)
- Ternary statement 三元运算符
- Throw (Exception) 抛出 (异常)

V

- Vagrant (开源项目名称, 不翻)
- Vagrant Box (开源项目名称, 不翻)
- View 视图
- Vendor 供应商
- View Composer 视图组件

W

- Workbench (开源项目名称, 不翻)
- Webhooks (不翻)
- Word Factor 加密系数
- Webpack (不翻)

Laravel 发行说明

- 支持策略
- [Laravel 5.4](#)
- [Laravel 5.3](#)
- [Laravel 5.2](#)
- [Laravel 5.1.11](#)
- [Laravel 5.1.4](#)
- [Laravel 5.1](#)
- [Laravel 5.0](#)
- [Laravel 4.2](#)
- [Laravel 4.1](#)

支持策略

对于 LTS 版本，比如 Laravel 5.1，会提供为期两年的 bug 修复和三年的安全修复支持。LTS 版本是 Laravel 能提供的维护时间最长的发行版。

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。

[Laravel 的发布路线图](#) - by Summer

Laravel 5.4.22

Laravel 5.4.22 为 Laravel 5.4 系列版本中能对应用程序中的用户进行网络钓鱼的安全漏洞安装了补丁。使用密码重置系统，恶意用户可以尝试欺骗你的用户将登录凭据输入到他们控制的单独应用程序中。由于密码重置通知使用主机传入请求来获取重置密码的 URL，因此用来生成重置密码的 URL 的主机可能会被欺骗。如果用户没有注意到他们访问的域名不正确，他们可能会意外将自己的登录凭据输入到恶意的应用程序中。

在 Laravel 5.1 的应用程序中，密码重置通知由开发人员维护，因此此漏洞可能存在或可能不存在。你应该验证应用程序生成密码重置的链接是否是绝对的 URL：

```
 {{ url('http://example.com/password/reset/'.$token) }}
```

Laravel 5.4

Laravel 5.4 继续在 Laravel 5.3 的基础上进行优化，新特性包括以下：

- 在邮件和通知中支持 [Markdown](#)；
- [Laravel Dusk 浏览器自动测试框架](#)；
- [Laravel Mix](#)；
- Blade "components" 和 "slots"；

- 在广播频道上进行路由模型绑定；
- 在集合中支持高阶消息传递；
- 基于对象的 Eloquent 事件；
- 任务级别的「重试」和「超时」设置；
- "实时" Facades；
- 更好的支持 Redis Cluster；
- 自定义 pivot 表模型；
- 两个新的中间件，用于输入修剪空格和清除非必要字段，等等。

此外，还对整个框架代码进行了 reviewed 和重构，以使代码更加干净和清晰。

{tip} 这个文档总结了许多框架值得注意的改进。需要知道更多细节请参考 Github 上的更新记录 [on GitHub](#)。

Markdown 邮件和通知

{video} Laracasts 上关于此新特性的免费视频 [video tutorial](#)。

此新特性允许我们在邮件通知中使用 Markdown 语法，Laravel 可以将这些消息渲染成美观、响应式的 HTML 模板，同时自动生成纯文本副本，下面是一个 Markdown 格式的邮件示例：

```
@component('mail::message')
# Order Shipped

Your order has been shipped!

@component('mail::button', ['url' => $url])
View Order
@endcomponent

Next Steps:

- Track Your Order On Our Website
- Pre-Sign For Delivery

Thanks,<br>
{{ config('app.name') }}
@endcomponent
```

利用这个简单的 Markdown 模板，Laravel 可以生成一个响应式 HTML 邮件和纯文本副本：



Order Shipped

Your order has been shipped!

[View Order](#)

Next Steps:

- Track Your Order On Our Website
- Pre-Sign For Delivery

Thanks,
Laravel



要想了解更多细节，查看 [mail](#) 和 [notification](#) 文档。

{tip} 你可以导出所有的 Markdown 邮件组件到自己的应用中进行自定义。使用 `vendor:publish` Artisan 命令 `laravel-mail` 选项来导出资源。

Laravel Dusk

{video} Laracasts 上关于此新特性的免费视频 [video tutorial](#)。

Laravel Dusk 提供一个优雅，简单易用的浏览器自动化测试 API。默认情况下，Dusk 不需要再你安装 JDK 或 Selenium 。Dusk 使用独立的 [ChromeDriver](#) 安装方式。你也可自由的使用其他 Selenium 兼容驱动。

Dusk 运行使用真实的浏览器，所有你可以轻松地对那些重度使用 JavaScript 的引用进行测试和交互：

```
/**
```

```

 * 一个基本浏览器测试例子.
 *
 * @return void
 */
public function testBasicExample()
{
    $user = factory(User::class)->create([
        'email' => 'taylor@laravel.com',
    ]);

    $this->browse(function ($browser) use ($user) {
        $browser->loginAs($user)
            ->visit('/home')
            ->press('Create Playlist')
            ->whenAvailable('.playlist-modal', function ($modal) {
                $modal->type('name', 'My Playlist')
                    ->press('Create');
            });

        $browser->waitForText('Playlist Created');
    });
}

```

需要了解更多关于 Dusk 的细节，查看 [Dusk 文档](#)

Laravel Mix

{video} Laracasts 上关于此新特性的免费视频 [video tutorial](#)。

Laravel Mix 是 Laravel Elixir 思想继承者，完全基于 Webpack 而非 Gulp。Laravel Mix 提供流式 API 定义 Webpack 构建步骤，有几种已经定义的 CSS 和 JavaScript 预处理器。通过简单的方法链，你可以流畅的定义你的资源构建流水线。例如：

```

mix.js('resources/assets/js/app.js', 'public/js')
    .sass('resources/assets/sass/app.scss', 'public/css');

```

Blade Components & Slots

{video} Laracasts 上关于此新特性的免费视频 [video tutorial](#)。

Blade components 和 slots 提供与 sections 和 layouts 相似功能；然而，有些人会觉得 components 和 slots 的思想更易理解。首先，让我们假设一个可重用的 "alert" 组件想要在整个应用进行重用：

```

<!-- /resources/views/alert.blade.php -->

<div class="alert alert-danger">
    {{ $slot }}

```

```
</div>
```

`{{ $slot }}` 变量包含我们想要插入的组件内容，要构建这个组件，我们可以使用 `component` Blade 指令：

```
@component('alert')
    <strong>Whoops!</strong> Something went wrong!
@endcomponent
```

命名 slots 允许在单个组件中定义多个 slots：

```
<!-- /resources/views/alert.blade.php -->

<div class="alert alert-danger">
    <div class="alert-title">{{ $title }}</div>

    {{ $slot }}
</div>
```

命名 slots 可以通过 `@slot` 指令注入。一个 `@slot` 中的所有内容都会被传递给 `@slot` 变量：

```
@component('alert')
    @slot('title')
        Forbidden
    @endslot

    You are not allowed to access this resource!
@endcomponent
```

需要了解更多关于 components 和 slots 的细节，查看 [Blade documentation](#)。

广播模型绑定

和 HTTP 路由一样，频道路由也使用隐式和显式 [路由模型绑定](#)。例如，可以通过请求一个实际的 `Order` 模型实例来取代之前获取字符串或数字订单ID ID：

```
use App\Order;

Broadcast::channel('order.{order}', function ($user, Order $order) {
    return $user->id === $order->user_id;
});
```

需要了解更多关于广播模型绑定，查看 [事件广播](#)。

集合高阶消息传递

{video} Laracasts 上关于此新特性的免费视频 [video tutorial](#)。

集合现在支持 "高阶消息传递"，从而使集合的操作更为精简，目前支持高阶消息传递方法有：

```
contains、each、every、filter、first、map、partition、reject、sortBy、
sortByDesc、sum。
```

每一个高阶消息传递都可以通过集合实例的动态属性进行访问，例如，使用 each 的高阶消息传递去调用集合的某个对象：

```
$users = User::where('votes', '>', 500)->get();

$users->each->markAsVip();
```

类似的，我们可以通过 sum 的高阶消息传递在用户集合中聚合所有的投票数：

```
$users = User::where('group', 'Development')->get();

return $users->sum->votes;
```

基于对象的 Eloquent 事件

{video} Laracasts 上关于此新特性的免费视频 [video tutorial](#)。

Eloquent 事件处理器现在可以被映射到事件对象上。这提供了一种直观处理 Eloquent 事件并易于测试的方式。在开始使用之前，先在你的 Eloquent 模型中定义一个 \$events 属性数组，在这个数组中可以定义 Eloquent 模型的生命周期中属于你的 [事件 classes](#)：

```
<?php

namespace App;

use App\Events\UserSaved;
use App\Events\UserDeleted;
use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * The event map for the model.
     *
     * @var array
     */
    protected $events = [
        'saved' => UserSaved::class,
```

```

        'deleted' => UserDeleted::class,
    ];
}

```

任务级别重试和超时

5.4版本以前，任务队列 "retry" 和 "timeout" 设置只能在全局的队列配置中用命令行设置。现在可以单独在任务类中配置每一个任务的 "重试" 和 "超时":

```

<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
{
    /**
     * The number of times the job may be attempted.
     *
     * @var int
     */
    public $tries = 5;

    /**
     * The number of seconds the job can run before timing out.
     *
     * @var int
     */
    public $timeout = 120;
}

```

需要了更多，查看 [队列](#).

请求清理中间件

{video} Laracasts 上关于此新特性的免费视频 [video tutorial](#)。

Laravel 5.4 在默认的中间件栈中引入了两个新的中间件：`TrimStrings` 和 `ConvertEmptyStringsToNull`：

```

/**
 * 应用的全局 HTTP 中间件栈
 *
 * 这些中间件会对你的每一个请求运行
 *
 * @var array
 */
protected $middleware = [
    \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
]

```

```
\Illuminate\Foundation\Http\Middleware\ValidatePostSize::class,
\App\Http\Middleware\TrimStrings::class,
\Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::class,
];
```

新增的中间件会自动去除请求输入值首尾的空格、将空字符串转换为 `null`。这可以帮助你获得正确的输入而不需要重复在每一个路由和控制器中调用 `trim` 方法。

"实时" Facades

{video} Laracasts 上关于此新特性的免费视频 [video tutorial](#)。

5.4 版本以前，只有 Laravel 内置的服务才可以提供 [Facades](#)，提供快速的，简短的方式访问这些服务容器的方法。在 Laravel 5.4 中，你可以轻松的将你的任意类库实时的转换成 Facades，只需要将类名导入到 `Facades` 中。例如，假设你的应用中有这样一个类：

```
<?php

namespace App\Services;

class PaymentGateway
{
    protected $tax;

    /**
     * 创建一个新的支付网关实例
     *
     * @param TaxCalculator $tax
     * @return void
     */
    public function __construct(TaxCalculator $tax)
    {
        $this->tax = $tax;
    }

    /**
     * Pay the given amount.
     *
     * @param int $amount
     * @return void
     */
    public function pay($amount)
    {
        // Pay an amount...
    }
}
```

你可以这样以 `Facades` 的方式调用这个类的方法：

```
use Facades\ {
    App\Services\PaymentGateway
};

Route::get('/pay/{amount}', function ($amount) {
    PaymentGateway::pay($amount);
});
```

当然，如果你以这种方式实现实时 Facades，就可以使用 Laravel 的 [Facades 模拟功能](#) 编写测试用例：

```
PaymentGateway::shouldReceive('pay')->with('100');
```

自定义 Pivot 表模型

在 Laravel 5.3，所有的 `belongsToMany` 关联关系使用同一个内置的 `Pivot` 模型实例。在 Laravel 5.4 中你可以为这些 pivot 表自定义模型类，如果你想要定义一个自定义模型，可以在定义关联关系时使用 `using` 方法：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Role extends Model
{
    /**
     * 属于这个角色的用户
     */
    public function users()
    {
        return $this->belongsToMany('App\User')->using('App\UserRole');
    }
}
```

优化 Redis 集群支持

5.4 版本以前，在同一个应用中不能同时定义 Redis 链接指向单个主机和集群，在 Laravel 5.4 中可以在同一个应用中定义 Redis 链接指向多个主机和多个集群。

更多关于 Laravel 中 Redis 的信息，查看 [Redis 文档](#)。

迁移默认字符长度

Laravel 5.4 默认使用 `utf8mb4` 字符编码，该编码支持对 "emojis" 表情在数据库进行存储。如果你从 Laravel 5.3 升级，不需要对字符编码做切换。

如果你选择手动切换到这个字符编码，并且运行小于 MySQL 5.7.7 release 版本的数据库，你可能还需要手动配置迁移命令生成默认字符长度，你可以在 `AppServiceProvider` 中调用

`Schema::defaultStringLength` 方法来实现这一配置：

```
use Illuminate\Support\Facades\Schema;

/**
 * 引导任何应用服务
 *
 * @return void
 */
public function boot()
{
    Schema::defaultStringLength(191);
}
```

Laravel 5.3

Laravel 5.3 在 5.2 基础上进行了优化，新特性包括以下：

- 消息通知 [Laravel Notifications](#)；
- 事件广播 [Laravel Echo](#)；
- OAuth2 授权认证 [Laravel Passport](#)；
- 全文搜索引擎 [Laravel Scout](#)；
- Laravel Elixir 开始支持 Webpack；
- Mail 操作类 [Laravel Mailable](#)；
- `web` 和 `api` 的路由分离；
- 基于闭包的控制台命令；
- 更加易用的辅助函数，用于存储上传的文件；
- 支持 POPO 和单动作控制；
- 优化默认的前端脚手架，等等。

消息通知

{video} Laracasts 上关于此新特性的免费视频 [video tutorial](#)。

[Laravel Notifications](#) 提供了简单、优雅的 API 支持你通过不同的渠道发送通知，例如电子邮件、Slack、手机短信等等。

例如，你可以定义一个单据，当该单据被付款后，则通过邮件和手机短信发送提醒通知：

你可用通过下面这个简单的方法来实现：

```
$user->notify(new InvoicePaid($invoice));
```

[Laravel 社区](#) 已经为通知系统编写了各式的驱动，甚至包括对 iOS 和 Android 通知的支持，更多关于通知系统的信息，请查看 [完整的文档](#)。

WebSockets / 事件广播

事件广播在之前版本的 Laravel 中已经存在，Laravel 5.3 现支持对私有和已存在的 WebSocket 频道添加频道级认证：

```
/*
 * 频道认证
 */
Broadcast::channel('orders.*', function ($user, $orderId) {
    return $user->placedOrder($orderId);
});
```

Laravel Echo 是一个可通过 NPM 安装的全新的 JavaScript 包，会随 Laravel 5.3 一起发布。Echo 提供了简单、优雅的 API 接口，支持你在 JavaScript 客户端应用中，订阅频道和监听服务器端事件。

Echo 提供的支持包括 [Pusher](#) 以及 [Socket.io](#)：

```
Echo.channel('orders.' + orderId)
    .listen('ShippingStatusUpdated', (e) => {
        console.log(e.description);
    });
});
```

除了订阅到传统频道上，Laravel Echo 也让频道的监听与管理变得更加简单：

```
Echo.join('chat.' + roomId)
    .here((users) => {
        //
    })
    .joining((user) => {
        console.log(user.name);
    })
    .leaving((user) => {
        console.log(user.name);
    });
});
```

了解更多关于 Echo 和事件广播的信息，请查阅 [完整文档](#)。

Laravel Passport (OAuth2 认证服务)

{video} Laracasts 上关于此功能的免费视频 [video tutorial](#)。

Laravel 5.3 的 Passport 让 API 认证变得简单。Laravel Passport 可以让你在几分钟内为应用程序创建一个完整的 OAuth2 认证服务，Passport 基于 Alex Bilbie 的 [League OAuth2 server](#) 实现。

Passport 让发放 OAuth2 令牌 (Access Token) 变得轻松，你还可以允许用户通过 Web 界面创建 个人访问令牌 。

为了方便提高开发效率，Passport 内置了一个 Vue 组件，该组件提供了 OAuth2 后台界面功能，允许用户创建客户端、撤销访问令牌，以及更多其他功能：

```
<passport-clients></passport-clients>
<passport-authorized-clients></passport-authorized-clients>
<passport-personal-access-tokens></passport-personal-access-tokens>
```

如果你不想使用 Vue 组件，你可以自由的定制用于管理客户端和访问令牌的前端及后台。Passport 提供了一个简单的 JSON API，你可以在前端使用任何 JavaScript 框架与之集成。

Passport 还提供了方便的 API 让你定制「Token 访问域」：

```
Passport::tokensCan([
    'place-orders' => 'Place new orders',
    'check-status' => 'Check order status',
]);
```

此外，Passport 还包含了一个用于检查「Token 访问域」访问权限的中间件：

```
Route::get('/orders/{order}/status', function (Order $order) {
    // 检查令牌是否拥有 "check-status" 访问域
})->middleware('scope:check-status');
```

最后，Passport 还支持从 JavaScript 应用访问你的 API，而不必担心访问令牌传输。

Passport 通过加密 JWT cookies 和同步「CSRF 令牌」来实现此功能，让你专注于业务开发。

更多关于 Passport 信息，请查看 [完整文档](#)。

搜索系统 Laravel Scout

Laravel Scout 提供了一个简单的、基于驱动的、针对 Eloquent 模型的全文搜索解决方案。

通过模型观察者，Scout 会自动同步更新 Eloquent 的搜索索引，目前，Scout 使用 Algolia 驱动，你可以自由的编写自己驱动来扩展 Scout。

你只需要添加 Searchable trait 到模型中，就能让模型支持搜索：

```
<?php

namespace App;

use Laravel\Scout\Searchable;
use Illuminate\Database\Eloquent\Model;
```

```
class Post extends Model
{
    use Searchable;
}
```

在你在模型中添加 trait 以后，数据会在 `save` 的时候自动保持同步：

```
$order = new Order;

// ...

$order->save();
```

在模型被成功索引以后，可以很轻松的使用全文搜索，你甚至可以为索引的结果进行分页操作：

```
return Order::search('Star Trek')->get();

return Order::search('Star Trek')->where('user_id', 1)->paginate();
```

更多 Scout 功能，请查阅 [Scout 的完整文档](#)。

Mailable 对象

{video} Laracasts 上关于此功能的免费视频 [video tutorial](#)。

Laravel 5.3 Mailable 是一个崭新的 Mail 操作类，通过一种更加优雅的方式发送邮件，而不再需要在闭包中自定义邮件信息。

例如，定义一个简单的邮寄对象用作发送欢迎邮件：

```
class WelcomeMessage extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * 新建消息
     *
     * @return $this
     */
    public function build()
    {
        return $this->view('emails.welcome');
    }
}
```

Mailable 对象被创建以后，你可以使用一个简单、优雅的 API 将其发送给用户：

```
Mail::to($user)->send(new WelcomeMessage);
```

Mailable 还支持队列操作，只需要在类声明里实现 `ShouldQueue` 即可：

```
class WelcomeMessage extends Mailable implements ShouldQueue
{
    //
}
```

更多关于 Mailable 的信息，请查看 [完整文档](#)。

存储上传文件

{video} Laracasts 上关于此功能的免费视频 [video tutorial](#)。

存储用户上传文件，在 Web 开发中是一个很常见的任务。

Laravel 5.3 提供了一个便捷的 `store` 方法，只需要对上传文件对象调用此方法，并传参准备存储的路径即可：

```
/**
 * 更新用户头像
 *
 * @param Request $request
 * @return Response
 */
public function update(Request $request)
{
    $path = $request->file('avatar')->store('avatars', 's3');

    return $path;
}
```

更多上传文件信息，请查看 [完整文档](#)。

Webpack 和 Laravel Elixir

Laravel Elixir 6.0 与 Laravel 5.3 共同发布，内置了 Webpack 和 Rollup JavaScript。

默认情况下，Laravel 5.3 的 `gulpfile.js` 使用 Webpack 来编译你的 JavaScript 文件：

```
elixir(mix => {
    mix.sass('app.scss')
        .webpack('app.js');
});
```

完整文档请见 [Laravel Elixir](#)。

前端架构

{video} Laracasts 上关于此功能的免费视频 [video tutorial](#)。

Laravel 5.3 提供了一个更加现代的前端架构。这主要会影响 `make:auth` 命令生成认证相关的前端脚手架代码，不再从 CDN 中加载前端资源，所有依赖被定义在默认的 `package.json` 文件中，你可以自行修改。

此外，支持单文件的 [Vue 组件](#) 现在直接开箱即用，`resources/assets/js/components` 目录下包含了一个简单的示例 `Example.vue`，新的 `resources/assets/js/app.js` 用来配置 JavaScript 类库依赖和 Vue 子模块。

这种架构对开始开发现代的、强大的 JavaScript 应用提供了更好的支持，而不需要要求应用使用任何特定 JavaScript 或者 CSS 框架。

更多信息，请查看对应文档 [前端文档](#)。

路由文件

默认情况下，新安装的 Laravel 5.3 应用在新的顶级目录 `routes` 下包含了 `web.php` 和 `api.php` 两个 HTTP 路由文件，你也可以按照此方法自行扩展。

API 相关的路由在 `RouteServiceProvider` 中指定了自动添加 `api` 前缀。

闭包控制台命令

除了通过命令类定义之外，Artisan 命令现支持在 `app\Console\Kernel.php` 文件中使用简单闭包的方式定义。

在新安装的 Laravel 5.3 应用中，`commands` 方法会加载 `routes/console.php` 文件，从而允许你基于闭包、以路由风格定义控制台命令：

```
Artisan::command('build {project}', function ($project) {
    $this->info('Building project...');
});
```

更多信息请参见 [Artisan 文档](#)。

Blade 中的 `$loop` 魔术变量

{video} Laracasts 上关于此功能的免费视频 [video tutorial](#)。

当我们在 Blade 模板中循环遍历的时候，`$loop` 魔术变量将会在循环中生效。通过该变量可以访问很多有用的信息，比如当前循环索引值，以及当前循环是第一个还是最后一个：

```

@foreach ($users as $user)
@if ($loop->first)
    This is the first iteration.
@endif

@if ($loop->last)
    This is the last iteration.
@endif

<p>This is user {{ $user->id }}</p>
@endforeach

```

更多信息请查看 [Blade 文档](#).

Laravel 5.2

Laravel 5.2 在 Laravel 5.1 的基础上进行了优化，新特性包括以下：

- 支持更多样的用户认证驱动；
- 隐式数据模型绑定；
- 简化 Eloquent 全局作用域；
- 内置用户认证脚手架支持；
- 中间件组；
- 访问频率限制中间件；
- 数组认证的优化等

用户认证驱动 / "多认证系统"

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。

Laravel 5.2 对此进行了改进，你可以定义多个认证驱动，还支持多个可认证的数据模型以及用户表，并且可以独立控制其认证。

例如，如果你的应用中有一张「admin」数据库用户表，一张「student」数据库用户表（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` 方法来实现后台用户和学生用户的独立登录而不相互影响。

用户认证脚手架

此前 Laravel 后端认证处理已经是相当容易了，现在 Laravel 5.2 提供了一个更加便捷、快速的方法来创建前台认证视图，只需要简单的在终端执行 `make:auth` 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。该命令还会顺带在路由文件中增加对应的授权路由。

{note} 该功能特性只能用于新创建的应用，不能用于升级后的应用。

隐式数据模型绑定

隐式模型绑定使得在路由和控制器中注入模型实例更加便捷。例如，假设你定义了一个如下的路由：

```
use App\User;

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

在 Laravel 5.1 中，你通常需要通过 `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 判断自动注入模型，从而允许你快速访问需要的模型实例。

如路由参数片段 `{user}` 匹配到 路由闭包 或 控制器方法 中对应参数 `$user`，且类型提示为 Eloquent 数据模型的话，Laravel 将会自动注入该模型。

更多隐式模型绑定信息，请查看 [HTTP 路由模型绑定部分](#)。

中间件群组

中间件群组允许你将多个路由中间件组织到单一、方便的键（即群组名称）下面，从而可以为某个路由一次指派多个中间件。例如，在同一个应用中构建 Web UI 或 API 时，这一特性很有用。例如，你可以将 `session` 及 `CSRF` 路由组合成一个 `web` 群组，并将访问频率限制分组到 `api` 群组。

实际上，默认的 Laravel 5.2 应用结构采用的正是这种方法。例如，在默认的 `App\Http\Kernel.php` 文件中你会看到如下内容：

```
/**
 * 路由中间件群组
 *
 * @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',
],
];
```

然后，`web` 组可以这样指定给路由：

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

默认的，所有的 `app/Http/routes.php` 中的路由已经在 `RouteServiceProvider` 的 `mapWebRoutes` 方法中指定了 `web` 中间件组，所以你不必重复指定。

访问频率限制

框架现在内置了一个新的访问频率限制中间件，允许你轻松控制给定 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 中，可轻松验证表单中的每一个数组输入字段。例如，要验证指定数组输入字段中每一个 `email` 是否唯一，可以这么实现：

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

同样的，你可以使用「`*`」符号来指定要验证数组字段，自定义验证数组字段的错误消息提醒：

```
'custom' => [
    'person.*.email' => [
        'unique' => '用户的 Email 必须是唯一的',
    ]
],
```

Bail 认证规则

Laravel 5.2 新添加了一个 `bail` 认证规则，此规则会在第一个失败认证后停止后面的其他认证检查。例如：你想在 `integer` 数值检查失败后停止对 `unique` 唯一性的检查：

```
$this->validate($request, [
    'user_id' => 'bail|integer|unique:users'
]);
```

Eloquent 全局作用域优化

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

更多关于全局作用域的使用，请查阅 [Eloquent 文档](#)。

Laravel 5.1.11

Laravel 5.1.11 推出了内置的 [授权](#) 功能！利用回调和授权策略类，能更方便的组织应用程序的授权逻辑。

更多的信息请参考 [授权的文档](#)。

Laravel 5.1.4

Laravel 5.1.4 增加了简单的登录限制功能。查阅 [认证的文档](#) 以获取更多信息。

Laravel 5.1

Laravel 5.1 由 Laravel 5.0 改进而成，变更包括但是不限于：

- 采用 PSR-2 规范
- 支持添加事件广播
- 中间件参数
- Artisan 的改进等等。

PHP 5.5.9+

由于 PHP 5.4 将在九月「结束寿命」，PHP 开发团队不再提供安全性更新，所以 Laravel 要求 PHP 5.5.9 或更高的版本。

PHP 5.5.9 同时也是最新版本的 PHP 函数库，像是 Guzzle 及 AWS SDK 需要的最小版本需要。

LTS

Laravel 5.1 是 Laravel 生态系统中第一个 长期支持 版本。Laravel 5.1 会获得两年的 BUG 修复及三年的安全性修复，此策略也使 Laravel 能更好的服务于较大型的企业客户及消费者。

PSR-2

[PSR-2 代码风格指南](#) 已经被 Laravel 框架采用为默认的代码风格指南。此外，所有的生成器都已进行更新，生成的文件将兼容 PSR-2 规范。

文档

Laravel 文档的每一页已被精心审阅，并得到显著的改善。所有的代码例子也进行了严密检查，使其有更高的上下文关联性。

事件广播

WebSockets 技术越来越多的被现代 Web 应用使用，当服务器上一些数据更新，WebSocket 会实时发送一个消息给客户端，实现即时更新用户状态功能。

Laravel 的事件广播机制很好的支持了此类应用的开发，广播事件允许服务器端代码和 JavaScript 框架间分享相同的事件名称。

了解更多关于事件广播，请查阅 [事件的文档](#)。

中间件参数

中间件支持接收自定义传参，例如要在运行特定操作之前，检查当前登录的用户是否具备「某角色」，可以创建 `RoleMiddleware` 来接收角色名称作为传参：

```
<?php

namespace App\Http\Middleware;

use Closure;

class RoleMiddleware
{
    /**
     * 运行请求过滤。
     *
     * @param \Illuminate\Http\Request $request
     * @param Closure $next
     * @param string $role
     * @return mixed
     */
    public function handle($request, Closure $next, $role)
    {
        if (! $request->user()->hasRole($role)) {
            // 重定向...
        }

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

```

    }
}
```

在路由中使用冒号 `:` 来区隔中间件名称与参数，多个参数可使用逗号作为分隔：

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

关于中间件的更多信息，请查阅 [中间件的文档](#)。

测试改进

Laravel 内置的测试功能已得到显著的改善，新版本提供了简明的接口与应用程序进行交互，响应检查也变得更加轻松。

例如下方的测试：

```
public function testNewUserRegistration()
{
    $this->visit('/register')
        ->type('Taylor', 'name')
        ->check('terms')
        ->press('Register')
        ->seePageIs('/dashboard');
}
```

关于测试的更多信息，请查阅 [测试的文档](#)。

模型工厂

Laravel 的 [模型工厂](#) 提供一种简单的方式来创建仿真 Eloquent 模型。

在为 Eloquent 模型定义一组「默认」填充字段后，即可为测试，或者数据填充生成测试模型实例。

另外，模型工厂支持使用 [Faker](#) 来生成随机数据：

```
$factory->define(App\User::class, function ($faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->email,
        'password' => str_random(10),
        'remember_token' => str_random(10),
    ];
});
```

更多关于模型工厂的信息，请查阅 [它的文档](#)。

Artisan 的改进

Artisan 命令现支持类似于命名路由的定义，以简单易懂的形式来定义命令行的参数及选项。

举个例子，你可以定义一个简单的命令及它的参数和选项，如下：

```
/**
 * 命令行的名字及署名。
 *
 * @var string
 */
protected $signature = 'email:send {user} {--force}';
```

更多关于定义 Artisan 命令的消息，请参考 [Artisan 的文档](#)。

文件夹结构

为了更方便理解，`app/Commands` 目录已经被更名为 `app/Jobs`。

此外，`app/Handler` 目录已经被合并成一个只包含事件侦听器的 `app/Listeners` 目录中。

这不是一个重大的改变，你不必更新成新的文件夹结构也能使用 Laravel 5.1。

加密

在 Laravel 之前的版本，加密是通过 `mcrypt` PHP 扩展进行处理。不过，从 Laravel 5.1 起，加密将采用更积极维护的 `openssl` 扩展进行处理。

Laravel 5.0

Laravel 5.0 引进了新的应用程序架构。新架构允许 Laravel 创建更加健壮的应用程序，新架构全面采用新的自动加载标准（PSR-4）。

以下是一些主要变化：

新的目录结构

旧的 `app/models` 目录已经完全被移除。对应的，你所有的代码都放在 `app` 目录下。

默认情况下使用 `App` 命名空间。可以使用 `app:name` Artisan 命令对默认命名空间进行修改。

控制器、中间件，以及表单请求（Laravel 5.0 中新型态的类），分门别类的放在 `app/Http` 目录下，因为他们都与应用程序的 HTTP 传输层相关。除了一个路由设置的文件外，所有中间件现都分开为独自的类文件。

`app/Providers` 目录取代了旧版 Laravel 4.x `app/start` 里的文件。这些服务提供者为应用程序提供了各种引导功能，像是错误处理，日志纪录，路由加载等等。当然，你可以任意的创建新的服务提供者。

应用程序的语言文件和视图都被移到 `resources` 目录下。

Contracts

所有 Laravel 组件实现所用的接口都放在 `illuminate/contracts` 文件夹中，他们没有其他依赖。这些方便集成的接口，让依赖注入变得低耦合，可简单作为 Laravel Facades 的替代选项。

更多关于 contracts 的信息，参考 [完整文档](#)。

路由缓存

如果你的应用程序全部使用控制器路由，新的 `route:cache` Artisan 命令可大幅度地优化路由注册寻找速度。

这对于拥有 100 个以上路由规则的应用程序来说很有用，可以大幅度地 加快应用程序路由部分的处理速度。

路由中间件

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

更多关于中间件的信息，参考 [完整文档](#)。

控制器方法注入

除了之前有的类的构造函数注入外，你现在可以在控制器方法中使用依赖注入。服务容器 会自动注入依赖，即使路由包含了其它参数也不成问题：

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

认证基本架构

认证系统默认包含了用户注册，认证，以及重设密码的控制器，还有对应的视图，视图文件存放在 `resources/views/auth`。

除此之外，「users」数据表迁移也默认包含在框架中。这些简单的资源，可以让你把心思放在产品开发上，而不用陷在编写认证模板的泥潭。

认证相关的视图可以通过 `auth/login` 以及 `auth/register` 路由访问。

`App\Services\Auth\Registrar` 会负责处理用户认证和注册用户的相关逻辑。

事件对象

你现在可以将事件定义成对象，而不是仅使用字符串。例：

```
<?php

class PodcastWasPurchased
{
    public $podcast;

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

这个事件可以像一般使用那样被派发：

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

当然，你的事件处理会收到事件的对象而不是数据的列表：

```
<?php

class ReportPodcastPurchase
{
    public function handle(PodcastWasPurchased $event)
    {
        //
    }
}
```

更多关于使用事件的信息，参考 [完整文档](#)。

命令及队列

除了 Laravel 4 形式的队列任务，Laravel 5 还支持命令对象直接作为队列任务。这些命令放在 `app/Commands` 目录下。下面是个例子的命令：

```
<?php

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

```

use SerializesModels;

protected $user, $podcast;

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

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

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

```

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

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

当然，你也可以将命令视为同步运行（而不会被放到队列里）的任务。事实上，「命令总线」是个不错的设计模式，可以封装应用程序需要运行的复杂任务。更多相关的信息，参考 [command bus](#) 文档。

数据库队列

`database` 队列驱动现在已经包含在 Laravel 中了，提供了简单的本地端队列驱动，除了数据库相关软件外不需安装其它扩展包，完全开箱即用。

Laravel 调度器

在过去，开发者是在 `crontab` 里配置任务调度的。然而，这是件很头痛的事情，因为你的命令行调度不在版本控制中，并且必须登录到服务器里才能添加新的 Cron 设置。

Laravel 命令行调度的存在，就是为了改变这一情况。

命令行调度系统让你在 Laravel 里定义富有表达性的命令调度，而且只需要在服务器里设置一个 Cron 设置即可。

看起来如下：

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

参考 [完整文档](#) 学习所有调度相关知识。

Tinker 与 Psysh

`php artisan tinker` 命令现在使用 Justin Hileman 的 [Psysh](#)，一个 PHP 更强大的 REPL。如果你喜欢 Laravel 4 的 Boris，你也会喜欢上 Psysh。更好的是，它可以在 Windows 上运行！

赶快尝试下吧：

```
php artisan tinker
```

DotEnv

比起一堆令人困惑的、嵌套的环境配置文件，Laravel 5 现在使用了 Vance Lucas 的 [DotEnv](#)。

这个扩展包提供了超级简单的方式管理配置文件，并且让 Laravel 5 环境侦测变得轻松。更多的细节，参考完整的 [配置文件文档](#)。

Laravel Elixir

Jeffrey Way 的 Laravel Elixir 提供了一个流畅、口语化的接口，可以编译以及合并静态资源。如果你曾经因为学习 Grunt 或 Gulp 而被吓到，不必再害怕了。Elixir 让使用 Gulp 编译 Less、Sass 及 CoffeeScript 变得简单。它甚至可以帮你运行测试！

更多关于 Elixir 的信息，参考 [完整文档](#)。

Laravel Socialite

Laravel Socialite 是可选的，兼容 Laravel 5.0 以上的 OAuth 认证扩展包。目前 Socialite 支持 Facebook、Twitter、Google 以及 GitHub。它写起来是这样的：

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

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

不再需要花上数小时编写 OAuth 的认证流程，只要几分钟！查看 [完整文档](#) 里有所有的细节。

文件系统集成

Laravel 现在包含了强大的 [Flysystem 文件系统](#)（一个文件系统的抽象函数库）。

文件系统以抽象的概念，把本地端文件系统、Amazon S3 和 Rackspace 云存储集成在一起，统一且优雅的 API！

现在要将文件存到 Amazon S3 相当简单：

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

更多关于 Laravel 文件系统集成，参考 [完整文档](#)。

Form Requests

Laravel 5.0 引进了 form requests，是继承自 `Illuminate\Foundation\Http\FormRequest` 的类。这些 request 对象可以和控制器方法依赖注入结合使用，提供一个不需模版的方法，来验证用户输入。让我们深入点，看一个 `FormRequest` 的例子：

```
<?php

namespace App\Http\Requests;

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

    public function authorize()
    {
        return true;
    }
}
```

定义好类后，我们可以在控制器动作里使用类型提示进行依赖注入：

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

```
}
```

当 Laravel 的服务容器辨别出要注入的类是个 `FormRequest` 实例，该请求将会被自动验证。意味着，框架会自动根据你在 form request 类里自定的规则，对请求进行检验。当控制器动作调用时，你可以安全的假设 HTTP 的请求输入已被验证过。

甚至，若这个请求验证不通过，一个 HTTP 重定向（可以自定义），会自动发出，错误消息可以被闪存到 session 中或是转换成 JSON 返回。表单验证再简单不过了。更多关于 `FormRequest` 验证，请参考 [文档](#)。

简易控制器请求验证

Laravel 5 基底控制器包含一个 `ValidatesRequests` trait。这个 trait 包含了一个简单的 `validate` 方法可以验证请求。如果对你的应用程序来说 `FormRequests` 太复杂了，可以考虑使用手动验证方法：

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

如果验证失败，会抛出异常以及返回适当的 HTTP 响应到浏览器。验证错误信息会被闪存到 session 里！而如果请求是 AJAX 请求，Laravel 会自动返回 JSON 格式的验证错误信息。

更多关于这个新方法的信息，参考 [这个文档](#)。

新的生成器

为了响应新的应用程序默认架构，框架新增了许多 Artisan generator 命令。使用 `php artisan list` 查看完整的命令列表。

配置文件缓存

你现在可以通过 `config:cache` 命令将所有的配置文件缓存在单个文件中，这样在一定程度上会加快框架的启动效率。

Symfony VarDumper

广为使用的 `dd` 辅助函数，用作在调试时输出变量信息，现采用令人惊艳的 Symfony VarDumper 扩展包。它提供了颜色标记的输出，甚至数组可以自动缩合。在项目中试试下列代码：

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

Laravel 4.2

此发行版本的完整的变更列表可以通过运行 `php artisan changes` 命令来获取，或者 [Github 上的更动纪录](#)。此纪录仅含括主要更新和此发行的更动部分。

附注: 在 4.2 开发期间，许多小的 BUG 修正与功能强化被整合至各个 4.1 的子发行版本中。所以，也请一并检查 Laravel 4.1 版本的更新列表。

PHP 5.4 需求

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

Laravel Forge

Laravel Forge，一个网页应用程序，提供一个简单的接口让你创建管理云端上的 PHP 服务器，像是 Linode、DigitalOcean、Rackspace 和 Amazon EC2。

支持自动化 nginx 设置、SSH 密钥管理、Cron job 自动化、通过 NewRelic & Papertrail 服务器监控、「推送部署」、Laravel queue worker 设置等等。Forge 提供最简单且更实惠的方式来部署所有你的 Laravel 应用程序。

默认 Laravel 4.2 的安装里，`app/config/database.php` 配置文件已为 Forge 设置完成，让你更方便的完成新平台上的全新应用程序的部署。

关于 Laravel Forge 的更多信息可以在 [官方 Forge 网站](#) 上找到。

Laravel Homestead

Laravel Homestead 是一个健全的 Laravel 和 PHP 应用程序 Vagrant 环境。软件依赖都已提前准备好，可以极快的被启用。

Homestead 包含 Nginx 1.6、PHP 5.5.12、MySQL、Postres、Redis、Memcached、Beanstalk、Node、Gulp、Grunt 和 Bower。Homestead 包含一个简单的 `Homestead.yaml` 配置文件，允许你在单个封装包中管理多个 Laravel 应用程序。

默认的 Laravel 4.2 安装中包含的 `app/config/local/database.php` 配置文件已经为你配置好了 Homestead 的数据库连接。让 Laravel 初始化安装与设置更为方便。

官方文档已经更新并包含在 [Homestead 文档](#) 中。

Laravel 收银台

Laravel 收银台是一个简单、具表达性的资源库，用来管理 Stripe 的订阅服务。虽然安装此组件是可选的，我们仍然将收银台文档包含在主要 Laravel 文档中。新版本的收银台带来了数个错误修正、多货币支持还有支持了最新的 Stripe API。

Queue Workers 常驻程序

Artisan `queue:work` 命令现在支持 `--daemon` 参数让 worker 可以作为「常驻程序」启用。代表 worker 可以持续的处理队列工作，而不需要重启框架。这让一个复杂的应用程序对 CPU 的使用率有显著的降低。

更多关于 Queue Workers 常驻程序信息请详阅 [queue 文档](#)。

Mail API Drivers

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

软删除 Traits

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

更多关于软删除的文档请见: [Eloquent documentation](#)。

更为方便的 认证(auth) & Remindable Traits

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

"简易分页"

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

迁移确认

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

Laravel 4.1

完整更动列表

此发行版本的完整更动列表，可以在版本 4.1 的安装中命令行运行 `php artisan changes` 获取，或者浏览 [Github 更新文件中](#) 中了解。其中只记录了该版本比较主要的强化功能和更动。

新的 SSH 组件

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

新的 `php artisan tail` 命令就是使用这个新的 SSH 组件。更多的信息，请参阅 [tail 命令集文档](#)。

Boris In Tinker

如果你的系统支持 [Boris REPL](#)，`php artisan thinker` 命令将会使用到它。系统中也必须先行安装好 `readline` 和 `pcntl` 两个 PHP 扩展包。如果你没这些扩展包，从 4.0 之后将会使用到它。

Eloquent 强化

Eloquent 添加了新的 `hasManyThrough` 关系链。想要了解更多，请参见 [Eloquent 文档](#)。

一个新的 `whereHas` 方法也同时登场，他将允许 [检索基于关系模型](#)。

数据库读写分离

Query Builder 和 Eloquent 目前通过数据库层，已经可以自动做到读写分离。更多的信息，请参考 [文档](#)。

队列排序

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

失败队列作业处理

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

缓存标签

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

更具弹性的密码提醒

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

强化路由引擎

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

强化 Session 引擎

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

Doctrine DBAL

如果你有在你的迁移中使用到 `renameColumn`，之后你必须在 `composer.json` 里加 `doctrine/dbal` 进依赖扩展包中。此扩展包不再默认包含在 Laravel 之中。

Laravel 升级索引

- [从 5.3 升级到 5.4.0](#)

从 5.3 升级到 5.4.0

预计升级耗时: 1-2 小时

{note} 我们尽量罗列出每一个不兼容的变更。但因为其中一些不兼容变更只存在于框架很不起眼的地方, 事实上只有一小部分会真正影响到你的应用程序。

更新依赖

在 `composer.json` 文件中将你的 `laravel/framework` 更新为 `5.4.*`。此外, 你应该更新你的 `phpunit/phpunit` 依赖关系到 `~5.7`。

删除编译的服务文件

如果存在, 你可以删除 `bootstrap/cache/compiled.php` 文件。它不再被框架使用。

刷新缓存

升级所有软件包之后, 你应该运行 `php artisan view:clear` 来避免与删除 `Illuminate\View\Factory::getFirstLoop()` 相关的 Blade 错误。此外, 你需要运行 `php artisan route:clear` 来刷新路由缓存。

Laravel Cashier

Laravel Cashier 已经兼容 Laravel 5.4。

Laravel Passport

为了兼容 Laravel 5.4 和 [Axios](#) JavaScript 库, Laravel Passport 已经发布了 `2.0.0`。如果你是从 Laravel 5.3 升级, 并使用了预置的 Passport Vue 组件, 你应该确保 Axios 库在你的应用程序中作为 `axios` 全局可用。

Laravel Scout

Laravel Scout `3.0.0` 已经发布, 以提供与 Laravel 5.4 的兼容性。

Laravel Socialite

Laravel Socialite `3.0.0` 已经发布, 以提供与 Laravel 5.4 的兼容性。

Laravel Tinker

为了继续使用 `tinker` Artisan 命令，你还应该安装 `laravel/tinker` 包：

```
composer require laravel/tinker
```

一旦软件包被安装，你应该添加 `Laravel\Tinker\TinkerServiceProvider::class` 到 `config/app.php` 配置文件中的 `providers` 数组。

Guzzle

Laravel 5.4 需要 Guzzle 6.0 或者更高版本。

授权

`getPolicyFor` 方法

在之前的版本中，调用 `Gate::getPolicyFor($class)` 方法的时候，如果没有找到对应的 policy，会抛出异常。现在，该方法在给定类中找不到 policy 的时候会返回 `null`。如果你直接调用这个方法，请确保你在重构代码中新增了对 `null` 的检查：

```
$policy = Gate::getPolicyFor($class);

if ($policy) {
    // code that was previously in the try block
} else {
    // code that was previously in the catch block
}
```

Blade

`@section` 转码

在 Laravel 5.4，传递给 section 的内联内容会自动进行转码：

```
@section('title', $content)
```

如果你想要在 section 中渲染原生内容，你必须使用传统的「长形式」声明该 section。

```
@section('title')
{!! $content !!}
@stop
```

引导程序

如果你在 HTTP 或 Console 启动类中手动覆盖了 `$bootstrappers` 数组，需要将 `DetectEnvironment` 重命名为 `LoadEnvironmentVariables`。

广播

频道模型绑定

在 Laravel 5.3 中定义频道名称占位符时，使用 `*` 字符。在 Laravel 5.4 中，你应该使用 `{foo}` 风格来定义这些占位符，如路由：

```
Broadcast::channel('App.User.{userId}', function ($user, $userId) {
    return (int) $user->id === (int) $userId;
});
```

集合

`every` 方法

`every` 方法的功能被合并到 `nth` 方法中以匹配被 Lodash 定义的方法名。

`random` 方法

调用 `$collection->random(1)` 现在会返回一个包含单个item的新的集合实例。在之前版本中，这个方法会返回单个对象。如果没有提供参数，该方法将只返回单个对象。

容器

别名 via `bind` / `instance`

在之前的 Laravel 版本中，你可以将数组作为第一个参数传递给 `bind` 或者 `instance` 方法来注册别名：

```
$container->bind(['foo' => FooContract::class], function () {
    return 'foo';
});
```

然而，这种行为已经在 Laravel 5.4 中删除。要注册别名，你应该使用 `alias` 方法：

```
$container->alias(FooContract::class, 'foo');
```

使用 \ 绑定类

将不再支持通过带有 \ 的类名绑定类到容器。因为该功能需要在底层容器中进行大量的字符串格式化调用。取而代之的，只需要通过下面这种不带 \ 的方式绑定即可：

```
$container->bind('Class\\Name', function () {
    //
});
```

```
$container->bind(ClassName::class, function () {
    //
});
```

make 方法参数

容器的 `make` 方法不再接受第二个数组参数，因为该特性表明了代码的坏味道。我们需要以更加直观的方式来构造对象。

解决回调

容器的 `resolving` 和 `afterResolving` 方法现在必须提供一个类名或绑定键作为方法的第一个参数：

```
$container->resolving('Class\\Name', function ($instance) {
    //
});

$container->afterResolving('Class\\Name', function ($instance) {
    //
});
```

share 方法已删除

`share` 方法已经从容器中删除。这是一个遗留的方法，多年未被列出来。如果你使用这个方法，你应该开始使用 `singleton` 方法：

```
$container->singleton('foo', function () {
    return 'foo';
});
```

控制台

The `Illuminate\Console\AppNamespaceDetectorTrait` Trait

如果你直接引用 `Illuminate\Console\AppNamespaceDetectorTrait` trait，更新你的代码替换为 `Illuminate\Console\DetectsApplicationNamespace`。

数据库

自定义连接

如果你之前为 `db.connection.{driver-name}` 绑定了一个服务容器绑定以便解析自定义数据库连接实例，现在需要在 `AppServiceProvider` 的 `register` 方法中使用 `Illuminate\Database\Connection::resolverFor` 方法：

```
use Illuminate\Database\Connection;

Connection::resolverFor('driver-name', function ($connection, $database, $prefix, $config) {
    //
});
```

Fetch 模式

Laravel不再支持在配置文件中定制PDO「fetch mode」的功能。取而代之，`PDO::FETCH_OBJ` 一直可以使用。如果你仍然想要为你的应用程序自定义提取模式，你可以监听新的 `Illuminate\Database\Events\StatementPrepared` 事件：

```
Event::listen(StatementPrepared::class, function ($event) {
    $event->statement->setFetchMode(...);
});
```

Eloquent

日期转换

日期转换 `date` 现在会将日期转化为 `Carbon` 对象并调用该对象上的 `startOfDay` 方法，如果你想保留日期的时间部分，需要使用 `datetime` 转换。

外键约束

在定义关联关系的时候如果外键没有被显式指定，Eloquent 将会使用关联模型的表名和主键名来构建外键。这适用于大多数应用，这不是行为的改变。例如：

```
public function user()
{
    return $this->belongsTo(User::class);
}
```

就像以前的 Laravel 版本一样，这种关系通常使用 `user_id` 作为外键。但是，如果你重写 `User` 模型的 `getKeyName` 方法，情况就会变得不一样，例如：

```
public function getKeyName()
{
    return 'key';
}
```

在这种情况下，Laravel 会以你自定义的主键名为准，外键名将会是 `user_key` 而不是 `user_id`。

一对一 / 多的 `createMany`

`hasOne` or `hasMany` 关系的 `createMany` 方法现在返回一个集合对象而不是一个数组。

相关模型连接

相关模型现在将使用与父模型相同的连接。例如，你执行以下查询：

```
User::on('example')->with('posts');
```

Eloquent 将查询 `example` 连接上的 `posts` 表，而不是默认的数据库连接。如果你想从默认连接读取 `posts` 关系，你应该明确地设置模型的连接到你的应用程序的默认连接。

create & forceCreate 方法

`Model::create` & `Model::forceCreate` 方法已经移动到 `Illuminate\Database\Eloquent\Builder` 类，以便为在多个连接上创建模型提供更好的支持。但是，如果要在自己的模型中扩展这些方法，则需要修改实现以在构建器上调用 `create` 方法。例如：

```
public static function create(array $attributes = [])
{
    $model = static::query()->create($attributes);

    // ...

    return $model;
}
```

hydrate 方法

如果你现在传递自定义的连接名到该方法，需要使用 `on` 方法：

```
User::on('connection')->hydrate($records);
```

hydrateRaw 方法

`Model::hydrateRaw` 方法已经被重命名为 `fromQuery`，如果你传递自定义的连接名到该方法，需要使用 `on` 方法：

```
User::on('connection')->fromQuery('...');
```

whereKey 方法

`whereKey($id)` 方法现在将为给定的主键值添加一个「`where`」子句。在之前的版本中，这将属于动态「`where`」子句构建器，并为「`key`」列添加「`where`」子句。如果你使用 `whereKey` 方法为 `key` 列动态添加一个条件，你现在应该使用 `where('key', ...)`。

factory Helper

调用 `factory(User::class, 1)->make()` 或 `factory(User::class, 1)->create()` 现在将返回一个集合。以前，这将返回单个模型。如果没有传入数量的话，返回的仍然只是单个模型。

事件

Contract 变更

如果你在应用程序或者包中手动实现 `Illuminate\Contracts\Events\Dispatcher` 接口，则应该将 `fire` 方法重命名为 `dispatch`。

事件优先级

不再支持事件处理器「优先级」，从未记录的功能通常表示事件功能的滥用。相反，请考虑使用一系列同步方法调用。或者，你可以从另一个事件的处理程序中分派一个新事件，以确保给定事件的处理程序不在相关的处理程序之后触发。

通配符事件处理器签名

通配符事件处理器现在接收事件名作为第一个参数以及事件数据作为第二个参数，`Event::firing` 方法被移除：

```
Event::listen('*', function ($eventName, array $data) {
    //
});
```

`kernel.handled` 事件

`kernel.handled` 事件现在是一个基于对象的事件，使用 `Illuminate\Foundation\Http\Events\RequestHandled` 类。

`locale.changed` 事件

`locale.changed` 事件现在是一个基于对象的事件，使用 `Illuminate\Foundation\Events\LocaleUpdated` 类。

`illuminate.log` 事件

`illuminate.log` 事件现在是一个基于对象的事件，使用 `Illuminate\Log\Events\MessageLogged` 类。

异常

`Illuminate\Http\Exception\HttpResponseException` 已被重命名为 `Illuminate\Http\Exceptions\HttpResponseException`。注意 `Exceptions` 现在是复数。类似的，`Illuminate\Http\Exception\PostTooLargeException` 已经被重命名为 `Illuminate\Http\Exceptions\PostTooLargeException`。

邮件

class@method 语法

不再支持使用 `Class@method` 语法发邮件。例如：

```
Mail::send('view.name', $data, 'Class@send');
```

如果你是以这种方式发送邮件，需要将这些调用做转化 [mailables](#)。

新的配置选项

为了支持 Laravel 5.4 版本的 Markdown 邮件组件，你需要添加如下区域配置到 `mail` 配置文件底部：

```
'markdown' => [
    'theme' => 'default',

    'paths' => [
        resource_path('views/vendor/mail'),
    ],
],
```

使用闭包将邮件推送到队列

为了将邮件推送到队列，必须使用 [mailable](#) 提供的方法。使用 `Mail::queue` 和 `Mail::later` 方法将邮件推送到队列，不再支持使用闭包来配置邮件消息。该功能需要使用指定的库来序列化闭包，因为 PHP 原生不支持这一功能特性。

Redis

改进的集群支持

Laravel 5.4 引入了改进的 Redis 集群支持。如果你正在使用 Redis 集群，则应该将集群连接置于 `config/database.php` 配置文件的 `Redis` 部分中的 `clusters` 配置选项内：

```
'redis' => [
    'client' => 'predis',
    'options' => [
        'cluster' => 'redis',
    ],
    'clusters' => [
        'default' => [
            [

```

```

    'host' => env('REDIS_HOST', '127.0.0.1'),
    'password' => env('REDIS_PASSWORD', null),
    'port' => env('REDIS_PORT', 6379),
    'database' => 0,
],
],
],
],
]
,
```

路由

Post 大小中间件

`Illuminate\Foundation\Http\Middleware\VerifyPostSize` 类被重命名为 `Illuminate\Foundation\Http\Middleware\ValidatePostSize`。

middleware 方法

`Illuminate\Routing\Router` 类的 `middleware` 方法已重命名为 `aliasMiddleware()`。似乎大多数应用都不会直接手动调用这个方法，因为这个方法只会被 HTTP kernel 调用，用于注册定义在 `$routeMiddleware` 数组中的路由级中间件。

Route 方法

`Illuminate\Routing\Route` 类的 `getUri` 方法已删除。你应该使用 `uri` 方法。

`Illuminate\Routing\Route` 类的 `getMethods` 方法已经删除。你应该使用 `methods` 方法。

`Illuminate\Routing\Route` 类的 `getParameter` 方法已经删除。你应该使用 `parameter` 方法。

`Illuminate\Routing\Route` 类的 `getPath` 方法已经删除。你应该使用 `uri` 方法。

会话

Symfony兼容性

Laravel 的会话处理程序不再实现 Symfony 的 `SessionInterface`。实现这个接口需要我们实现框架不需要的无关特性。取而代之，已经定义了新的 `Illuminate\Contracts\Session\Session` 接口，并且可以使用。还应该修改一下代码：

所有调用 `->set()` 方法应该更改为 `->put()`。通常，Laravel 应用从不调用 `set` 方法，因为它从未在 Laravel 文档中记录。不过，谨慎起见，这里我们依然罗列出来。

所有调用 `->getToken()` 方法的地方需要修改为 `->token()`。

所有调用 `$request->setSession()` 方法的地方需要求改为 `setLaravelSession()`。

测试

Laravel 5.4 的测试层已经被重成更加简单和更加轻量级。如果你想继续使用 Laravel 5.3 中的测试层，你可以安装 `laravel/browser-kit-testing` package 到你的应用程序。该包提供了与 Laravel 5.3 测试层的完全兼容。实际上，你可以同时运行 Laravel 5.3 和 Laravel 5.4 的测试层。

在一个应用里面运行 Laravel 5.3 & 5.4 的测试

在开始之前，你应该将 `Tests` 命名空间添加到 `composer.json` 文件中的 `autoload-dev` 块中。这将允许 Laravel 自动加载你使用 Laravel 5.4 测试生成器生成的任何新测试：

```
"psr-4": {
    "Tests\\": "tests/"
}
```

接下来，安装 `laravel/browser-kit-testing` 包：

```
composer require laravel/browser-kit-testing
```

安装软件包后，创建 `tests\TestCase.php` 文件的副本，并将其作为 `BrowserKitTestCase.php` 保存到你的 `tests` 文件夹中。然后，修改该文件以扩展 `Laravel\BrowserKitTesting\TestCase` 类。完成之后，你应该在你的 `tests` 目录中有两个基本测试类：`TestCase.php` 和 `BrowserKitTestCase.php`。为了正确加载你的 `BrowserKitTestCase` 类，你可能需要将它添加到你的 `composer.json` 文件：

```
"autoload-dev": {
    "classmap": [
        "tests/Testcase.php",
        "tests/BrowserKitTestCase.php"
    ]
},
```

在 Laravel 5.3 上编写的测试将扩展 `BrowserKitTestCase` 类，而使用 Laravel 5.4 测试层的任何新测试将扩展 `TestCase` 类。你的 `BrowserKitTestCase` 类应该如下所示：

```
<?php

use Illuminate\Contracts\Console\Kernel;
use Laravel\BrowserKitTesting\TestCase as BaseTestCase;

abstract class BrowserKitTestCase extends BaseTestCase
{
    /**
     * 应用的基础 URL 。
     *
     * @var string
    
```

```

    */
public $baseUrl = 'http://localhost';

/**
 * 创建应用。
 *
 * @return \Illuminate\Foundation\Application
 */
public function createApplication()
{
    $app = require __DIR__ . '/../bootstrap/app.php';

    $app->make(Kernel::class)->bootstrap();

    return $app;
}
}

```

创建好这些类之后，确保更新所有已编写的测试类继承自 `BrowserKitTestCase` 类，这将使得所有基于 Laravel 5.3 编写的测试类在 Laravel 5.4 中得以继续运行。如果你选择，你可以慢慢地开始将它们移植到新的 [Laravel 5.4 测试语法](#) 或者 [Laravel Dusk](#)。

{note} 如果你正在编写新的测试，并且希望它们使用 Laravel 5.4 的测试层，确保继承自 `TestCase` 类。

在已升级应用中安装 Duck

如果你想在已升级应用中安装 Laravel Duck，首先通过 Composer 安装：

```
composer require laravel/dusk
```

记下来，你需要在 `tests` 目录中创建一个 `CreatesApplication trait`，该 trait 用于为测试用例创建新的应用实例。该 trait 代码如下：

```

<?php

use Illuminate\Contracts\Console\Kernel;

trait CreatesApplication
{
    /**
     * 创建应用。
     *
     * @return \Illuminate\Foundation\Application
     */
    public function createApplication()
    {
        $app = require __DIR__ . '/../bootstrap/app.php';
    }
}

```

```

    $app->make(Kernel::class)->bootstrap();

    return $app;
}
}

```

{note} 如果你的测试类位于某个命名空间下，并使用 PSR-4 自动加载标准来加载 `tests` 目录，则应该将 `CreatesApplication` trait 放置在合适的命名空间下。

完成了这些准备工作，你就可以按照正常的 [Dusk 安装指南](#) 进行操作。

环境

Laravel 5.4 不需要在每个测试类中手动强制设置 `putenv('APP_ENV=testing')`，取而代之，框架使用从 `.env` 文件中载入 `APP_ENV` 变量。

时间伪造

`Event` 伪造的 `assertFired` 方法应该更新为 `assertDispatched`，并且 `assertNotFired` 方法应该更新为 `assertNotDispatched`。方法的签名没有变更。

邮件伪造

`Mail` 伪造在 Laravel 5.4 中极大的简化。现在应该使用 `assertSent` 方法以及 `hasTo`，`hasCc` 等辅助函数即可，不再需要调用 `assertSentTo` 方法：

```

Mail::assertSent(MailableName::class, function ($mailable) {
    return $mailable->hasTo('email@example.com');
});

```

翻译

{Inf} 占位符

如果你在使用 `{Inf}` 占位符处理字符串复数格式的翻译，你需要更新为使用 `*` 字符：

```
{0} First Message|{1,*} Second Message
```

URL 生成

forceSchema 方法

`Illuminate\Routing\UrlGenerator` 类中的 `forceSchema` 方法已经重命名为 `forceScheme`。

验证

日期格式验证

日期格式验证现在更加严格，并支持对 PHP [date function](#) 中列出的占位符进行校验。在之前版本中，时区占位符 `P` 会接受所有的时区格式；然而，在 Laravel 5.4 中，每种时区格式像在 PHP 文档一样，都有一个唯一的占位符。

方法名

`addError` 方法已经重命名为 `addFailure`。此外，`doReplacements` 方法重命名为 `makeReplacements`。通常，这些改变只有当你去扩展 `Validator` 类时才会有影响。

杂项

我们同样鼓励你去翻阅 `laravel/laravel` [GitHub 仓库](#) 中的修改记录。虽然很多修改不是必需的，你可能希望这些文件与你的应用程序同步。另外的一些变更将在本升级指南中介绍，但是其他更改不会。例如更改配置文件或者注释。你可以通过 [Github comparison tool](#) 很轻松的查看到这些变更，并选择对你重要的更新。

Laravel 源代码贡献指南

- [错误反馈](#)
- [核心开发讨论](#)
- [选择分支](#)
- [安全漏洞](#)
- [代码风格](#)
 - [PHPDoc](#)
 - [StyleCI](#)

错误反馈

为了提倡积极协作，Laravel 强烈地鼓励使用 Pull Request，而不仅仅只是反馈错误。「错误反馈」可以以一个包含失败测试的 Pull Request 的形式发送。

假如你提交了错误反馈，那么反馈中应该包含着标题和详尽的问题描述，并尽可能多的提供相关的信息和错误问题的代码示例。错误反馈的主要目的是让自己和其他人可以简单地重现并修复错误。

请记住，错误反馈的初衷是让其它有相同问题的人可以协作解决问题。不要期望反馈错误后会很快有人会马上修复它。创建错误反馈主要是为了能帮助你自己和其他人开始着手修复问题。

Laravel 源代码托管在 GitHub 上面，并且每个 Laravel 的项目都有自己的代码仓库：

- [Laravel Framework](#)
- [Laravel Application](#)
- [Laravel Documentation](#)
- [Laravel Cashier](#)
- [Laravel Cashier for Braintree](#)
- [Laravel Envoy](#)
- [Laravel Homestead](#)
- [Laravel Homestead Build Scripts](#)
- [Laravel Website](#)
- [Laravel Art](#)

核心开发讨论

如果你想提出功能建议，或者改进现有的 Laravel 的行为，请到 Laravel Internals 项目的 [反馈栏](#) 讨论。如果你想提出功能建议，我们希望你愿意为此功能贡献一些代码。

有关错误、新功能和现有功能的实现讨论会在 Slack 的 [LaraChat](#) 群组上的 `#internals` 频道中进行。Laravel 的维护者 Taylor Otwell 在工作日的 8am 到 5pm (UTC-06:00 或 America/Chicago) 通常都会出现在频道上，其它时间偶尔也会出现。

选择分支

所有的 错误修复都应该发送到最新的稳定分支上。除非它们修复的功能只存在下一版的发布中，不然错误修复永远不应该发送到 `master` 分支。

次要的且与现有的 Laravel 发布版本完全向下兼容的功能可以发送到最新的稳定分支。

主要的新功能应该都发送到 `master` 分支，它包含下一版的 Laravel 发布内容。

如果不确定你的功能是主要的还是次要的，请在 Slack 的 [LaraChat](#) 群组上的 `#internals` 频道询问 Taylor Otwell。

安全漏洞

如果你发现 Laravel 存在安全漏洞，请发送电子邮件到 taylor@laravel.com 给 Taylor Otwell。所有的安全漏洞将会及时予以处理。

编码风格

Laravel 遵守 [PSR-2](#) 编码规范和 [PSR-4](#) 自动加载规范。

译者注：扩展阅读 - [所有 PSR 的标准规范](#)。

注释区块

`@param` 标签应该分行显示，并且每一个参数中间需相隔两个空格，举个例子：

```
/**
 * 为服务容器注册一个绑定
 *
 * @param string|array $abstract
 * @param \Closure|string|null $concrete
 * @param bool $shared
 * @return void
 */
public function bind($abstract, $concrete = null, $shared = false)
{
    //
}
```

StyleCI

Laravel 使用 [StyleCI](#) 来做代码矫正，所有 PR 都会在合并后被矫正代码样式，这样做允许我们把精力放到贡献的内容上，而不是代码风格。

2 入门指南

Laravel 安装指南

- 安装
 - 服务器要求
 - 安装 Laravel
 - 配置
- Web 服务器配置
 - 优雅链接

安装

{video} 你是一个喜欢看视频的学习者么？Laracasts 为刚刚使用这个框架的新手们提供了一个 [免费、深入的 Laravel 视频](#)。这是一个开始你学习之途的好地方。

服务器要求

Laravel 框架会有一些系统上的要求。当然，这些要求在 [Laravel Homestead](#) 虚拟机上都已经完全配置好了。所以，非常推荐你使用 Homestead 作为你的本地 Laravel 开发环境。

然而，如果你没有使用 Homestead，你需要确保你的服务器上安装了下面的几个拓展：

- PHP >= 5.6.4 - OpenSSL PHP Extension - PDO PHP Extension - Mbstring PHP Extension - Tokenizer PHP Extension - XML PHP Extension

译者注：强烈推荐使用 Homestead 作为开发环境，尤其是新手，可以避免很多不必要的麻烦。
线上环境可以参考 [Homestead 的环境部署脚本](#) 进行部署。

安装 Laravel

Laravel 使用 [Composer](#) 来管理代码依赖。所以，在使用 Laravel 之前，请先确认你的电脑上安装了 Composer。

通过 Laravel 安装工具

首先，使用 Composer 下载 Laravel 安装包：

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

请确定你已将 `~/.composer/vendor/bin` 路径加到 PATH，只有这样系统才能找到 `laravel` 的执行文件。

一旦安装完成，就可以使用 `laravel new` 命令在指定目录创建一个新的 Laravel 项目，例如：`laravel new blog` 将会在当前目录下创建一个叫 `blog` 的目录，此目录里面存放着新安装的 Laravel 和代码依赖。这个方法的安装速度比通过 Composer 安装要快上许多：

```
laravel new blog
```

因为代码依赖是直接一起打包安装的。

通过 Composer Create-Project

除此之外，你也可以通过 Composer 在命令行运行 `create-project` 命令来安装 Laravel：

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

本地开发服务器

如果你在本地安装了 PHP，你可能希望像运行 PHP 内置的开发服务器一样来访问自己的应用程序，你可以使用 `serve` Artisan 命令来启动一个本地开发服务器，这样你就可以在 `http://localhost:8000` 来访问它。

```
php artisan serve
```

不过有更健壮的本地开发选项可用，比如 [Homestead](#) 和 [Valet](#)。

配置

入口目录

在安装 Laravel 之后，你需要配置你的 Web 服务器的根目录为 `public` 目录。这个目录的 `index.php` 文件作为所有 HTTP 请求进入应用的前端处理器。

配置文件

Laravel 框架所有的配置文件都存放在 `config` 目录下。每个选项都被加入文档，所以你可以自由的浏览文件，轻松的熟悉你的选项。

目录权限

安装 Laravel 之后，你需要配置一些权限。`storage` 和 `bootstrap/cache` 目录应该允许你的 Web 服务器写入，否则 Laravel 将无法写入。如果你使用 [Homestead](#) 虚拟机，这些权限应该已经被设置好了。

应用程序密钥

在你安装完 Laravel 后，首先需要做的事情是设置一个随机字符串的密钥。假设你是通过 Composer 或是 Laravel 安装工具安装的 Laravel，那么这个密钥已经通过 `key:generate` 命令帮你设置完成。

通常这个密钥会有 32 字符长。这个密钥可以被设置在 `.env` 环境文件中。如果你还没将 `.env.example` 文件重命名为 `.env`，那么你现在应该去设置下。如果你没有设置应用程序密钥，你的用户 Session 和其他加密数据将不安全！

额外配置

Laravel 几乎不需做任何其它设置就可以马上使用，但是建议你先浏览 `config/app.php` 文件和对应的文档，这里面包含着一些选项，如 时区 和 语言环境，你可以根据应用程序的情况来修改。

你也可以设置 Laravel 的几个附加组件，像是：

- 缓存
- 数据库
- Session

一旦 Laravel 安装完成，你应该立即 [设置本机环境](#)。

Web 服务器配置

优雅链接

Apache

Laravel 框架通过 `public/.htaccess` 文件来让 URL 不需要 `index.php` 即可访问。在 Apache 启用 Laravel 之前，请确认是否有开启 `mod_rewrite` 模块，以便 `.htaccess` 文件发挥作用。

如果 Laravel 附带的 `.htaccess` 文件在 Apache 中无法使用的话，请尝试下方的做法：

```
Options +FollowSymLinks
RewriteEngine On

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

Nginx

如果你使用 Nginx，在你的网站配置中加入下述代码将会转发所有的请求到 `index.php` 前端控制器。

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

当然如果你使用了 [Homestead](#) 或者 [Valet](#) 的话，它会自动的帮你设置好优雅链接。

Laravel 的配置信息

- [基础介绍](#)
- [环境配置](#)
 - [判定目前使用的环境](#)
- [获取设置值](#)
- [缓存配置信息](#)
- [维护模式](#)

基础介绍

所有 Laravel 框架的配置文件都放置在 `config` 目录下。每个选项都有说明，请仔细阅读这些说明，并熟悉这些选项配置。

环境配置

应用程序常常需要根据不同的运行环境设置不同的值。例如，你会希望在本机开发环境上有与正式环境不同的缓存驱动。类似这种环境变量，只需通过 `.env` 配置文件就可轻松完成。

Laravel 使用 Vance Lucas 的 [DotEnv](#) PHP 函数库来实现项目内环境变量的控制，在安装好的全新 Laravel 应用程序里，在根目录下会包含一个 `.env.example` 文件。如果你通过 Composer 安装 Laravel，这个文件将自动被更名为 `.env`，否则你只能手动更改文件名。

{tip} 你也可以新建一个 `.env.testing` 文件。当在运行 PHPUnit 测试或者带有 `--env=testing` 选项运行 Artisan 命令的时候，这个 `.env.testing` 文件会覆盖掉 `.env` 文件中对应的值。

获取环境变量

当你的应用程序收到请求时，这个文件所有的变量都会被加载到 PHP 超级全局变量 `$_ENV` 里。你可以使用辅助函数 `env` 来获取这些变量的值。事实上，如果你阅读过 Laravel 的相关配置文件，你会注意到里面有几个选项已经在使用着这个辅助函数！

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

`env` 函数的第二个参数是默认值，如果未找到对应的环境变量配置的话，此值就会被返回。

根据本机服务器或者正式环境的需求的不同，可自由修改环境变量。但是，`.env` 文件不应该被提交到版本控制系统，因为每个开发人员或服务器在使用应用程序时，可能需要不同的环境配置。

不妨将 `.env.example` 文件放进你的应用程序，通过样本配置文件里的预设值，团队中的其他开发人员就可以清楚地知道，在运行你的应用程序时有哪些环境变量是必须有的。

判定目前使用的环境

应用程序的当前环境是由 `.env` 文件中的 `APP_ENV` 变量所决定的。你可以通过 `App facade` 的 `environment` 方法来获取该值：

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

你也可以传递参数至 `environment` 方法来确认当前环境是否与参数相符合：

```
if (App::environment('local')) {
    // 当前正处于本地开发环境
}

if (App::environment('local', 'staging')) {
    // 当前环境处于 `local` 或者 `staging`
}
```

获取设置值

可以使用 `config` 辅助函数获取你的设置值，设置值可以通过「点」语法来获取，其中包含了文件与选项的名称。你也可以指定一个默认值，当该设置选项不存在时就会返回默认值：

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

若要在运行期间修改设置值，请传递一个数组至 `config` 辅助函数：

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

缓存配置信息

为了让应用程序的速度获得提升，可以使用 Artisan 命令 `config:cache` 将所有的配置文件缓存到单个文件。通过此命令将所有的设置选项合并成一个文件，让框架能够更快速的加载。

你应该将运行 `php artisan config:cache` 命令作为部署工作的一部分。此命令不应该在开发时运行，因为设置选项会在开发时经常变动。

{note} 如果你在部署过程中执行 `config:cache` 命令，你应该确保在你的配置文件中你只调用了 `env` 函数。

译者注：更多 Laravel 程序调优技巧请参阅：[Laravel 5 程序优化技巧](#)

维护模式

当你的应用程序处于维护模式时，所有传递至应用程序的请求都会显示出一个自定义视图。在你更新应用或进行性能维护时，这么做可以很轻松的「关闭」整个应用程序。维护模式会检查包含在应用程序的默认的中间件堆栈。如果应用程序处于维护模式，则 `MaintenanceModeException` 会抛出 503

的状态码。

启用维护模式，只需要运行 Artisan 命令 `down`：

```
php artisan down
```

你可以指定 `down` 命令的 `message` 和 `retry` 选项。`message` 自定义显示给用户的信息，`retry` 作为 `Retry-After` HTTP 标头返回：

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

关闭维护模式，请使用 Artisan 命令 `up`：

```
php artisan up
```

维护模式的响应模板

维护模式的默认模板放在 `resources/views/errors/503.blade.php`。你可以根据你的需求来修改这个模版

维护模式与队列

当应用程序处于维护模式中时，将不会处理任何 [队列工作](#)。所有的队列工作将会在应用程序离开维护模式后被继续运行。

维护模式的替代方案

维护模式有几秒钟的服务器不可用时间，如果你想做到平滑迁移的话，推荐使用 [Envoyer](#) 服务。

Laravel 的文件夹结构

- 简介
- 根目录
 - app 目录
 - bootstrap 目录
 - config 目录
 - database 目录
 - public 目录
 - resources 目录
 - routes 目录
 - storage 目录
 - tests 目录
 - vendor 目录
- App 目录
 - Console 目录
 - Events 目录
 - Exceptions 目录
 - Http 目录
 - Jobs 目录
 - Listeners 目录
 - Mail 目录
 - Notifications 目录
 - Policies 目录
 - Providers 目录

简介

Laravel 默认的目录结构意在为构建不同大小的应用提供一个好的起点，当然，你可以自己按照喜好组织应用目录结构， 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。

为什么没有 Models 目录？

许多初学者都会困惑 Laravel 为什么没有 models 目录，当然，这是 laravel 故意为之，因为 models 这个词对不同开发者而言有不同的含义，容易造成歧义，有些开发者认为应用的模型指的是业务逻辑，还有些开发者则认为模型指的是与关联数据库的交互。

正是因为如此，我们默认将 Eloquent 的模型放置到 app 目录下，从而允许开发者自行选择放置的位置。

根目录

app 目录

`app` 目录，如你所料，这里面包含应用程序的核心代码。另外，你为应用编写的代码绝大多数也会放到这里，我们之后将很快对这个目录的细节进行深入探讨。

bootstrap 目录

`bootstrap` 目录包含了几个框架启动和自动加载设置的文件。`cache` 文件夹用于包含框架为提升性能所生成的文件，如路由和服务缓存文件。

config 目录

`config` 目录，顾名思义，包含所有应用程序的配置文件。通读这些配置文件可以应对自己对配置修改的需求。

database 目录

`database` 目录包含了数据迁移及填充文件，你还可以将其作为 SQLite 数据库的存放目录。

public 目录

`public` 目录包含了 Laravel 的 HTTP 入口文件 `index.php` 和前端资源文件（图片、JavaScript、CSS 等）。

resources 目录

`resources` 目录包含了视图、原始的资源文件 (LESS、SASS、CoffeeScript)，以及语言包。

routes 目录

`routes` 目录包含了应用的所有路由定义。Laravel 默认提供了三个路由文件：`web.php`，`api.php`，和 `console.php`。

`web.php` 文件里定义的路由都会在 `RouteServiceProvider` 中被指定应用到 `web` 中间件组，具备 Session、CSRF 防护以及 Cookie 加密功能，如果应用无需提供无状态的、RESTful 风格的 API，所有路由都会定义在 `web.php` 文件。

`api.php` 文件里定义的路由都会在 `RouteServiceProvider` 中被指定应用到 `api` 中间件组，具备频率限制功能，这些路由是无状态的，所以请求通过这些路由进入应用需要通过 API 令牌进行认证并且不能访问 Session 状态。

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

storage 目录

`storage` 目录包含编译后的 Blade 模板、基于文件的 session、文件缓存和其它框架生成的文件。此文件夹分格成 `app` 、 `framework` ，及 `logs` 目录。`app` 目录可用于存储应用程序使用的任何文件。`framework` 目录被用于保存框架生成的文件及缓存。最后，`logs` 目录包含了应用程序的日志文件。

`storage/app/public` 可以用来存储用户生成的文件，例如头像文件，这是一个公开的目录。你还需要在 `public/storage` 目录下生成一个软连接指向这个目录，你可以使用 `php artisan storage:link` 来创建软链接。

tests 目录

`tests` 目录包含自动化测试。Laravel 推荐了一个 [PHPUnit](#) 例子。每一个测试类都需要添加 `Test` 前缀，你可以使用 `phpunit` 或者 `php vendor/bin/phpunit` 命令来运行测试。

vendor 目录

`vendor` 目录包含所有 [Composer](#) 依赖。

app 目录

应用的核心代码位于 `app` 目录下，默认情况下，该目录位于命名空间 `App` 下，并且被 Composer 通过 [PSR-4](#) 自动载入标准自动加载。

`app` 目录下包含多个子目录，如 `Console` 、 `Http` 、 `Providers` 等。其中 `Console` 和 `Http` 目录为进入应用程序核心提供了一个 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，它们是两种简单地发布命令给应用程序的方法。`Console` 目录包含你全部的 Artisan 命令，而 `Http` 目录包含你的控制器、中间件和请求。

其他目录将会在你通过 Artisan 命令 `make` 生成相应类的时候生成到 `app` 目录下。例如，`app/Jobs` 目录在你执行 `make:job` 命令生成任务类时，才会出现在 `app` 目录下。

{tip} `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。

Console 目录

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

Events 目录

`Events` 目录默认不存在，它会在你使用 `event:generate` 或者 `event:make` 命令以后才会生成。如你所料，此目录是用来放置 [事件类](#) 的。事件类用于当指定事件发生时，通知应用程序的其它部分，并提供了很棒的灵活性及解耦。

Exceptions 目录

`Exceptions` 目录包含应用的异常处理，同时还是处理应用抛出的任何异常的好位置。如果你想自定义异常的记录和渲染，你应该修改此目录下的 Handler 类。

Http 目录

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

Jobs 目录

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

Listeners 目录

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

Mail 目录

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

Notifications 目录

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

Policies 目录

`Policies` 你可以通过执行 `make:policy` 命令来创建，`Policies` 目录包含了所有的授权策略类，策略用于判断某个用户是否有权限去访问指定资源。更多详情，请查看 [授权文档](#)。

Providers 目录

`Providers` 目录包含应用的 [服务提供者](#)。服务提供者在启动应用过程中绑定服务到容器、注册事件，以及执行其他任务，为即将到来的请求处理做准备。

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

Laravel 的请求生命周期

- [介绍](#)
- [请求生命周期概述](#)
- [聚焦服务提供者](#)

介绍

使用「现实世界」中的任何工具时，如果理解这个工具的运作原理，那么你会更加得心应手的使用这个工具。应用开发也是这样。当你明白你的开发工具如何运行的，你就会对它们的使用游刃有余。

这篇文档的目的是让你更好的理解 Laravel 框架如何进行工作及它的工作原理。通过对框架进行全面的了解，一切都不会那么「神奇」，也将会让你更有自信的构建你的应用。如果你不能理解所有的这些术语，也不要丧失信心！只要对现在提到的东西有个基本概念，随着对本文档和其他章节的不断探索，你对它们的理解会不断提升。

生命周期概述

第一件事

一个 Laravel 应用的所有请求的入口都是 `public/index.php` 文件。通过网页服务器 (Apache / Nginx) 所有请求都会导向这个文件。`index.php` 文件没有太多的代码，只是加载框架其他部分的一个入口。

`index.php` 文件载入 Composer 生成的自动加载器定义，并从 `bootstrap/app.php` 文件获取到 Laravel 应用实例。Laravel 的第一个动作就是创建一个自身应用实例 / [服务容器](#)。

HTTP / Console 内核

接下来，传入的请求会被发送给 HTTP 内核或者 console 内核，这根据进入应用的请求的类型而定。这两个内核服务是所有请求都经过的中枢。让我们现在只关注位于 `app/Http/Kernel.php` 的 HTTP 内核。

HTTP 内核继承自 `Illuminate\Foundation\Http\Kernel` 类，它定义了一个 `bootstrappers` 数组，数组中的类在请求真正执行前进行前置执行。这些引导程序配置了错误处理，日志记录，[检测应用程序环境](#)，以及其他在请求被处理前需要完成的工作。

HTTP 内核同时定义了一个 HTTP [中间件](#) 列表，所有的请求必须在处理前通过这些中间件，这些中间件处理 [HTTP session](#) 的读写，判断应用是否在维护模式，[验证 CSRF token](#) 等等。

HTTP 内核的标志性 `handle` 方法是相当简单的：接收一个 `Request` 并返回一个 `Response`。你可以把内核想成一个代表你应用的大黑盒子。给它喂 HTTP 请求然后它就会吐给你 HTTP 响应。

服务提供者

在内核引导启动的过程中最重要的动作之一就是载入 [服务提供者](#) 到你的应用。所有的服务提供者都配置在 `config/app.php` 文件中的 `providers` 数组中。首先，所有提供者的 `register` 方法会被调用，接下来，一旦所有提供者注册完成，`boot` 方法将会被调用。

服务提供者负责引导启动框架的全部各种组件，例如数据库、队列、验证器以及路由组件。因为这些组件引导和配置了框架的各种功能，所以服务提供者是整个 Laravel 启动过程中最为重要的部分。

分发请求

一旦应用完成引导和所有服务提供者都注册完成，`Request` 将会移交给路由进行分发。路由将分发请求给一个路由或控制器，同时运行路由指定的中间件。

聚焦服务提供者

服务提供者是 Laravel 应用的真正关键部分，应用实例被创建后，服务提供者就会被注册完成，并将请求传递给应用进行处理，真的就是这么简单！

了解 Laravel 是怎样通过服务提供者构建和引导一个稳定的应用是非常有价值的，当然，应用的默认服务提供者都存放在 `app/Providers` 目录中。

在新创建的应用中，`AppServiceProvider` 文件中方法实现都是空的。这个提供者是你添加应用专属的引导和服务的最佳位置，当然的，对于大型应用你可能希望创建几个服务提供者，每个都具有粒度更精细的引导。

3 开发环境部署

Laravel 虚拟开发环境 Homestead

- [简介](#)
- [安装与设置](#)
 - [第一步](#)
 - [配置 Homestead](#)
 - [启动 Vagrant Box](#)
 - [根据项目分开安装](#)
 - [安装 MariaDB](#)
- [常见用法](#)
 - [全局可用的 Homestead](#)
 - [通过 SSH 连接](#)
 - [连接数据库](#)
 - [增加更多网站](#)
 - [配置 Cron 调度器](#)
 - [连接端口](#)
 - [共享你的环境](#)
- [网络接口](#)
- [更新 Homestead](#)
- [历史版本](#)
- [Provider 的特殊设置](#)
 - [VirtualBox](#)

简介

Laravel 致力于让 PHP 的开发过程更加轻松愉快，这其中也包括你的本地开发环境。[Vagrant](#) 提供了一种简单，优雅的方式来管理和调配虚拟机。

Laravel Homestead 是一个官方预封装的 Vagrant box，它为你提供了一个完美的开发环境，你无需在本地安装 PHP，web 服务器，或其他服务软件。并且不用担心系统被搞乱！Vagrant box 是完全一次性的。如果有什么地方出错了，你也可以在几分钟内销毁并重建 box！

Homestead 可以运行在 Windows、Mac 或 Linux 系统上，并且里面包含了 Nginx Web 服务器、PHP 7.1、MySQL、Postgres、Redis、Memcached、Node、以及所有利于你开发 laravel 应用的其他程序。

{note} 如果你是 Windows 用户，你可能需要启用硬件虚拟化（VT-x）。这通常需要通过 BIOS 来启用它。如果你在一个 UEFI 系统上使用的是 Hyper-V，你需要关闭 Hyper-V 才能启用 VT-x。

内置软件

- Ubuntu 16.04
- Git

- PHP 7.1
- Nginx
- MySQL
- MariaDB
- Sqlite3
- Postgres
- Composer
- Node (With Yarn, Bower, Grunt, and Gulp)
- Redis
- Memcached
- Beanstalkd
- Mailhog
- ngrok

安装与设置

第一步

在你启动你的 Homestead 环境之前，你必须安装 [VirtualBox 5.1](#)、[VMWare](#) 或 [Parallels](#) 的其中之一，以及 [Vagrant](#)。这些软件在各个常用的平台都有提供简单易用的界面安装包。

若要使用 VMware provider，你需要同时购买 VMware Fusion / Workstation 以及 [VMware Vagrant plug-in](#) 的软件授权。使用 VMware 可以在共享文件夹上获得较快的性能。

若要使用 Parallels provider，你需要安装 [Parallels Vagrant plug-in](#)。这是免费使用的。

安装 Homestead Vagrant Box

当 VirtualBox / VMware 以及 Vagrant 安装完成后，你使用以下命令将 laravel/homestead 这个 box 安装进你的 Vagrant 程序中。box 的下载会花费你一点时间，具体的下载时长由网络速度决定：

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

如果上面的命令运行失败，代表你使用的可能是旧版的 Vagrant，请升级你的 Vagrant。

安装 Homestead

你可以通过手动克隆代码仓库的方式来安装 Homestead。建议将代码仓库克隆至「home」目录中的 Homestead 文件夹，如此一来 Homestead box 就能将主机服务提供给你所有的 Laravel 项目：

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

由于 Homestead 的 `master` 分支并不是稳定分支，你应该检出已经标签过的稳定版本。你可以在 [Github Release Page](#) 找到最新的稳定版本。

```
cd Homestead

// 检出所需要的版本...
git checkout v4.0.5
```

一旦你克隆完 Homestead 的代码仓库，即可在 Homestead 目录中运行 `bash init.sh` 命令来创建 `Homestead.yaml` 配置文件。`Homestead.yaml` 文件会被放置在你的 Homestead 目录中：

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

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

配置 Homestead

配置你的提供者

`Homestead.yaml` 中的 `provider` 参数设置取决于你用的是哪一个 Vagrant 提供者 `virtualbox`、`vmware_fusion`、`vmware_workstation`，或者 `parallels`。你可以根据自己的喜好来设置提供者：

```
provider: virtualbox
```

配置共享文件夹

你可以在 `Homestead.yaml` 文件的 `folders` 属性里列出所有想与 Homestead 环境共享的文件夹。这些文件夹中的文件若有变更，它们将会在你的本机电脑与 Homestead 环境自动更新同步。你可以在这里设置多个共享文件夹：

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

若要启动 [NFS](#)，只需要在共享文件夹的设置值中加入一个简单的参数：

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

你也可以在配置中传递任何 Vagrant 中的 [共享文件夹](#) 支持的参数，在 `options` 参数下列出它们：

```

folders:
  - map: ~/Code
    to: /home/vagrant/Code
    type: "rsync"
    options:
      rsync__args: ["--verbose", "--archive", "--delete", "-zz"]
      rsync__exclude: ["node_modules"]

```

配置 Nginx 站点

对 Nginx 不熟悉吗？没关系。`sites` 属性可以帮助你可以轻易指定一个 `域名` 来对应到 homestead 环境中的一个目录上。在 `Homestead.yaml` 文件中已包含了一个网站设置范本。同样的，你也可以增加多个网站到你的 Homestead 环境中。Homestead 可以同时为多个 Laravel 应用提供虚拟化环境：

```

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

```

如果你在 Homestead box 配置之后更改了 `sites` 属性，那么应该重新运行 `vagrant reload --provision` 来更新 Nginx 配置到虚拟机上。

关于 Hosts 文件

你必须将在 Nginx sites 中所添加的「域名」也添加到你本机电脑的 `hosts` 上。`hosts` 文件会将请求重定向至 Homestead 环境中设置的本地域名。在 Mac 或 Linux 上，该文件通常会存放在 `/etc/hosts`。在 Windows 上，则存放于 `C:\Windows\System32\drivers\etc\hosts`。设置内容如下所示：

```
192.168.10.10 homestead.app
```

务必确认 IP 地址与 `Homestead.yaml` 文件中设置的相同。将域名设置在 `hosts` 文件之后，你就可以通过网页浏览器访问你的网站。

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

启动 Vagrant Box

编辑完 `Homestead.yaml` 后，进入你的 Homestead 目录并运行 `vagrant up` 命令。Vagrant 就会根据 `Homestead.yaml` 里的配置信息，为虚拟机设置共享文件夹和 Nginx 网站。

如果要移除虚拟机，你可以使用 `vagrant destroy --force` 命令

为每个项目分开安装

除了全局使用同一个 Homestead 环境， Homestead 还允许你为项目独立配置一个独占的 Homestead。

通过传递 `vagrantfile`，可以实现为每个项目分别安装上 Homestead，其他项目成员只需要通过简单的 `vagrant up` 即能跟你拥有一样的 Homestead 环境。

使用 Composer 将 Homestead 直接安装至项目中：

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

一旦 Homestead 安装完毕，可以使用 `make` 命令生成 `Vagrantfile` 与 `Homestead.yaml` 文件，并存放于项目的根目录。

`make` 命令将会自动在 `Homestead.yaml` 文件中配置 `sites` 及 `folders`：

Mac / Linux:

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

Windows:

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

接下来，在命令行中运行 `vagrant up` 并通过网页浏览器访问 `http://homestead.app`。再次提醒，你仍然需要在 `/etc/hosts` 里配置 `homestead.app` 或其它想要使用的域名。

安装 MariaDB

如果你希望使用 MariaDB 来替换 MySQL，你可以在 `Homestead.yaml` 文件中增加一个 `mariadb` 的选项，这个选项会移除 MySQL 并安装 MariaDB。因为 MariaDB 可用作 MySQL 的替代品，所以在你的数据库配置信息里，还是选用 `mysql` 配置项。

```
box: laravel/homestead
ip: "192.168.20.20"
memory: 2048
cpus: 4
provider: virtualbox
mariadb: true
```

常见用法

全局使用

如果你希望在文件系统的任何地方都可以 `vagrant up` 开启 Homestead 虚拟机，你可以把以下代码放到你的 Bash profile 里面，这个函数允许你在文件系统的任何位置都可以对 Homestead 运行 Vagrant 命令：

Mac / Linux

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

请将 `~/Homestead` 这个路径修改为你的实际 Homestead 的安装路径，一旦这个函数安装成功，就可以在系统的任意位置运行 `homestead up` 或 `homestead ssh` 命令

Windows

在系统的任意位置创建一个批处理文件 `homestead.bat`，并添加如下内容：

```
@echo off

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

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

set cwd=
set homesteadVagrant=
```

请将 `C:\Homestead` 这个路径修改为你的实际 Homestead 的安装路径，创建完这个文件后，将这个文件路径添加到 `PATH` 环境变量中，就可以在系统的任意位置运行 `homestead up` 或 `homestead ssh` 命令

通过SSH连接

在 Homestead 目录运行 `vagrant ssh` 命令来连接虚拟主机。

你可能会经常需要使用 SSH 来连接 Homestead 主机，你可以考虑将上述「function」添加到你的主机，以便快速的通过 SSH 进入你的 Homestead box

连接数据库

在 box 中已经为 MySQL 和 Postgres 配置好了一个开箱即用的数据库 `homestead`，为了更方便的使用它，Laravel 中的 `.env` 文件将这个数据库设置成了框架默认使用的数据库。

如果想要从你主机上的数据库客户端连接 MySQL 或 Postgres，可以通过 `127.0.0.1` 来使用端口 `33060` (MySQL) 或 `54320` (Postgres) 连接。账号密码分别是 `homestead / secret`

{note} 因为虚拟机做了端口转发，所以本机电脑上你应当只使用这些非标准的连接端口，虚拟机里依然使用默认的 3306 及 5432 连接端口。

增加更多网站

一旦 Homestead 环境配置完毕且成功运行后，你可能会想要为 Laravel 应用程序增加更多的 Nginx 网站。你可以在单个 Homestead 环境中运行多个 Laravel 程序。要添加额外的网站，只需将网站添加到您的 `Homestead.yaml` 文件中：

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

如果 Vagrant 没有自动管理你的「hosts」文件，你可能需要手动把新增的站点加入到「hosts」文件中：

```
192.168.10.10 homestead.app
192.168.10.10 another.app
```

当你的网站添加完成，从你的 Homestead 目录运行 `vagrant reload --provision` 命令就可以应用新的更改。

Site Types

Homestead supports several types of sites which allow you to easily run projects that are not based on Laravel. For example, we may easily add a Symfony application to Homestead using the `symfony2` site type:

```
sites:
  - map: symfony2.app
    to: /home/vagrant/Code/Symfony/public
    type: symfony2
```

The available site types are: `apache` , `laravel` (the default), `proxy` , `silverstripe` , `statamic` , and `symfony2` .

配置 Cron 调度器

Laravel 提供了便利的方式来 [调度 Cron 任务](#)，通过 `schedule:run` Artisan 命令，调度便会在每分钟被运行。`schedule:run` 命令会检查定义在你 `App\Console\Kernel` 类中调度的任务，判断哪个任务该被运行。

如果你想为 Homestead 网站使用 `schedule:run` 命令，你可以在定义网站时将 `schedule` 选项设置为 `true`

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

该网站的 Cron 任务会被定义在虚拟机的 `/etc/cron.d` 文件夹中。

连接端口

以下本地电脑连接端口将会被转发至 Homestead 环境：

- 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
- Mailhog: 8025 → Forwards To 8025

转发更多的端口

如果你需要的话，也可以借助指定连接端口的通信协议来转发更多额外的连接端口给 Vagrant box：

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

共享你的环境

有时候你想跟你的同事或者是客户共享你目前的工作进度。Vagrant 为此提供了一个内置方法 `vagrant share`；不过，如果你在你的 `Homestead.yaml` 文件中配置了多个站点则这条命令将会变得没多大用处。

为了解决这个问题，Homestead 提供了自己的 `share` 命令。开始之前，通过 `vagrant ssh` 命令 SSH 进你的 Homestead 机器中，然后运行 `share homestead.app`。这会从你的 `Homestead.yaml` 配置文件中共享 `homestead.app` 站点。当然，你也可以用其他已经配置的站点来代替 `homestead.app`。

运行完命令之后，你可以看到一个包含活动日志和共享站点外网访问路径的 Ngrok 界面。

{note} 谨记，Vagrant 本质上是不安全的，当你运行 `share` 命令的时候，你会把你的虚拟机暴露在互联网中。

网络接口

Network Interfaces

`Homestead.yaml` 文件里的 `nets` 配置项允许你为 Homestead 环境配置网络接口。你可以任意配置多个网络接口：

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

想要配置一个 [桥接](#) 接口的话，增加 `bridge` 配置项，然后 `type` 填写为 `public_network`：

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

想要配置一个 [DHCP](#) 接口的话，请从配置中移除 `ip` 选项：

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

更新 Homestead

你可以简单的用两个步骤来更新 Homestead，第一步，使用 `vagrant box update` 命令更新 Vagrant box：

```
vagrant box update
```

接下来。你需要更新 Homestead 的源代码，如果你是通过克隆仓库的方式安装的 Homestead，你可以简单的运行 `git pull origin master` 命令在你最初克隆仓库的位置。

如果你已经通过你的项目中的 `composer.json` 文件安装了 Homestead，你应该确认你的 `composer.json` 文件中是否包含 `"laravel/homestead: " ^4"` 并且更新你的依赖：

```
composer update
```

历史版本

你可以通过添加以下配置到你的 `Homestead.yaml` 文件来方便的覆盖 Homestead 使用的 box 版本

```
version: 0.6.0
```

例如：

```
box: laravel/homestead
version: 0.6.0
ip: "192.168.20.20"
memory: 2048
cpus: 4
provider: virtualbox
```

当你使用旧版本的 box 时，你需要确保 Homestead 源代码的版本与之对应，下面的图表展示了支持的 box 版本，以及与之对应的 Homestead 的源代码版本和 box 所提供的 PHP 版本：

	Homestead Version	Box Version
PHP 7.0	3.1.0	0.6.0
PHP 7.1	4.0.0	1.0.0

Provider 的特殊设置

VirtualBox

Homestead 默认将 `natdnshostresolver` 设置为 `on`。这允许 Homestead 使用你的主机系统中的 DNS 设置。如果你想重写这行为，你可以在你的 `Homestead.yaml` 文件中添加下面这几行：

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

Laravel 的开发环境 Valet

- 简介
 - 选择 Valet 还是 Homestead
- 安装
 - 升级
- 服务站点
 - 「Park」命令
 - 「Link」命令
 - 配置 TLS 让站点更安全
- 分享站点
- 自定义 Valet 驱动
 - 本地驱动
- 其他 Valet 命令

简介

Valet 是为 Mac 提供的极简主义开发环境，没有 Vagrant，也无需 `/etc/hosts` 文件，甚至可以使用本地隧道公开共享你的站点。Yeah, we like it too.

Laravel Valet 会在你的 Mac 上将 Nginx 设置为随系统启动后台运行，然后使用 DnsMasq，Valet 将所有的请求代理到 `*.dev` 域名并指向本地安装的站点目录。

换句话说，一个速度极快的 Laravel 开发环境仅仅需要占用 7MB 内存。Valet 并不是想要替代 Vagrant 或者 Homestead，只是提供另外一种选择，更加灵活、方便、以及占用更小的内存。

开箱即用，Valet 为我们提供以下软件和工具支持，然而不仅限于此：

- [Laravel](https://laravel.com) - [Lumen](https://lumen.laravel.com) - [Bedrock]
 (https://roots.io/bedrock/) - [CakePHP 3](https://cakephp.org) - [Concrete5]
 (http://www.concrete5.org/) - [Contao](https://contao.org/en/) - [Craft](https://craftcms.com) - [Drupal]
 (https://www.drupal.org/) - [Jigsaw](http://jigsaw.tighten.co) - [Joomla](https://www.joomla.org/) -
 [Katana](https://github.com/themsaid/katana) - [Kirby](https://getkirby.com/) - [Magento]
 (https://magento.com/) - [OctoberCMS](https://octobercms.com/) - [Sculpin](https://sculpin.io/) - [Slim]
 (https://www.slimframework.com) - [Statamic](https://statamic.com) - Static HTML - [Symfony]
 (https://symfony.com) - [WordPress](https://wordpress.org) - [Zend](https://framework.zend.com)

当然，你还可以通过 [自定义驱动](#) 来扩展 Valet.

选择 Valet 还是 Homestead

正如你所知道的，Laravel 提供另外一个开发环境 Homestead，Homestead 和 Valet 不同之处在于两者的目标受众和本地开发方式。Homestead 提供一个完整的包含自动化配置 Nginx 的 Ubuntu 虚拟机。如果你需要一个完整的虚拟化 Linux 开发环境或者是使用 Windows / Linux 操作系统，那么 Homestead 无疑是最佳选择。

Valet 只支持 Mac，并且要求本地安装 PHP 和数据库服务器，这可以通过使用 [Homebrew](#) 命令 `brew install php71` 和 `brew install mysql` 轻松实现。Valet 通过最小的资源消耗提供一个本地极速开发环境，如果你只需要 PHP / MySQL 而不是完整的虚拟化开发环境，那么 Valet 将是最好的选择。

Valet 和 Homestead 都是配置你本地 Laravel 开发环境的好帮手。选择使用哪一个取决于你的个人喜好和团队需求。

安装

Valet 要求 macOS 和 [Homebrew](#) 安装之前，你需要确保没有其他程序如 Apache 或者 Nginx 占用你本地机器的 80 端口。安装步骤如下：

- 安装或更新 [Homebrew](http://brew.sh/) 到最新版本，使用命令 `brew update` - 使用命令 `brew install homebrew/php/php71` 安装 PHP 7.1。 - 通过 `composer global require laravel/valet` 命令安装 Valet。请确定 `~/.composer/vendor/bin` 存在于你的系统环境变量 「PATH」 中。 - 运行 `valet install` 命令。它将会配置并安装 Valet 和 DnsMasq，并且将 Valet 的进程注册为随系统启动

一旦你完成 Valet 安装，尝试使用像 `ping foobar.dev` 这样的命令在终端 ping 任意的 `*.dev` 域名。如果 Valet 正常安装你会看到来自 `127.0.0.1` 的响应

Valet 将会在每次系统启动时自动启动，而不需要你每次运行 `valet start` 或 `valet install`。

使用其他的顶级域名

默认情况下，Valet 使用 `.dev` 顶级域名。如果你喜欢其他域名，可以使用 `valet domain tld-name` 命令。

例如，如果你想要使用 `.app` 来代替 `.dev`，运行 `valet domain app` 然后 Valet 会自动使用 `*.app` 来为你的项目命名。

数据库

如果你需要一个数据库，可以使用 `brew install mysql` 命令试一试 MySQL。如果你的 Mac 是第一次安装 MySQL 数据库，你可能需要执行 `brew services start mysql` 命令来启动它。之后，你可以使用 host 为 `127.0.0.1`，用户名 `root`，密码为空进行数据库连接。

升级

你可以使用 `composer global update` 命令升级你的 Valet 程序，升级之后，最好使用 `valet install` 命令更新 Valet 的配置文件。

升级到 Valet 2.0

Valet 2.0 将 Valet 的底层 Web 服务从 Caddy 切换到了 Nginx。在你升级到这个版本之前你应该运行下面的命令来停止并卸载已经启动的 Caddy 进程：

```
valet stop
```

`valet uninstall`

接下来，你应该升级到 Valet 的最新版本。取决于你安装 Valet 的方式，这通常通过 Git 或 Composer 来实现。如果你是通过 Composer 安装的 Valet 你应该使用下面的命令来更新到最新的主版本：

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

当新的 Valet 源代码下载好了之后，你应该运行 `install` 命令：

```
valet install
```

`valet restart`

在升级之后，它需要 re-park 或 re-link 你的站点。

服务站点

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

`park` 命令

- 在你的 Mac 中创建一个新的目录，例如 `mkdir ~/Sites`。然后 `cd ~/Sites` 并且运行 `valet park`。这个命令将在当前所在目录作为 Web 根目录，Valet 将会在这个目录中搜索站点。- 接下来，在这个目录中创建一个新的 Laravel 站点： `laravel new blog`。- 在浏览器中访问 `http://blog.dev`。这就是我们所要做的全部工作。现在，你所「parked」目录中的所有 Laravel 项目都可以通过 `http://folder-name.dev` 这种方式访问，是不是很方便。

`link` 命令

`link` 命令也被用来服务你的 Laravel 站点。这个命令在你想要在目录中提供单个站点是很有用。

- 要使用这个命令，在你的终端中切换到你的某个项目并运行 `valet link app-name`。Valet 将会在 `~/valet/Sites` 中创建一个符号链接并指向当前工作目录。- 运行完 `link` 命令，你可以在浏览器中通过 `http://app-name.dev` 来访问站点。

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

{tip} 你可以通过使用 `valet link` 将多个（子）域名指向同一个应用，要添加子域名或其它域名到应用，可以在应用目录下运行 `valet link subdomain.app-name`。

配置 TLS 让站点更安全

默认的情况下，Valet 通过纯 HTTP 协议服务网站。然而，如果你想利用 HTTP/2 提供加密的 TLS，你可以使用 `secure` 命令。例如，你有一个站点 `laravel.dev`，可以使用以下命令让其更安全：

```
valet secure laravel
```

想恢复一个站点到普通的 HTTP 使用 `unsecure` 命令，这个命令可以去除 `secure` 增加的安全加密：

```
valet unsecure laravel
```

分享站点

Valet 还提供一个命令将本地站点分享给其他人，这不需要任何额外安装软件即可实现。

要分享站点，在你的终端中切换到站点目录使用 `valet share` 命令。这会生成一个可以公开访问的 URL 并插入你的剪切板，以便你直接粘贴到浏览器，就是这么简单。

要停止分享站点，使用 `Control + C` 快捷组合键即可。

{note} `valet share` 目前尚不支持分享使用 `valet secure` 命令进行安全处理的站点。

自定义 Valet 驱动

你可以编写自定义的 Valet 「驱动」运行非原生支持的其他 PHP 框架或 CMS。安装完 Valet 时会创建一个 `~/.valet/Drivers` 目录，该目录中有一个 `SampleValetDriver.php` 文件。这个文件中简单演示如何编写自定义驱动。编写驱动只需要实现三个方法：`serves`，`isStaticFile` 和 `frontControllerPath`。

这三个方法都接收 `$sitePath`，`$siteName` 和 `$uri` 作为参数。`$sitePath` 表示站点的绝对路径，例如 `/Users/Lisa/Sites/my-project`。`$siteName` 表示站点的「host」/「站点名称」部分，如 (`my-project`)。`$uri` 则是输入的请求 URI，如 (`/foo/bar`)。

编写好你的自定义 Valet 驱动，将其放到 `~/.valet/Drivers` 目录并遵循 `FrameworkValetDriver.php` 这种命名规范。例如，如果编写一个自定义的 WordPress 驱动，对应的文件名称应是 `WordPressValetDriver.php`。

下面我们来具体讨论并演示自定义 Valet 驱动需要实现的三个方法。

`serves` 方法

如果自定义驱动要继续处理输入请求，`serves` 方法应该返回 `true`，否则该方法返回 `false`。因此，这个方法应该判断给定的 `$sitePath` 是否是包含你服务项目的类型。

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

```
/**
 * 判断驱动服务请求。
 *
 * @param string $sitePath
 * @param string $siteName
 * @param string $uri
 * @return bool
 */
```

```
public function serves($sitePath, $siteName, $uri)
{
    return is_dir($sitePath.'/wp-admin');
}
```

isStaticFile 方法

`isStaticFile` 应该判断进入的请求是否是静态文件，例如图片或者样式文件，如果文件是静态的，该方法会返回磁盘上的绝对路径，否则返回 `false`：

```
/**
 * 判断请求内容是否是静态文件。
 *
 * @param string $sitePath
 * @param string $siteName
 * @param string $uri
 * @return string|false
 */
public function isStaticFile($sitePath, $siteName, $uri)
{
    if (file_exists($staticFilePath = $sitePath.'/public/'.$uri)) {
        return $staticFilePath;
    }

    return false;
}
```

{note} `isStaticFile` 方法只有在 `serves` 方法返回 `true` 并且请求 URI 不是 `/` 才会被调用。

frontControllerPath 方法

`frontControllerPath` 方法应该返回「前端控制器」的绝对路径，通常是你「index.php」文件或其他同等文件：

```
/**
 * 获取应用前端控制器绝对路径。
 *
 * @param string $sitePath
 * @param string $siteName
 * @param string $uri
 * @return string
 */
public function frontControllerPath($sitePath, $siteName, $uri)
{
    return $sitePath.'/public/index.php';
}
```

本地驱动

如果你希望为一些单独的应用编写各自的 Valet 驱动，首先你需要在这些应用的根目录创建一个 `LocalValetDriver.php` 文件。在文件中你需要通过扩展基础类 `ValetDriver` 或使用这些应用特定的方法来编写 Valet 驱动，例如下面的 `LaravelValetDriver` 类。

```
class LocalValetDriver extends LaravelValetDriver
{
    /**
     * 判断驱动服务请求。
     *
     * @param string $sitePath
     * @param string $siteName
     * @param string $uri
     * @return bool
     */
    public function serves($sitePath, $siteName, $uri)
    {
        return true;
    }

    /**
     * 获取应用前端控制器绝对路径。
     *
     * @param string $sitePath
     * @param string $siteName
     * @param string $uri
     * @return string
     */
    public function frontControllerPath($sitePath, $siteName, $uri)
    {
        return $sitePath . '/public_html/index.php';
    }
}
```

其他 Valet 命令

命令	描述
<code>valet forget</code>	在某个站点根路径运行该命令可在根目录列表中移除该目录
<code>valet paths</code>	查看所有站点根路径
<code>valet restart</code>	重启
<code>valet start</code>	启动
<code>valet stop</code>	停止
<code>valet uninstall</code>	卸载

4 核心概念

Laravel 服务容器解析

- 简介
- 绑定
 - 绑定基础
 - 绑定接口至实现
 - 情境绑定
 - 标记
- 解析
 - Make 方法
 - 自动注入
- 容器事件

简介

Laravel 服务容器是管理类依赖和运行依赖注入的有力工具。依赖注入是一个花俏的名词，它实质上是指：类的依赖通过构造器或在某些情况下通过「setter」方法进行「注入」。

来看一个简单的例子：

```
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller
{
    /**
     * User Repository 的实现。
     *
     * @var UserRepository
     */
    protected $users;

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

```

    }

    /**
     * 显示指定用户的详细信息。
     *
     * @param int $id
     * @return Response
     */
    public function show($id)
    {
        $user = $this->users->find($id);

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

```

在这个例子中，控制器 `UserController` 需要从数据源中获取 `users`。因此，我们要注入可以获取 `users` 的服务。在这种情况下，`UserRepository` 可能是通过使用 [Eloquent](#) 来从数据库中获取 user 信息。因为 `UserRepository` 是通过注入获取，所以我们可以容易地切换为其他实现。当测试应用程序时，我们还可以轻松地「mock」，或创建假的 `UserRepository` 实例。

在构建强大的应用程序，和为 Laravel 核心贡献代码时，必须深入理解 Laravel 的服务容器。

绑定

绑定基础

几乎所有服务容器的绑定都是在 [服务提供者](#) 中进行的，所以下面的例子将示范在该情景中使用容器。

{tip} 但是，如果类没有依赖任何接口，那么就没有必要将类绑定到容器中了。容器绑定时，并不需要指定如何构建这些类，因为容器中会通过 PHP 的反射自动解析对象。

简单绑定

在服务提供者中，你经常可以通过 `$this->app` 属性访问容器。我们可以通过 `bind` 方法注册一个绑定，通过传递注册类或接口的名称、及返回该实例的 `closure` 作为参数：

```

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

```

注意，我们将获得的容器本身作为参数传递到解析器中，这样就可以使用容器来解决绑定对象对容器的子依赖。

绑定一个单例

通过 `singleton` 方法可以绑定一个只会被解析一次的类或接口到容器中。且后面的调用都会从容器中返回相同的实例：

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

绑定实例

你也可以使用 `instance` 方法绑定一个已经存在的对象至容器中。后面的调用都会从容器中返回指定的实例：

```
$api = new HelpSpot\API(new HttpClient);

$this->app->instance('HelpSpot\Api', $api);
```

绑定初始数据

有时，你的类不仅需要注入类，还需要注入一些原始数据，如一个整数。此时，你可以容易地通过情景绑定注入需要的任何值：

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

绑定接口至实现

服务容器有一个强大的功能，就是将一个指定接口的实现绑定到接口上。例如，如果我们有一个 `EventPusher` 接口和一个它的实现类 `RedisEventPusher`。编写完接口的 `RedisEventPusher` 实现类后，我们就可以在服务容器中像下面例子一样注册它：

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

这么做会告诉容器当一个类需要 `EventPusher` 接口的实例时，`RedisEventPusher` 的实例将会被容器注入。现在我们就可以在构造函数中，或者任何其他需要通过容器注入依赖的地方，使用 `EventPusher` 接口的类型提示：

```
use App\Contracts\EventPusher;

/**
 * Create a new class instance.
 *
```

```

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

```

情境绑定

有时候，你可能有两个类使用到相同的接口，但你希望每个类都能注入不同的实现。例如，两个控制器可能需要依赖不同的 `Illuminate\Contracts\Filesystem\Filesystem` 契约 的实现类。 Laravel 为此定义了一种简单、平滑的接口：

```

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

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

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

```

标记

有时候，你可能需要解析某个「分类」下的所有绑定。例如，你正在构建一个报表的聚合器，它需要接受不同 `Report` 接口的实例。分别注册了 `Report` 实例后，你可以使用 `tag` 方法为他们赋予一个标签：

```

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

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

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

```

一旦服务被标记后，你可以通过 `tagged` 方法轻松地将它们全部解析：

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

解析

`make` 方法

你可以在服务容器外使用 `make` 方法来获得一个实例化的类。它接受你希望解析的类或是接口名称作为参数：

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

如果你的代码不能直接使用 `$app` 变量，你可以使用全局的 `resolve` 助手：

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

自动注入

另外，并且也是重要的，你可以在类的构造函数中对依赖使用「类型提示」，依赖的类将会被容器自动进行解析，包括在 [控制器](#)，[事件监听器](#)，[队列任务](#)，[中间件](#) 等地方。事实上，这也是大部分类被容器解析的方式。

例如，你可以在控制器的构造函数中对应用程序定义的 `Repository` 使用类型提示。这样 `Repository` 实例会被自动解析并注入到类中：

```
<?php

namespace App\Http\Controllers;

use App\Users\Repository as UserRepository;

class UserController extends Controller
{
    /**
     * user repository 实例。
     */
    protected $users;

    /**
     * 控制器构造方法。
     *
     * @param UserRepository $users
     * @return void
     */
}
```

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

/**
 * 显示指定 ID 的用户信息。
 *
 * @param int $id
 * @return Response
 */
public function show($id)
{
    //
}
```

容器事件

每当服务容器解析一个对象时就会触发一个事件。你可以使用 `resolving` 方法监听这个事件：

```
$this->app->resolving(function ($object, $app) {
    // 解析任何类型的对象时都会调用该方法...
});

$this->app->resolving(HelperSpot\API::class, function ($api, $app) {
    // 解析「HelperSpot\API」类型的对象时调用...
});
```

如你所见，被解析的对象会被传递至回调中，让你在对象被传递到消费者前可以设置任何额外属性到对象上。

服务提供者

- [简介](#)
- [编写服务提供者](#)
 - [注册方法](#)
 - [引导方法](#)
- [注册提供者](#)
- [延迟的提供者](#)

简介

服务提供者是所有 Laravel 应用程序引导启动的中心所在。包括您自己的应用程序，以及所有的 Laravel 核心服务，都是通过服务提供者引导启动的。

但我们所谓的「引导启动」指的是什么？一般而言，我们指的是注册事务，包括注册服务容器绑定，事件监听器，中间件，甚至路由。服务提供者是配置应用程序的中心所在。

如果您打开 Laravel 的 `config/app.php` 文件，您会看到一个 `providers` 数组。这些是将被应用程序加载的服务提供者类。当然，它们其中有许多是「延迟」提供者，意味着它们不会每次请求都加载，只会按需加载。

在本概述中，您将学习如何编写自己的服务提供商，并在您的 Laravel 应用程序中注册它们。

编写服务提供者

所有服务提供者都需要继承 `Illuminate\Support\ServiceProvider` 类。大多数服务提供者都包含 `register` 和 `boot` 方法。在 `register` 方法中，您应该只能将事务绑定到 [服务容器](#)。不应该在 `register` 方法中尝试注册任何事件监听器，路由或者任何其他功能。

Artisan 命令行可以生成一个新的提供者通过 `make:provider` 命令：

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

注册方法

如前所述，在 `register` 方法中，您只能将事务绑定到 [服务容器](#)。不应该在 `register` 方法中尝试注册任何事件监听器，路由或者任何其他功能。否则，您可能会意外的使用到尚未加载的服务提供者提供的服务。

让我们来看看一个基本的服务提供者。在服务提供者的方法中，都会提供一个有服务容器访问权限的 `$app` 属性：

现在，让我们来看看基本的服务提供者。在你的任意一个服务提供者方法中，你总是可以通过访问 `$app` 属性使用服务容器：

```
<?php

namespace App\Providers;

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

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

服务提供者只定义了一个 `register` 方法，并且使用该方法在服务容器中定义了一个 `Riak\Connection` 类的实现。如果您不明白服务容器的工作原理，请查看 [服务容器](#)。

引导方法

那么，如果我们需要在我们的服务提供商中注册一个视图合成器呢？这应该在 `boot` 方法中完成。此方法在所有其他服务提供者均已注册之后调用，这意味着您可以访问已由框架注册的所有服务：

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class ComposerServiceProvider extends ServiceProvider
{
    /**
     * 引导启动任何应用程序服务
     *
     * @return void
     */
    public function boot()
    {
        view()->composer('view', function () {
            //
        });
    }
}
```

```

    }
}
```

引导方法依赖注入

您可以为服务提供者的 `boot` 方法设置类型提示。服务容器 会自动注入您需要的任何依赖：

```

use Illuminate\Contracts\Routing\ResponseFactory;

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

注册提供者

所有服务提供者都在 `config/app.php` 配置文件中注册。此文件包含一个服务提供者类数组 `providers`。默认情况下，它只会列出 Laravel 核心服务提供者类。这些服务提供者引导启动 Laravel 核心组件，例如邮件程序，队列，缓存和其他。

要注册您的提供程序，只需将其添加到数组：

```

'providers' => [
    // Other Service Providers

    App\Providers\ComposerServiceProvider::class,
],
```

延迟的提供者

如果您的提供程序 仅 在 服务容器 中注册绑定，您可以选择推迟其注册，直到真正需要注册绑定时。延迟加载服务提供者将提高应用程序的性能，因为它不会每次都从文件系统中加载。

Laravel 编译并保存了一份清单，包括由延缓服务提供者所提供的所有服务，以及其服务提供者类的类名。因此，只有在当您在试图解析其中的服务时，Laravel 才会加载该服务提供者。

若要推迟提供者的加载，请将 `defer` 属性设置为 `true`，并定义 `provides` 方法。`provides` 应该返回由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
{
    /**
     * 显示是否延迟提供程序的加载
     *
     * @var bool
     */
    protected $defer = true;

    /**
     * 注册一个服务提供者
     *
     * @return void
     */
    public function register()
    {
        $this->app->singleton(Connection::class, function ($app) {
            return new Connection($app['config']['riak']);
        });
    }

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

Laravel 的 Facades 介绍

- 简介
- 何时使用 Facades
 - Facades Vs. 依赖注入
 - Facades Vs. 辅助函数
- Facades 工作原理
- Facade 类参考

简介

Facades（读音：/fə'säd/）为应用程序的 [服务容器](#) 中可用的类提供了一个「静态」接口。Laravel 自带了很多 facades，几乎可以用来访问到 Laravel 中所有的服务。Laravel facades 实际上是服务容器中那些底层类的「静态代理」，相比于传统的静态方法，facades 在提供了简洁且丰富的语法同时，还带来了更好的可测试性和扩展性。

所有的 Laravel facades 都需要定义在命名空间 `Illuminate\Support\Facades` 下。所以，我们可以容易地向下面这样调用 facade：

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

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

在 Laravel 的文档中，很多示例代码都是使用 facades 来演示框架的各种特性的。

何时使用 Facades

Facades 有很多好处，它为我们使用 Laravel 的各种功能提供了简单，易记的语法，让你不需要记住长长的类名来实现依赖注入和手动配置。还有，因为它们对于 PHP 动态方法的独特用法，测试起来非常容易。

然而，在使用 facades 时，有些地方还需要特别注意。使用 facades 最主要的风险就是会引起类作用范围的膨胀。因为 facades 使用起来非常简单而且不需要注入，我们会不经意的在单个类中大量使用。它不会像使用依赖注入那样，使用的类越多，构造方法会越长，在视觉上就会引起注意，提醒你这个类有点庞大了。所以在使用 facades 的时候，要特别注意控制好类的大小，让类的作用范围保持短小。

{tip} 在开发与 Laravel 交互的第三方扩展包时，最好是在包中通过注入 [Laravel contracts](#)，而不是在包中通过 facades 来使用 Laravel 的类。因为扩展包不是在 Laravel 内部使用的，无法使用 Laravel's facade 的测试辅助函数。

Facades Vs. 依赖注入

依赖注入的一个主要的好处是可以切换注入类的具体实现。这在测试的时候很有用，因为你可以注入一个 mock 或者 stub，并且对在 stub 中被调用的各种方法进行断言。

通常，静态方法是不可以被 mock 或者 stub。但是，因为 facades 调用的是对象的动态方法，我们可以像测试注入类的实例一样测试 facades，例如，像下面的路由：

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

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

我们可以用下面的测试代码去验证 `Cache::get` 方法是否被调用，当传入预期的参数时。

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

/**
 * 一个基础功能的测试用例。
 *
 * @return void
 */
public function testBasicExample()
{
    Cache::shouldReceive('get')
        ->with('key')
        ->andReturn('value');

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

Facades Vs. 辅助函数

除了 facades，Laravel 包含一些「辅助函数」来实现一些常用的功能，比如生成视图，触发事件，调度任务或者发送 HTTP 响应。许多辅助函数的功能和对应的 facades 一样。例如，下面这个 facade 和辅助函数的作用是一样的：

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

return view('profile');
```

这里的 facades 和辅助函数是没有任何区别的。当你使用辅助函数时，你依然可以向使用对应的 facade 一样测试他们。例如，下面的路由：

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

在底层，辅助函数 `cache` 实际是调用 `Cache facade` 中的 `get` 方法。因此，尽管我们是在使用辅助函数，我们依然可以用下面的测试代码来验证是否方法被正确调用，在传入预期的参数时：

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

/**
 * 一个基础功能的测试用例。
 *
 * @return void
 */
public function testBasicExample()
{
    Cache::shouldReceive('get')
        ->with('key')
        ->andReturn('value');

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

Facades 工作原理

在 Laravel 应用中，一个 `facade` 就是一个提供访问容器中对象的类。其中核心的部件就是 `Facade` 类。不管是 Laravel 自带的 `Facades`，还是用户自定义的 `Facades`，都继承自 `Illuminate\Support\Facades\Facade` 类。

`Facade` 基类使用 `__callStatic()` 魔术方法在你的 `facades` 中延迟调用容器中对应对象的方法，在下面的例子中，调用了 Laravel 的缓存系统。在代码里，我们可能认为是 `Cache` 类中的静态方法 `get` 被调用了：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * 显示给定用户的大体信息。
     *
     * @param int $id
     * @return Response
     */
    public function showProfile($id)
    {
```

```

    $user = Cache::get('user:'.$id);

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

```

注意在代码的最上面，我们导入的是 `cache facade`。这个 `facade` 其实是我们获取底层 `Illuminate\Contracts\Cache\Factory` 接口实现的一个代理。我们通过这个 `facade` 调用的任何方法，都会被传递到 Laravel 缓存服务的底层实例中。

如果我们看一下 `Illuminate\Support\Facades\Cache` 这个类，你会发现类中根本没有 `get` 这个静态方法：

```

class Cache extends Facade
{
    /**
     * 获取组件在容器中注册的名称。
     *
     * @return string
     */
    protected static function getFacadeAccessor() { return 'cache'; }
}

```

其实，`Cache facade` 是继承了 `Facade` 基类，并且定义了 `getFacadeAccessor()` 方法。这个方法的作用是返回服务容器中对应名字的绑定内容。当用户调用 `Cache facade` 中的任何静态方法时，Laravel 会解析到服务容器中绑定的键值为 `cache` 实例对象，并调用这个对象对应的方法（在这个例子中就是 `get` 方法）。

Facade 类参考

在下面你可以找到每个 `facade` 类及其对应的底层类。这是一个查找给定 `facade` 类 API 文档的有用工具。也列出了绑定在 [服务容器](#) 中 `facade` 类对应的可用键值。

Facade	Class	Service Container Binding
App	Illuminate\Foundation\Application	app
Artisan	Illuminate\Contracts\Console\Kernel	artisan
Auth	Illuminate\Auth\AuthManager	auth
Blade	Illuminate\View\Compilers\BladeCompiler	blade.compiler
Bus	Illuminate\Contracts\Bus\Dispatcher	
Cache	Illuminate\Cache\Repository	cache
Config	Illuminate\Config\Repository	config
Cookie	Illuminate\Cookie\CookieJar	cookie

Crypt	Illuminate\Encryption\Encrypter	crypt
DB	Illuminate\Database\DatabaseManager	db
DB (Instance)	Illuminate\Database\Connection	
Event	Illuminate\Events\Dispatcher	events
File	Illuminate\Filesystem\Filesystem	files
Gate	Illuminate\Contracts\Auth\Access\Gate	
Hash	Illuminate\Contracts\Hashing\Hasher	hash
Lang	Illuminate\Translation\Translator	translator
Log	Illuminate\Log\Writer	log
Mail	Illuminate\Mail\Mailer	mailer
Notification	Illuminate\Notifications\ChannelManager	
Password	Illuminate\Auth\Passwords\PasswordBrokerManager	auth.password
Queue	Illuminate\Queue\QueueManager	queue
Queue (Instance)	Illuminate\Contracts\Queue\Queue	queue
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	
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	

契约 (Contracts)

- 简介
 - 契约 (Contracts) Vs. 门面 (Facades)
- 何时使用契约
 - 低耦合
 - 简单性
- 如何使用 Contracts
- Contract 参考

简介

Laravel 的 契约(Contracts) 是一系列框架用来定义核心服务的接口。例如，`Illuminate\Contracts\Queue\Queue` 契约中定义了队列任务所需的方法，而`Illuminate\Contracts\Mail\Mailer` 契约中定义了发送电子邮件所需的方法。

框架对每个契约都提供了相应的实现。例如，Laravel 为队列提供了各种驱动的实现，为邮件提供了由 `SwiftMailer` 驱动的实现。

所有 Laravel 契约都有相应的 [GitHub 库](#)，这给所有可用的契约提供了一个快速参考指南，同时也可单独作为低耦合的扩展包给其他包开发者去使用。

契约 (contracts) VS 门面 (facades)

[门面 \(facades\)](#) 和一些辅助函数提供了一种使用 Laravel 服务的简单方法，即不再需要通过类型约束和在服务容器之外解析契约。在大多数情况下，每个门面都有一个等效契约。

不像门面，门面不需要你在类构造函数中约束什么，契约则是需要你明显地定义依赖关系。一些开发人员喜欢契约这种方式明显地去定义它们的依赖关系，而另一些开发人员则更喜欢门面带来的便捷。

{tip} 对于大多数应用程序来说不管是使用门面还是契约只要你喜欢都行。但是，如果你正在构建一个扩展包，为了方便测试建议考虑使用契约比较好。

何时使用 contracts

综上所述，使用契约或者门面很大程度上归结于个人或者开发团队的喜好。不管是契约还是门面都可以创建出强大的、容易测试的 Laravel 应用程序。如果你长期关注类的单一职责，你会注意到使用契约和门面其实没多少实际意义上的区别。

然而，你可能还是会有几个关于契约的问题。像是，为什么要使用接口？使用接口会不会变得更加复杂？下面让我们简单阐述一下使用接口的原因：低耦合和简单性。

低耦合

首先，让我们回顾一些与缓存实现的高耦合代码。如下：

```
<?php

namespace App\Orders;

class Repository
{
    /**
     * 缓存实例。
     */
    protected $cache;

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

    /**
     * 按照Id检索订单。
     *
     * @param int $id
     * @return Order
     */
    public function find($id)
    {
        if ($this->cache->has($id)) {
            //
        }
    }
}
```

在这个类中，程序跟给定缓存实现之间是高耦合的。因为我们依赖于一个扩展包的特定缓存类。一旦这个扩展包的 API 被更改了，那我们的代码也必须得跟着改变。

同样的，如果想要将底层的缓存技术（Memcached）替换成另一种技术来实现（Redis），那又得再一次修改这个 `repository` 类。而 `repository` 类不应该知道这么多信息，比如关于谁提供了这些数据，或是他们又是如何提供的等等。

比起上面的做法，我们可以使用一个简单、和扩展包无关的接口来改进代码：

```
<?php

namespace App\Orders;
```

```

use Illuminate\Contracts\Cache\Repository as Cache;

class Repository
{
    /**
     * 缓存实例。
     */
    protected $cache;

    /**
     * 创建一个仓库实例。
     *
     * @param Cache $cache
     * @return void
     */
    public function __construct(Cache $cache)
    {
        $this->cache = $cache;
    }
}

```

现在，更改之后的代码没有与任何扩展包耦合，甚至是 Laravel。而契约扩展包不包含实现和依赖，你可以轻松地对任何契约包进行实现，比如不需要修改任何关于缓存的代码就可以替换缓存实现。

简单性

当所有的 Laravel 服务都使用简洁的接口定义，就能够很容易决定一个服务需要提供的功能。可以将契约视为说明框架特色的简洁文档。

除此之外，当依赖的接口足够简洁时，代码的可读性和可维护性会大大提高。比起搜索一个大型复杂的类里有哪些可用的方法，不如检索一个简单、干净的接口来参考更妥当。

如何使用 contracts

那么，如何获取一个契约的实现呢？这其实很简单。

Laravel 中的许多类型的类都是通过 [服务容器](#) 解析出来的。包括控制器、事件监听器、中间件、任务队列，甚至路由的闭包。所以说，要获得一个契约的实现，你只需要解析在类的构造函数中相应的类型约束即可。

例如，看看这个事件监听器：

```

<?php

namespace App\Listeners;

use App\User;

```

```

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

class CacheOrderInformation
{
    /**
     * Redis 数据库实现。
     */
    protected $redis;

    /**
     * 创建事件处理器实例。
     *
     * @param Database $redis
     * @return void
     */
    public function __construct(Database $redis)
    {
        $this->redis = $redis;
    }

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

```

当事件监听器被解析时，服务容器会从构造函数里读取到类型约束，并注入对应的值。想了解关于容器的注册绑定，可以查看 [服务容器](#)。

Contract 参考

下面的表格提供了 Laravel 契约及其对应的门面的参考：

Contract	References Facade
Illuminate\Contracts\Auth\Factory	Auth
Illuminate\Contracts\Auth>PasswordBroker	Password
Illuminate\Contracts\Bus\Dispatcher	Bus
Illuminate\Contracts\Broadcasting\Broadcaster	
Illuminate\Contracts\Cache\Repository	Cache

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

5 HTTP层

Laravel HTTP 路由功能

- 基本路由
- 路由参数
 - 必选路由参数
 - 可选路由参数
 - 正则表达式约束
- 命名路由
- 路由组
 - 中间件
 - 命名空间
 - 子域名路由
 - 路由前缀
- 路由模型绑定
 - 隐式绑定
 - 显式绑定
- 表单方法伪造
- 获取当前路由信息

基本路由

构建最基本的路由只需要一个 URI 与一个 闭包，这里提供了一个非常简单优雅的定义路由的方法：

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

默认路由文件

所有的 Laravel 路由都在 `routes` 目录中的路由文件中定义，这些文件都由框架自动加载。在 `routes/web.php` 文件中定义你的 web 页面路由。这些路由都会应用 `web` 中间件组，其提供了诸如 `Session` 和 `CSRF` 保护等特性。定义在 `routes/api.php` 中的路由都是无状态的，并且会应用 `api` 中间件组。

大多数的应用构建，都是以在 `routes/web.php` 文件定义路由开始的。

可用的路由方法

我们可以注册路由来响应所有的 HTTP 操作：

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

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

有的时候你可能需要注册一个可响应多个 HTTP 方法的路由，这时你可以使用 `match` 方法，也可以使用 `any` 方法注册一个实现响应所有 HTTP 的请求的路由：

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

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

CSRF 保护

任何指向 `web` 中 `POST`, `PUT` 或 `DELETE` 路由的 HTML 表单请求都应该包含一个 CSRF 令牌，否则，这个请求将会被拒绝。更多的关于 CSRF 的说明在 [CSRF 说明文档](#)：

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

路由参数

必选路由参数

当然，有时我们需要在路由中捕获一些 URL 片段。例如，我们需要从 URL 中捕获用户的 ID，我们可以这样定义路由参数：

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

也可以根据需要在路由中定义多个参数：

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

路由的参数通常都会被放在 `{}` 内，并且参数名只能为字母，当运行路由时，参数会通过路由闭包来传递。

注意： 路由参数不能包含 `-` 字符。请用下划线（`_`）替换。

可选路由参数

声明路由参数时，如需指定该参数为可选，可以在参数后面加上 `?` 来实现，但是相应的变量必须有默认值：

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

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

正则表达式约束

你可以使用 `where` 方法来规范你的路由参数格式。`where` 方法接受参数名称和定义参数约束规则的正则表达式：

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

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

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

全局约束

如果你希望路由参数在全局范围内都遵循一个确定的正则表达式约束，则可以使用 `pattern` 方法。你应该在 `RouteServiceProvider` 的 `boot` 方法里定义这些模式：

```
/**
 * 定义你的路由模型绑定, pattern 过滤器等。
 *
 * @return void
 */
public function boot()
{
    Route::pattern('id', '[0-9]+');

    parent::boot();
}
```

Pattern 一旦被定义，便会自动应用到所有使用该参数名称的路由上：

```
Route::get('user/{id}', function ($id) {
    // 仅在 {id} 为数字时执行...
});
```

命名路由

命名路由可以方便的生成 URL 或者重定向到指定的路由，你可以在定义路由后使用 name 方法实现：

```
Route::get('user/profile', function () {
    //
})->name('profile');
```

你还可以为控制器方法指定路由名称：

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

为命名路由生成 URL

为路由指定了名称后，我们可以使用全局辅助函数 route 来生成 URL 或者重定向到该条路由：

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

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

如果是有定义参数的命名路由，可以把参数作为 route 函数的第二个参数传入，指定的参数将会自动插入到 URL 中对应的位置：

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

$url = route('profile', ['id' => 1]);
```

路由组

路由组允许共享路由属性，例如中间件和命名空间等，我们没有必要为每个路由单独设置共有属性，共有属性会以数组的形式放到 Route::group 方法的第一个参数中。

中间件

要给路由组中定义的所有路由分配中间件，可以在路由组中使用 `middleware` 键，中间件将会依照列表内指定的顺序运行：

```
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

另一个常见的例子是，为控制器组指定公共的 PHP 命名空间。这时使用 `namespace` 参数来指定组内所有控制器的公共命名空间：

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

请记住，默认 `RouteServiceProvider` 会在命名空间组中引入你的路由文件，让你不用指定完整的 `App\Http\Controllers` 命名空间前缀就能注册控制器路由，因此，我们在定义的时候只需要指定命名空间 `App\Http\Controllers` 以后的部分。

子域名路由

路由组也可以用作子域名的通配符，子域名可以像 URI 一样当作路由组的参数，因此允许把捕获的子域名一部分用于我们的路由或控制器。可以使用路由组属性的 `domain` 键声明子域名。

```
Route::group(['domain' => '{account}.myapp.com'], function () {
    Route::get('user/{id}', function ($account, $id) {
        //
    });
});
```

路由前缀

通过路由组数组属性中的 `prefix` 键可以给每个路由组中的路由加上指定的 URI 前缀，例如，我们可以给路由组中所有的 URI 加上路由前缀 `admin`：

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

```
    });
});
```

路由模型绑定

当向路由控制器中注入模型 ID 时，我们通常需要查询这个 ID 对应的模型，Laravel 路由模型绑定提供了一个方便的方法自动将模型注入到我们的路由中，例如，除了注入一个用户的 ID，你也可以注入与指定 ID 匹配的完整 `User` 类实例。

隐式绑定

Laravel 会自动解析定义在路由或控制器方法（方法包含和路由片段匹配的已声明类型变量）中的 Eloquent 模型，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，将会自动生成产生一个 404 HTTP 响应。

自定义键名

如果你想要隐式模型绑定除 `id` 以外的数据库字段，你可以重写 Eloquent 模型类的 `getRouteKeyName` 方法：

```
/**
 * 为路由模型获取键名
 *
 * @return string
 */
public function getRouteKeyName()
{
    return 'slug';
}
```

显式绑定

使用路由的 `model` 方法来为已有参数声明 class。你应该在 `RouteServiceProvider` 类中的 `boot` 方法内定义这些显式绑定：

```
public function boot()
{
```

```
parent::boot();

Route::model('user', App\User::class);
}
```

接着，定义包含 `{user}` 参数的路由：

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

因为我们已经绑定 `{user}` 参数至 `App\User` 模型，所以 `User` 实例将被注入该路由。举个例子，一个至 `profile/1` 的请求会注入 ID 为 1 的 `User` 实例。

注意：如果在数据库不存在对应 ID 的数据，就会自动抛出一个 404 异常。

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：

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

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

表单方法伪造

HTML 表单没有支持 `PUT`、`PATCH` 或 `DELETE` 动作。所以在定义要在 HTML 表单中调用的 `PUT`、`PATCH` 或 `DELETE` 路由时，你将需要在表单中增加隐藏的 `_method` 字段。`_method` 字段的值将被作为 HTTP 的请求方法使用：

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

你也可以使用辅助函数 `method_field` 来生成隐藏的输入字段 `_method`：

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

获取当前路由信息

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

```
$route = Route::current();  
  
$name = Route::currentRouteName();  
  
$action = Route::currentRouteAction();
```

完整的方法列表请参考 [Route facade](#) 和 [Route 实例](#)

Laravel 的路由中间件

- [简介](#)
- [创建中间件](#)
- [注册中间件](#)
 - [全局中间件](#)
 - [为路由指定中间件](#)
 - [中间件组](#)
- [中间件参数](#)
- [Terminable 中间件](#)

简介

Laravel 中间件提供了一种方便的机制来过滤进入应用的 HTTP 请求，例如，Laravel 包含验证用户身份权限的中间件。如果用户没有通过身份验证，中间件会重定向到登录页，引导用户登录。反之，中间件将允许该请求继续传递到应用程序。

当然，除了身份验证以外，中间件还可以被用来执行各式各样的任务，如：CORS 中间件负责为所有即将离开应用的响应添加适当的头信息；日志中间件可以记录所有传入应用的请求。

Laravel 已经内置了一些中间件，包括身份验证、CSRF 保护等。所有的中间件都放在 `app/Http/Middleware` 目录内。

创建中间件

使用 `make:middleware` 这个 Artisan 命令创建新的中间件：

```
php artisan make:middleware CheckAge
```

该命令将会在 `app/Http/Middleware` 目录内新建一个 `CheckAge` 类。在这个中间件内，我们仅允许请求的 `age` 参数大于 200 时访问该路由，否则，会将用户请求重定向到 `home` URI。

```
<?php

namespace App\Http\Middleware;

use Closure;

class CheckAge
{
    /**
     * 处理传入的请求
     *
     * @param \Illuminate\Http\Request $request
     */
    public function handle($request, Closure $next)
    {
        if ($request->age > 200) {
            return redirect('/home');
        }

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

```

 * @param \Closure $next
 * @return mixed
 */
public function handle($request, Closure $next)
{
    if ($request->age <= 200) {
        return redirect('home');
    }

    return $next($request);
}

}

```

如你所见，若请求参数 `age` 小于等于 `200`，中间件将返回给客户端 HTTP 重定向，反之应用程序才会继续处理该请求。若将请求继续传递到应用程序（即允许通过中间件验证），只需将 `$request` 作为参数调用 `$next` 回调函数。

最好将中间件想象为一系列的「层」，HTTP 请求必须经过它们才会触发您的应用程序。每一层都可以检测接收的请求，甚至可以完全拒绝请求访问您的应用。

前置中间件 / 后置中间件

中间件运行在请求之前或之后取决于中间件本身。例如，以下中间件会在请求被应用处理 之前 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，这个中间件会在请求被应用处理 之后 执行它的任务：

```

<?php

namespace App\Http\Middleware;

```

```

use Closure;

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

        // 执行动作

        return $response;
    }
}

```

注册中间件

全局中间件

如果你希望访问你应用的每个 HTTP 请求都经过某个中间件，只需将该中间件类列入 `app/Http/Kernel.php` 类里的 `$middleware` 属性。

为路由指定中间件

如果你想为特殊的路由指定中间件，首先应该在 `app/Http/Kernel.php` 文件内为该中间件指定一个键值。默认情况下 `Kernel` 类的 `$routeMiddleware` 属性已经包含了 Laravel 内置的中间件条目。加入自定义的中间件，只需把它附加到此列表并指定你定义的 键值 即可。例如：

```

// App\Http\Kernel 类内

protected $routeMiddleware = [
    'auth' => \Illuminate\Auth\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
];

```

一旦在 `HTTP kernel` 文件内定义了中间件，即可使用 `middleware` 方法将中间件分配给路由：

```

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

```

为路由指定多个中间件：

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

也可使用完整类名指定中间件：

```
use App\Http\Middleware\CheckAge;

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

中间件组

有时您可能想要将多个中间件分组到同一个 键值 下，从而使它们更方便地分配给路由，你可以使用 HTTP kernel 的 `$middlewareGroups` 属性来实现。

Laravel 带有开箱即用的 `web` 和 `api` 中间件，包含了可能应用到 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,
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],
    'api' => [
        'throttle:60,1',
        'auth:api',
    ],
];
```

中间件组可以像单个中间件一样使用相同的语法指定给路由和控制器。重申，路由组仅仅是为了使一次将多个中间件指定给某个路由的实现变得更加方便。

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

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

{tip} 开箱即用的 `web` 中间件组被自动应用于 `RouteServiceProvider` 中定义的 `routes/web.php` 路由组。

中间件参数

中间件也可以接受其他附加的参数。例如，如果应用需要在运行特定操作前验证该用户具备该操作的权限的「角色」，你可以新建一个 `CheckRole` 中间件，该中间件接收「角色」名字作为附加参数。

附加的中间件参数将在 `$next` 参数之后被传入：

```
<?php

namespace App\Http\Middleware;

use Closure;

class CheckRole
{
    /**
     * 处理传入的请求
     *
     * @param \Illuminate\Http\Request $request
     * @param Closure $next
     * @param string $role
     * @return mixed
     */
    public function handle($request, Closure $next, $role)
    {
        if (! $request->user()->hasRole($role)) {
            // Redirect...
        }

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

定义路由时，指定中间件参数可以通过冒号 `:` 来隔开中间件与参数，多个参数可以使用逗号分隔：

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

Terminable 中间件

有些时候中间件需要在 HTTP 响应发送到浏览器后运行来处理一些任务。比如，Laravel 内置的「session」中间件存储的 session 数据是在响应被发送到浏览器之后才进行写入的。想实现这一点，你需要在中间件中定义一个 `terminate` 方法，它会在响应发送后自动被调用：

```
<?php

namespace Illuminate\Session\Middleware;

use Closure;

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

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

`terminate` 方法必需接收请求及响应两个参数。一旦定义了 terminable 中间件，你便需要将它增加到 HTTP kernel 文件的全局中间件清单列表中。

中间件的 `terminate` 调用时，Laravel 会从 [服务容器](#) 中解析一个全新的中间件实例。如果你想在 `handle` 和 `terminate` 被调用时使用同一个中间件实例，可使用容器的 `singleton` 方法向容器注册中间件。

Laravel 下的伪造跨站请求保护 CSRF

- 简介
- CSRF 令牌和 Vue
- CSRF 白名单
- X-CSRF-Token
- X-XSRF-Token

简介

Laravel 提供了简单的方法使你的应用免受 跨站请求伪造 (CSRF) 的袭击。跨站请求伪造是一种恶意的攻击，它凭借已通过身份验证的用户身份来运行未经过授权的命令。

Laravel 为每个活跃用户的 Session 自动生成一个 CSRF 令牌。该令牌用来核实应用接收到的请求是通过身份验证的用户出于本意发送的。

任何情况下在你的应用程序中定义 HTML 表单时都应该包含 CSRF 令牌隐藏域，这样 CSRF 保护中间件才可以验证请求。辅助函数 `csrf_field` 可以用来生成令牌字段：

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

包含在 `web` 中间件组里的 `VerifyCsrfToken` 中间件会自动验证请求里的令牌 `token` 与 Session 中存储的令牌 `token` 是否匹配。

CSRF 令牌和 Vue

如果你使用 `Vue.js` 框架，没有 `make:auth` 命令提供的身份验证过渡，那么你需要在你应用的主要布局中手动定义一个 `Laravel` Javascript 对象。这个对象会指定 Vue 在做请求时需要的 CSRF 令牌：

```
<script>
    window.Laravel = {!! json_encode([
        'csrfToken' => csrf_token(),
    ]) !!};
</script>
```

CSRF 白名单

有时候你可能希望设置一组并不需要 CSRF 保护的 URI。例如，如果你正在使用 `Stripe` 处理付款并使用了他们的 webhook 系统，你会需要将 Stripe webhook 处理的路由排除在 CSRF 保护外，因为 Stripe 并不知道发送给你路由的 CSRF 令牌是什么。

一般地，你可以把这类路由放到 `web` 中间件外，因为 `RouteServiceProvider` 适用于 `routes/web.php` 中的所有路由。不过如果一定要这么做，你也可以将这类 URI 添加到 `VerifyCsrfToken` 中间件中的 `$except` 属性来排除对这类路由的 CSRF 保护：

```
<?php

namespace App\Http\Middleware;

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

class VerifyCsrfToken extends BaseVerifier
{
    /**
     * 这些 URI 会被免除 CSRF 验证
     *
     * @var array
     */
    protected $except = [
        'stripe/*',
    ];
}
```

X-CSRF-TOKEN

除了检查 POST 参数中的 CSRF token 外，`VerifyCsrfToken` 中间件还会检查 `X-CSRF-TOKEN` 请求头。你可以将令牌保存在 HTML `meta` 标签中：

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

一旦创建了 `meta` 标签，你就可以使用类似 jQuery 的库将令牌自动添加到所有请求的头信息中。这可以为您基于 AJAX 的应用提供简单、方便的 CSRF 保护。

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

X-XSRF-TOKEN

Laravel 将当前的 CSRF 令牌存储在由框架生成的每个响应中包含的一个 `XSRF-TOKEN` cookie 中。你可以使用该令牌的值来设置 `X-XSRF-TOKEN` 请求头信息。

这个 cookie 作为头信息发送主要是为了方便，因为一些 JavaScript 框架，如 Angular，会自动将其值添加到 `X-XSRF-TOKEN` 头中。

Laravel 的 HTTP 控制器

- 简介
- 基础控制器
 - 定义控制器
 - 控制器与命名空间
 - 单一操作控制器
- 控制器中间件
- 资源控制器
 - 部分资源路由
 - 命名资源路由
 - 命名资源路由参数
 - 本地化资源 URI
 - 附加资源控制器
- 依赖注入与控制器
- 路由缓存

简介

除了在路由文件中以闭包的形式定义所有的请求处理逻辑外，你可能还想使用控制器类来组织此类操作。控制器能够将相关的请求处理逻辑组成一个单独的类。控制器被存放在 `app/Http/Controllers` 目录下。

基础控制器

定义控制器

以下是一个基础控制器类的例子。需要注意的是，该控制器继承了 Laravel 内置的基础控制器类。该基础类提供了一些便捷的方法，比如 `middleware` 方法，该方法可以用来给控制器操作添加中间件：

```
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller
{
    /**
     * 展示给定用户的信息。
     *
     * @param int $id
     * @return Response
  
```

```
 */
public function show($id)
{
    return view('user.profile', ['user' => User::findOrFail($id)]);
}
```

你可以这样定义一个指向该控制器操作的路由：

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

现在，当请求和此特定路由的 URI 匹配时，`UserController` 类的 `show` 方法就会被执行。当然，路由参数也会被传递至该方法。

{tip} 控制器并不是一定 要继承基础类。只是，你将无法使用一些便捷的功能，比如 `middleware`，`validate` 和 `dispatch` 方法。

控制器与命名空间

这一点很重要，我们在定义控制器路由时，不必指定完整的控制器命名空间。`RouteServiceProvider` 会在一个包含命名空间的路由组中加载路由文件，因此我们只需要指定类名中 `App\Http\Controllers` 命名空间之后的部分就可以了。

如果你选择将控制器存放在 `App\Http\Controllers` 目录下，只需简单地使用相对于根命名空间 `App\Http\Controllers` 的特定类名。因此，如果你的完整控制器类是 `App\Http\Controllers\Photos\AdminController`，你应该用这种方式注册指向该控制器的路由：

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

单一操作控制器

如果想定义一个只处理单个操作的控制器，你可以在控制器中只设置一个 `__invoke` 方法：

```
<?php

namespace App\Http\Controllers;

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

class ShowProfile extends Controller
{
    /**
     * 展示给定用户的信息。
     *
     * @param int $id

```

```

    * @return Response
    */
public function __invoke($id)
{
    return view('user.profile', ['user' => User::findOrFail($id)]);
}
}

```

为单一操作控制器注册路由时，无需指定方法：

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

控制器中间件

[中间件](#) 可以在路由文件中指定给控制器路由：

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

然而，在控制器的构造方法中指定中间件会更为便捷。在控制器构造方法中使用 `middleware` 方法，你可以很容易地将中间件指定给控制器操作。你甚至可以约束中间件只对控制器类中的某个特定方法生效：

```

class UserController extends Controller
{
    /**
     * 创建一个新的控制器实例。
     *
     * @return void
     */
    public function __construct()
    {
        $this->middleware('auth');

        $this->middleware('log')->only('index');

        $this->middleware('subscribed')->except('store');
    }
}

```

控制器也允许你使用闭包的方式注册中间件。这提供了一种很便捷地为单个控制器定义中间件的方式，而不用定义一个完整的中间件类：

```

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

    return $next($request);
}

```

```
});
```

{tip} 你可能将中间件指定到控制器的部分操作上，然而，这会使你的控制器过于臃肿。换个角度，考虑将控制器分成多个更小的控制器。

资源控制器

Laravel 资源路由可以将典型的「CURD」路由指定到一个控制器上，仅需一行代码就可以实现。比如，你可能希望创建一个控制器来处理所有应用保存的「相片」的 HTTP 请求。使用 Artisan 命令 `make:controller`，就能快速创建这样一个控制器：

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

这个命令会在 `app/Http/Controllers/PhotoController.php` 中生成一个控制器，该控制器包含了各种可用的资源操作方法。

接下来，你可以给控制器注册一个资源路由：

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

这个路由声明会创建多个路由来处理各种各样的资源操作。前面生成的控制器已经包含了这些操作的方法，还包括了 HTTP 动作和操作 URI 的注释。

资源控制器操作处理

动作	URI	操作	路由名称
GET	/photos	index	photos.index
GET	/photos/create	create	photos.create
POST	/photos	store	photos.store
GET	/photos/{photo}	show	photos.show
GET	/photos/{photo}/edit	edit	photos.edit
PUT/PATCH	/photos/{photo}	update	photos.update
DELETE	/photos/{photo}	destroy	photos.destroy

指定资源模型

如果你使用了路由模型绑定，并且想在资源控制器的方法中对某个模型实例做类型约束，你可以在生成控制器的时候使用 `--model` 选项：

```
php artisan make:controller PhotoController --resource --model=Photo
```

伪造表单方法

送 `PUT` , `PATCH` 或者 `DELETE` 请求, 你需要添加一个 `_method` 隐藏域字段来伪造 HTTP 动作。 `method_field` 辅助函数可以为你创建这个字段:

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

部分资源路由

声明资源路由的时候, 你可以指定控制器处理部分操作, 而不必使用全部默认的操作:

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

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

命名资源路由

默认地, 所有的资源路由操作都有一个路由名称; 不过你可以在参数选项中传入一个 `names` 数组来重写这些名称:

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

命名资源路由参数

默认地, `Route::resource` 会基于资源名称的「单数」形式生成路由参数。你可以在选项数组中传入 `parameters` 参数, 实现每个资源基础中参数名称的重写。`parameters` 应该是一个将资源名称和参数名称联系在一起的数组:

```
Route::resource('user', 'AdminUserController', ['parameters' => [
    'user' => 'admin_user'
]]);
```

上例将会为 `show` 方法的路由生成如下的 URI:

```
/user/{admin_user}
```

本地化资源 URI

默认地，`Route::resource` 将会用英文动词创建资源 URI。如果你想本地化 `create` 和 `edit` 的动作名，可以使用 `Route::resourceVerb` 方法，可以在 `AppServiceProvider` 的 `boot` 方法中实现：

```
use Illuminate\Support\Facades\Route;

/**
 * 自定义任何应用服务。
 *
 * @return void
 */
public function boot()
{
    Route::resourceVerbs([
        'create' => 'crear',
        'edit' => 'editar',
    ]);
}
```

动作名被自定义后，像 `Route::resource('fotos', 'PhotoController')` 这样注册的资源路由将会产生如下的 URI：

```
/fotos/crear  
/fotos/{foto}/editar
```

附加资源控制器

如果你想在默认的资源路由之外增加资源控制器路由，你应该在调用 `Route::resource` 之前定义这些路由；否则，`resource` 方法定义的路由可能会不小心覆盖你的附加路由：

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

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

{tip} 记住保持控制器的专一性。如果你需要典型的资源操作之外的方法，考虑将你的控制器分成两个更小的控制器吧。

依赖注入与控制器

构造方法注入

Laravel 使用 [服务容器](#) 来解析所有的控制器。因此，你可以在控制器的构造方法中对任何依赖使用类型约束，声明的依赖会自动被解析并注入控制器实例中：

```

<?php

namespace App\Http\Controllers;

use App\Repositories\UserRepository;

class UserController extends Controller
{
    /**
     * 用户 repository 实例。
     */
    protected $users;

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

```

当然，你也可以对任何的 [Laravel contract](#) 使用类型约束。当容器解析 contract 的时候，就会使用类型约束。直接将依赖注入控制器可能会提供更好的可测试性，但这取决于你的项目的具体情况。

方法注入

除了构造方法注入之外，你还可以在控制器方法中使用依赖类型约束。一个常见的用法就是将 `Illuminate\Http\Request` 实例注入控制器方法中：

```

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * 保存一个新用户。
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {

```

```

    $name = $request->name;

    //
}

}

```

如果控制器方法需要从路由参数中获取输入内容，只需在其他依赖后列出路由参数即可。比如，路由定义如下：

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

通过以下方式定义控制器方法，可以让你在使用 `Illuminate\Http\Request` 类型约束的同时仍然可以获取参数 `id`：

```

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * 更新给定用户的信息。
     *
     * @param Request $request
     * @param string $id
     * @return Response
     */
    public function update(Request $request, $id)
    {
        //
    }
}

```

路由缓存

{note} 基于闭包的路由并不能被缓存。如果要使用路由缓存，你必须将所有的闭包路由转换成控制器类。

如果你的应用只用到了基于控制器的路由，那么你应该充分利用 Laravel 的路由缓存。使用路由缓存将极大地减少注册全部应用路由的时间。某些情况下，路由注册甚至可以快一百倍。要生成路由缓存，只需在 Artisan 命令行中执行 `route:cache` 命令：

```
php artisan route:cache
```

运行这个命令之后，每一次请求的时候都将会加载缓存的路由文件。记住，如果添加了新的路由，你需要刷新路由缓存。因此，你应该只在项目部署时才运行 `route:cache` 命令：

你可以使用 `route:clear` 命令清除路由缓存：

```
php artisan route:clear
```

Laravel 的 HTTP 请求 Request

- 获取请求
 - 请求路径 & 方法
 - PSR-7 请求
- 输入数据的预处理和规范化
- 获取输入数据
 - 旧输入数据
 - Cookies
- 文件资源
 - 获取上传文件
 - 储存上传文件

获取请求

要通过依赖注入的方式来获取当前 HTTP 请求的实例，你应该在控制器方法中使用 `Illuminate\Http\Request` 类型提示。当前的请求实例将通过 [服务容器](#) 自动注入：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * 储存一个新用户。
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        $name = $request->input('name');

        //
    }
}
```

依赖注入 & 路由参数

如果控制器方法也有输入数据是从路由参数中传入的，只需将路由参数置于其他依赖之后。例如，你的路由是这样定义的：

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

只要像下方一样定义控制器方法，你就可以使用 `Illuminate\Http\Request` 类型提示了，同时获取到路由参数 `id`：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * 更新指定的用户。
     *
     * @param Request $request
     * @param string $id
     * @return Response
     */
    public function update(Request $request, $id)
    {
        //
    }
}
```

通过路由闭包获取请求

你也可以在一个路由闭包中使用 `Illuminate\Http\Request` 类型提示。当它执行时，服务容器会自动注入当前请求到闭包中：

```
use Illuminate\Http\Request;

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

请求路径 & 方法

`Illuminate\Http\Request` 的实例提供了多种方法去检查应用程序的 HTTP 请求，Laravel 的 `Illuminate\Http\Request` 继承了 `Symfony\Component\HttpFoundation\Request` 类。下面是该类几个有用的方法：

获取请求路径

`path` 方法返回请求路径信息。所以，如果收到的请求目标地址是 `http://domain.com/foo/bar`，那么 `path` 将会返回 `foo/bar`：

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

`is` 方法可以验证收到的请求路径和指定规则是否匹配。使用这个方法的时候你也可以传递一个 `*` 字符作为通配符：

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

获取请求的 URL

你可以使用 `url` 或 `fullUrl` 方法去获取传入请求的完整 URL。`url` 方法返回不带有查询字符串的 URL，而 `fullUrl` 方法的返回值包含查询字符串：

```
// Without Query String...
$url = $request->url();

// With Query String...
$url = $request->fullUrl();
```

获取请求方法

对于传入的请求 `method` 方法将返回 HTTP 的请求方式。你也可以使用 `isMethod` 方法去验证 HTTP 的请求方式与指定规则是否相配：

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

if ($request->isMethod('post')) {
    //
}
```

PSR-7 请求

[PSR-7 标准](#)制定的 HTTP 消息接口包含了请求和响应。如果你想使用一个 PSR-7 请求来代替一个 Laravel 请求，那么你首先要安装几个函数库。Laravel 使用了 Symfony 的 HTTP 消息桥接组件，将原来的 Laravel 请求和响应转换到 PSR-7 所兼容的实现：

```
composer require symfony/psr-http-message-bridge
composer require zendframework/zend-diactoros
```

安装完这些库后，就可以在路由闭包或控制器中，简单的对请求类型使用类型提示来获取 PSR-7 的请求：

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

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

{tip} 如果你从路由或者控制器返回了一个 PSR-7 响应实例，那么这个实例将被自动转换回一个 Laravel 响应实例，同时由框架显示。

输入数据的预处理和规范化

在 Laravel 的全局中间件中默认包含了 `TrimStrings` 和 `ConvertEmptyStringsToNull` 两个中间件。这些中间件被列在 `App\Http\Kernel` 类中。它们会自动处理所有请求中传入的字符串字段，比如将空的字符串字段转变成 `null` 值。你再也不用担心路由和控制器中数据规范化的问题。

如果你想停用这些功能，你可以在 `App\Http\Kernel` 类的 `$middleware` 属性中移除这些中间件。

获取输入数据

获取所有输入数据

你可以使用 `all` 方法以 `数组` 形式获取到所有输入数据：

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

获取指定输入值

你可以通过 `Illuminate\Http\Request` 的实例，借助几个简单的方法，就可以获取到用户输入的所有数据。而不需要担心发起请求时使用了哪一种请求方式，`input` 方法通常被用来获取用户输入数据：

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

你可以给 `input` 方法的第二个参数传入一个默认值。当请求的输入数据不存在于此请求时，返回该默认值：

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

如果传输表单数据中包含「数组」形式的数据，那么可以使用「点」语法来获取数组：

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

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

通过动态属性获取输入数据

你也可以通过 `Illuminate\Http\Request` 实例的动态属性来获取用户输入的数据。例如，如果你应用程序表单中包含 `name` 字段，那么可以像这样访问提交的值：

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

Laravel 在处理动态属性的优先级是，先从请求的数据中查找，没有的话再到路由参数中找。

获取 JSON 输入信息

当你发送 JSON 请求到应用时，只要请求表头中设置了 `Content-Type` 为 `application/json`，你就可以直接从 `Input` 方法中获取 JSON 数据。你也可以通过「点」语法来读取 JSON 数组：

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

获取部分输入数据

如果你需要获取输入数据的子集，则可以用 `only` 和 `except` 方法。这两个方法都接收单个数组或动态列表作为参数：

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

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

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

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

`only` 方法会返回所有你指定的键值对，即使这个键在输入数据中并不存在。如果一个键在输入数据中并不存在时，它对应的值是 `null`。当你想要获取请求中实际存在的输入数据时，你可以使用 `intersect` 方法。

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

确定是否有输入值

要判断请求是否存在该数据，可以使用 `has` 方法。当数据存在 并且 字符串不为空时，`has` 方法就会返回 `true`：

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

日输入数据

Laravel 允许你将本次的输入数据保留到下一次请求发送前。这个特性在表单验证错误后重新填写表单相当有用。但是，如果你使用了 Laravel 的 [验证特性](#)，你就不需要在手动实现这些方法，因为 Laravel 内置的验证工具会自动调用他们。

将输入数据闪存至 Session

`Illuminate\Http\Request` 的 `flash` 方法会将当前输入的数据存进 `session` 中，因此下次用户发送请求到应用程序时就可以使用它们：

```
$request->flash();
```

你也可以使用 `flashOnly` 和 `flashExcept` 方法将当前请求数据的子集保存到 `session`。这些方法对敏感信息的保护非常有用：

```
$request->flashOnly(['username', 'email']);  
  
$request->flashExcept('password');
```

闪存输入数据到 Session 后重定向

你可能需要把输入数据闪存到 `session` 并重定向到前一个页面，这时只需要在重定向方法后加上 `withInput` 即可：

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

获取旧输入数据

若要获取上一次请求后所闪存的输入数据，则可以使用 `Request` 实例中的 `old` 方法。`old` 方法提供一个简便的方式从 `Session` 取出被闪存的输入数据：

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

Laravel 也提供了全局辅助函数 `old`。如果你要在 [Blade 模板](#) 中显示旧输入数据，可以使用更加方便的 `old` 辅助函数。如果旧数据不存在，则返回 `null`：

```
<input type="text" name="username" value="{{ old('username') }}">
```

Cookies

从请求中获取 Cookie 值

Laravel 框架创建的每个 cookie 都会被加密并且加上认证标识，这代表着用户擅自更改的 cookie 都会失效。若要从此次请求获取 cookie 值，你可以使用 `Illuminate\Http\Request` 实例中的 `cookie` 方法：

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

将 Cookies 附加到响应

你可以使用 `cookie` 方法附加一个 cookie 到 `Illuminate\Http\Response` 实例。有效 cookie 应该传递字段名称，字段值和过期时间给这个方法：

```
return response('Hello World')->cookie(
    'name', 'value', $minutes
);
```

`cookie` 方法还可以接受更多参数，只是使用频率较低。通常作用是传递参数给 PHP 原生 [设置 cookie](#) 方法：

```
return response('Hello World')->cookie(
    'name', 'value', $minutes, $path, $domain, $secure, $httpOnly
);
```

生成 Cookie 实例

如果你想要在一段时间以后生成一个可以给定 `Symfony\Component\HttpFoundation\Cookie` 的响应实例，你可以先生成 `$cookie` 实例，然后再指定给 `response` 实例，否则这个 cookie 不会被发送回客户端：

```
$cookie = cookie('name', 'value', $minutes);

return response('Hello World')->cookie($cookie);
```

译者注：关于 Cookie，需要注意一点，默认 Laravel 创建的所有 Cookie 都是加密过的，创建未加密的 Cookie 的方法请见 [【小技巧分享】在 Laravel 中设置没有加密的 cookie](#)

文件资源

获取上传文件

你可以使用 `Illuminate\Http\Request` 实例中的 `file` 方法获取上传的文件。`file` 方法返回的对象是 `Symfony\Component\HttpFoundation\File\UploadedFile` 类的实例，该类继承了 PHP 的 `SplFileInfo` 类，并提供了许多和文件交互的方法：

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

```
$file = $request->photo;
```

你可以使用请求的 `hasFile` 方法确认上传的文件是否存在：

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

确认上传的文件是否有效

除了检查上传的文件是否存在外，你也可以通过 `isValid` 方法验证上传的文件是否有效：

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

文件路径 & 扩展

`UploadedFile` 这个类也包含了访问文件完整路径和扩展的方法。`extension` 方法会尝试根据文件内容猜测文件的扩展名。猜测结果可能不同于客户端原始的扩展名：

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

$extension = $request->photo->extension();
```

其它上传文件的方法

`UploadedFile` 的实例还有许多可用的方法，可以到该对象的 [API 文档](#) 了解这些方法的详细信息。

储存上传文件

在设置好 [文件系统](#) 的配置信息后，你可以使用 `UploadedFile` 的 `store` 方法把上传文件储存到本地磁盘，或者是亚马逊 S3 云存储上。

`store` 方法允许存储文件到相对于文件系统根目录配置的路径。这个路径不能包含文件名，名称将使用 MD5 散列文件内容自动生成。

`store` 方法还接受一个可选的第二个参数，用于文件存储到磁盘的名称。这个方法会返回文件相对于磁盘根目录的路径：

```
$path = $request->photo->store('images');

$path = $request->photo->store('images', 's3');
```

如果你不想自动生成文件名，那么可以使用 `storeAs` 方法去设置路径，文件名和磁盘名作为方法参数：

```
$path = $request->photo->storeAs('images', 'filename.jpg');

$path = $request->photo->storeAs('images', 'filename.jpg', 's3');
```

Laravel 的请求返回 Response

- [创建响应](#)
 - [附加头信息至响应](#)
 - [附加 Cookie 至响应](#)
 - [Cookies 加密](#)
- [重定向](#)
 - [重定向至命名路由](#)
 - [重定向至控制器行为](#)
 - [重定向并附加 Session 闪存数据](#)
- [其他响应类型](#)
 - [视图响应](#)
 - [JSON 响应](#)
 - [文件下载](#)
 - [文件响应](#)
- [响应宏](#)

创建响应

字符串 & 数组

所有路由和控制器都会返回一个响应并返回给用户的浏览器。Laravel 提供了几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串字符串。框架会自动将字符串转换为一个完整的 HTTP 响应：

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

从路由和控制器不仅能返回字符串，也可以返回数组。框架也会自动地将数组转为 JSON 响应：

```
Route::get('/', function () {
    return [1, 2, 3];
});
```

{tip} 你知道吗？ [Eloquent 集合](#) 也可以从路由和控制器中直接返回，它们会自动转为 JSON 响应。试试吧！

响应对象

一般来说，你不需要从路由方法返回简单的字符串或数组。而是需要返回整个 [Illuminate\Http\Response](#) 实例或 [视图](#)。

当返回整个 `Response` 实例时，Laravel 允许自定义响应的 HTTP 状态码和响应头信息。`Response` 实例继承自 `Symfony\Component\HttpFoundation\Response` 类，该类提供了丰富的构建 HTTP 响应的方法：

```
Route::get('home', function () {
    return response('Hello World', 200)
        ->header('Content-Type', 'text/plain');
});
```

附加头信息至响应

大部分的响应方法都是可链式调用的，以使你更酣畅淋漓的创建响应实例。例如，你可以在响应返回给用户前使用 `header` 方法向响应实例中附加一系列的头信息：

```
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` 方法来指定一个包含头信息的数组：

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

附加 Cookie 至响应

通过响应对象的 `cookie` 方法可以让你轻松的附加 cookies 至响应。你可以使用 `cookie` 方法来生成 cookie 并附加至响应实例，像这样：

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

`cookie` 方法也接受另外几个参数，它们的使用频率较低。通常，这些参数和给予原生 PHP 的 `setcookie` 方法的参数有着相同的目的和含义：

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

Cookies 加密

默认情况下，Laravel 生成的所有 cookie 都是加密并通过签名验证的，因此他们并不能够在客户端被修改和读取。如果你想对你的应用程序生成的部分 cookie 禁用加密，可以使用 `App\Http\Middleware\EncryptCookies` 中间件的 `$except` 属性，该文件存储在 `app/Http/Middleware`：

```
/**
 * 无需被加密的 cookie 名
 *
 * @var array
 */
protected $except = [
    'cookie_name',
];
```

重定向

重定向响应是 `Illuminate\Http\RedirectResponse` 类的实例，并且包含用户需要重定向至另一个 URL 所需的头信息。Laravel 提供了许多方法用于生成 `RedirectResponse` 实例。最简单的方法是使用全局的 `redirect` 辅助函数：

```
Route::get('dashboard', function () {
    return redirect('home/dashboard');
});
```

有些情况下你可能希望用户重定向至上级一页面，比如，当提交表单失败时。这时可以使用全局辅助函数 `back`。由于此功能利用了 [Session](#)，请确保调用 `back` 函数的路由是使用 `web` 中间件组或应用了所有的 `Session` 中间件：

```
Route::post('user/profile', function () {
    // 验证请求...

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

重定向至命名路由

当调用不带参数的辅助函数 `redirect` 时，会返回一个 `Illuminate\Routing\Redirector` 实例，该实例允许你调用 `Redirector` 实例的任何方法。例如，生成一个 `RedirectResponse` 重定向至一个被命名的路由时，您可以使用 `route` 方法：

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

如果你的路由有参数，它们可以作为 `route` 方法的第二个参数来传递：

```
// 对于该 URI 的路由: profile/{id}

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

通过 Eloquent 模型填充参数

如果要重定向到一个使用了 Eloquent 模型并需要传递 ID 参数的路由上，你只需传递模型本身即可，ID 会自动提取。

```
// 对于此路由: profile/{id}

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

如果想要更改自动提取的路由参数的键值，你应该重写 Eloquent 模型里的 `getRouteKey` 方法：

```
/**
 * 获取模型的路由键值.
 *
 * @return mixed
 */
public function getRouteKey()
{
    return $this->slug;
}
```

重定向至控制器行为

你可能也会用到生成重定向至 [控制器行为](#) 的响应。要实现此功能，可以向 `action` 方法传递控制器和行为名称作为参数来实现。请记住，这里并不需要指定完整的命名空间，因为 Laravel 的 `RouteServiceProvider` 会自动设置基本的控制器命名空间：

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

如果控制器路由包含参数则需要把他们作为 `action` 函数的第二个参数传递：

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

重定向并附加 Session 闪存数据

重定向至一个新的 URL 的同时通常会 [附加 Session 闪存数据](#)。一般来说，在控制器行为成功地执行之后才会向 Session 中闪存成功的消息。为了方便，你可以利用链式调用的方式创建一个 `RedirectResponse` 的实例并闪存数据至 Session：

```
Route::post('user/profile', function () {
    // 更新用户的信息

    return redirect('dashboard')->with('status', 'Profile updated!');
});
```

用户重定向至指定页面后，你可以从 `Session` 中获取并展示闪存数据。例如，使用 [Blade 语法](#)：

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

其他响应类型

使用全局辅助函数 `response` 可以轻松的生成其他类型的响应实例。当不带任何参数调用 `response` 时，将会返回 `Illuminate\Contracts\Routing\ResponseFactory` [Contract](#) 的实现。Contract 包含许多有用的用来辅助生成响应的方法。

视图响应

如果你的响应内容不但需要控制响应状态码和响应头信息而且还需要返回一个 [视图](#)，这时你应该使用 `view` 方法：

```
return response()
    ->view('hello', $data, 200)
    ->header('Content-Type', $type);
```

当然，如果不需要自定义 HTTP 状态码和响应头信息，则可使用全局的 `view` 辅助函数。

JSON 响应

`json` 方法会自动将 `Content-Type` 响应头信息设置为 `application/json`，并使用 PHP 的 `json_encode` 函数将数组转换为 JSON 字符串。

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

如果想要创建一个 JSONP 响应，则可以使用 `json` 方法并结合 `withCallback` 函数：

```
return response()
```

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

文件下载

`download` 方法可以用于生成强制让用户的浏览器下载指定路径文件的响应。`download` 方法接受文件名称作为方法的第二个参数，此名称为用户下载文件时看见的文件名称。最后，你可以传递一个包含 HTTP 头信息的数组作为第三个参数传入该方法：

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

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

{note} 管理文件下载的扩展包 Symfony HttpFoundation，要求下载文件名必须是 ASCII 编码。

文件响应

`file` 方法可以用来显示一个文件，例如图片或者 PDF，直接在用户的浏览器中显示，而不是开始下载。这个方法的第一个参数是文件的路径，第二个参数是包含头信息的数组：

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

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

响应宏

如果你想要自定义可以在很多路由和控制器重复使用的响应，可以使用 `Response Facade` 实现的 `macro` 方法。举个例子，来自 [服务提供者](#) 的 `boot` 方法：

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\Response;

class ResponseMacroServiceProvider extends ServiceProvider
{
    /**
     * 注册应用的响应宏
     *
     * @return void
     */
    public function boot()
    {
        Response::macro('caps', function ($value) {
```

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

macro 函数第一个参数为宏名称，第二个参数为闭包函数。宏的闭包函数会在 ResponseFactory 的实现或者辅助函数 response 调用宏名称的时候运行：

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

Laravel 的视图功能

- [创建视图](#)
- [传递数据到视图](#)
 - [共享数据给所有视图](#)
- [视图合成器](#)

创建视图

视图的用途是用来存放应用程序中 HTML 内容，并且能够将你的控制器层（或应用逻辑层）与展现层分开。视图文件目录为 `resources/views`，示例视图如下：

```
<!-- 此视图文件位置: resources/views/greeting.blade.php -->

<html>
    <body>
        <h1>Hello, {{ $name }}</h1>
    </body>
</html>
```

上述视图文件位置为 `resources/views/greeting.blade.php`，我们可以通过全局函数 `view` 来使用这个视图，如下：

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

如你所见，`view` 函数中，第一个参数是 `resources/views` 目录中视图文件的文件名，第二个参数是一个数组，数组中的数据可以直接在视图文件中使用。在上面示例中，我们将 `name` 变量传递到了视图中，并在视图中使用 [Blade 模板语言](#) 打印出来。

当然，视图文件也可能存放在 `resources/views` 的子目录中，你可以使用英文句点 `.` 来引用深层子目录中的视图文件。例如，一个视图的位置为 `resources/views/admin/profile.blade.php`，使用示例如下：

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

判断视图文件是否存在

如果需要判断一个视图文件是否存在，你可以使用 `View Facade` 上的 `exists` 方法来判定，如果视图文件存在，则返回值为 `true`：

```
use Illuminate\Support\Facades\View;
```

```
if (View::exists('emails.customer')) {
    //
}
```

传递数据到视图

如上述例子中，你可以使用数组将数据传递到视图文件：

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

当使用上面方式传递数据时，第二个参数（`$data`）必须是键值对数组（关联数组）。在视图文件中，你可以通过对应的关键字（`$key`）取用相应的数据值，例如`<?php echo $key; ?>`。如果只需要传递特定数据而非一个臃肿的数组到视图文件，可以使用`with`辅助函数，示例如下：

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

把数据共享给所有视图

有时候可能需要共享特定的数据给应用程序中所有的视图，那这时候你需要`view Facade`的`share`方法。通常需要将所有`share`方法的调用代码放到[服务提供者](#)的`boot`方法中，此时你可以选择使用[AppServiceProvider](#)或创建独立的[服务提供者](#)。示例代码如下：

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        View::share('key', 'value');
    }

    /**
     * Register the service provider.
     *
     * @return void
     */
}
```

```
public function register()
{
    //
}
```

视图合成器

视图合成器是在视图渲染时调用的一些回调或者类方法。如果你需要在某些视图渲染时绑定一些数据上去，那么视图合成器就是你的不二之选，另外他还可以帮你将这些绑定逻辑整理到特定的位置。

下面例子中，我们会在一个 [服务提供者](#) 中注册一些视图合成器。同时使用 `View Facade` 来访问 `Illuminate\Contracts\View\Factory` contract 的底层实现。注意：Laravel 没有存放视图合成器的默认目录，但你可以根据自己的喜好来重新组织，例如：`App\Http\ViewComposers`。

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;
use Illuminate\Support\ServiceProvider;

class ComposerServiceProvider extends ServiceProvider
{
    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function boot()
    {
        // Using class based composers...
        View::composer(
            'profile', 'App\Http\ViewComposers\ProfileComposer'
        );

        // Using Closure based composers...
        View::composer('dashboard', function ($view) {
            //
        });
    }

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
```

```

        //
    }
}

```

{note} 注意，如果你创建了新的一个 [服务提供者](#) 来存放你视图合成器的注册项，那么你需要将这个 [服务提供者](#) 添加到配置文件 `config/app.php` 的 `providers` 数组中。

到此我们已经注册了上面的视图合成器，效果是每次 `profile` 视图渲染时，都会执行 `ProfileComposer@compose` 方法。那么下面我们就来定义这个合成器类吧。

```

<?php

namespace App\Http\ViewComposers;

use Illuminate\View\View;
use App\Repositories\UserRepository;

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

    /**
     * Create a new profile composer.
     *
     * @param UserRepository $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        // Dependencies automatically resolved by service container...
        $this->users = $users;
    }

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

```

每当视图渲染时，该合成器的 `compose` 方法都会被调用，并且传入一个 `Illuminate\View\View` 实例作为参数，在这个过程中，你可以使用 `with` 方法绑定数据到目标视图。

{tip} 所有的视图合成器都会由 [服务容器](#) 来解析，所以你可以在合成器的构造函数中使用类型提示来注入需要的任何依赖。

将视图合成器附加到多个视图

通过将目标视图文件数组作为第一个参数传入 `composer` 方法，你可以把一个视图合成器同时附加到多个视图。

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

`composer` 方法同时也接受通配符 `*`，可以让你将一个视图合成器一次性绑定到所有的视图：

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

视图构造器

视图 构造器 和视图合成器非常相似。不同之处在于：视图构造器在视图实例化时执行，而视图合成器在视图渲染时执行。如下，可以使用 `creator` 方法来注册一个视图塑造器：

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

Laravel 的 HTTP 会话机制

- 简介
 - 配置
 - 驱动条件
- 使用 Session
 - 获取 Session 数据
 - 存储 Session 数据
 - 闪存数据到 Session
 - 删除 Session 数据
 - 重新生成 Session ID
- 添加自定义 Session 驱动
 - 实现驱动
 - 注册驱动

简介

由于 HTTP 是无状态的，Session 提供了一种在多个请求之间存储有关用户信息的方法。Laravel 附带支持了多种 Session 后端驱动，它们都可以通过语义化统一的 API 访问。Laravel 本身支持比较热门的 Session 后端驱动，如 [Memcached](#)、[Redis](#) 和数据库。

配置

Session 相关的配置文件存储在 `config/session.php`。请务必查看此文件中对于你可用的选项。默认设置下，Laravel 的配置是使用文件作为 Session 驱动，大多数情况下能够运行良好。在生产环境下，你可以考虑使用 `memcached` 或 `redis` 驱动来达到更出色的性能表现。

Session 配置的 `driver` 的选项定义了每次请求的 Session 数据的存储位置。Laravel 附带了几个不错且可开箱即用的驱动：

- `'file'` - 将 Session 保存在 `'storage/framework/sessions'`。
- `'cookie'` - Session 保存在安全加密的 Cookie 中。
- `'database'` - Session 保存在关系型数据库。
- `'memcached'` / `'redis'` - 将 Sessions 保存在其中一个快速且基于缓存的存储系统中。
- `'array'` - 将 Sessions 保存在简单的 PHP 数组中，并只存在于本次请求。

{tip} 数组驱动一般用于 [测试](#) 防止存储在 Session 的数据被持久化。

驱动条件

数据库

使用 [数据库](#) 作为 Session 驱动时，你需要创建一张包含 Session 各项数据的表。以下例子是使用 `Schema` 建表：

```
Schema::create('sessions', function ($table) {
    $table->string('id')->unique();
    $table->integer('user_id')->nullable();
    $table->string('ip_address', 45)->nullable();
    $table->text('user_agent')->nullable();
    $table->text('payload');
    $table->integer('last_activity');
});
```

也可以使用 Artisan 的 `session:table` 命令生成一个迁移文件：

```
php artisan session:table

php artisan migrate
```

Redis

在使用 Redis 作为 Session 驱动之前，你需要通过 Composer 安装 `predis/predis` 扩展包(~1.0)。你还需要在 `database` 配置文件中指定 Redis 连接参数信息。在 Session 配置文件中的 `connection` 选项中指定 Session 使用的 Redis 连接。

使用 Session

获取 Session 数据

Laravel 中有两种主要的方式使用 Session 数据的方式：一种是全局的辅助函数 `session`，另一种是通过 HTTP 请求实例。首先，我们先看一下第二种方法，就是通过具有控制器方法类型提示的 HTTP 请求实例来访问 Session。请记住，控制器方法的依赖关系会通过 Laravel 的 [服务容器](#)自动注入：

```
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller
{
    /**
     * 展示用户个人信息
     *
     * @param Request $request
     * @param int $id
     * @return Response
     */
    public function show(Request $request, $id)
```

```

{
    $value = $request->session()->get('key');

    //
}

}

```

当从 Session 获取值时，你也可以传递一个默认值作为 `get` 方法的第二个参数。如果 Session 中并不存在指定的键值便会返回传入的默认值。若传递一个闭包作为 `get` 方法的默认值且请求的键值并不存在时，此时 `get` 方法会返回这个闭包函数运行后的返回值：

```

$value = $request->session()->get('key', 'default');

$value = $request->session()->get('key', function () {
    return 'default';
});

```

全局 Session 辅助函数

你也可以使用全局的 PHP 函数 `session` 来获取和存储 Session 数据。使用单个字符串类型的值作为参数调用 `session` 函数时，它将返回字该字符串参数对应的 Session 键值。当使用一个 key / value 键值对数组作为参数调用 `session` 函数时，传入的键值将会存入 Session：

```

Route::get('home', function () {
    // 获取 Session 中的一条数据...
    $value = session('key');

    // 指定一个默认值...
    $value = session('key', 'default');

    // 存储一条数据至 Session 中...
    session(['key' => 'value']);
});

```

{tip} HTTP 请求实例与 `Session` 全局辅助函数使用 Session 并没有实质上的区别。两种方法都是可以通过 `assertSessionHas` 方法 测试，`assertSessionHas` 方法在所有的测试用例都是可用的。关于测试的更多信息，请阅读文档 测试

获取所有 Session 数据

如果你想要获取所有的 Session 数据，可以使用 `all` 方法：

```
$data = $request->session()->all();
```

判断某个 Session 值是否存在

使用 `has` 方法检查某个值是否存在于 Session 内，如果该值存在并且不为 `null`，那么则返回 `true`:

```
if ($request->session()->has('users')) {
    //
}
```

在判断值是否在 Session 中是否存在时，如果该值可能为 `null`，你需要使用 `exists` 方法，如果该值存在，那么则返回 `true`:

```
if ($request->session()->exists('users')) {
    //
}
```

存储 Session 数据

存储数据到 Session，你可用使用 `put` 方法，或者 `session` 辅助函数。

```
// 通过 HTTP 请求实例...
$request->session()->put('key', 'value');

// 通过全局辅助函数
session(['key' => 'value']);
```

保存数据进 Session 数组值中

`push` 方法可以将一个新的值加入至一个 Session 数组内。例如，假设 `user.teams` 这个键是包含团队名称的数组，你可以将一个新的值加入此数组中。比如这样:

```
$request->session()->push('user.teams', 'developers');
```

从 Session 中取出并删除数据

`pull` 方法将把数据从 Session 内取出，并且删除:

```
$value = $request->session()->pull('key', 'default');
```

闪存数据到 Session

有时候你想存入一条缓存的数据，让它只在下一次的请求内有效，则可以使用 `flash` 方法。使用这个方法保存 session，只能将数据保留到下个 HTTP 请求，然后就会被自动删除。闪存数据在短期的状态消息中很有用:

```
$request->session()->flash('status', 'Task was successful!');
```

如果需要保留闪存数据给更多请求，可以使用 `reflash` 方法，这将会将所有的闪存数据保留在额外的请求。如果想保留特定的闪存数据，则可以使用 `keep` 方法：

```
$request->session()->reflash();

$request->session()->keep(['username', 'email']);
```

删除 Session 数据

`forget` 方法可以从 Session 内删除一条数据。如果你想删除 Session 内所有数据，则可以使用 `flush` 方法：

```
$request->session()->forget('key');

$request->session()->flush();
```

重新生成 Session ID

重新生成 Session ID，通常时为了防止恶意用户利用 [session fixation](#) 对应用进行攻击。

如果你使用了内置函数 `LoginController`，那么 Laravel 会自动重新生成 Session ID，否则，你需要手动使用 `regenerate` 方法重新生成 Session ID

```
$request->session()->regenerate();
```

添加自定义 Session 驱动

实现驱动

你自定义的 Session 驱动必须实现 `SessionHandlerInterface` 接口。这个接口包含了一些基本需要实现的方法。一个基本的 MongoDB 实现应该看起来像这样：

```
<?php

namespace App\Extensions;

class MongoHandler implements SessionHandlerInterface
{
    public function open($savePath, $sessionId) {}
    public function close() {}
    public function read($sessionId) {}
    public function write($sessionId, $data) {}
    public function destroy($sessionId) {}
    public function gc($lifetime) {}

}
```

{tip} Laravel 默认没有附带扩展目录，你可以把它放在你喜欢的目录内。在下面这个例子中，我们创建了一个 `Extensions` 目录放置自定义的 `MongoHandler` 扩展。

接口中的这些方法不太容易理解。让我们来快速了解每个方法的作用：

- `'open'` 方法通常用于基于文件的 Session 存储系统。因为 Laravel 已经附带了一个 `'file'` 的驱动，所以在该方法中不需要放置任何代码。PHP 要求必需要有这个方法的实现，但你可以把这方法置空也没关系。 - `'close'` 方法跟 `'open'` 方法很相似，通常也可以被忽略。对大多数的驱动而言，此方法并不是需要的。 - `'read'` 方法应当返回与给定的 `'$sessionId'` 相匹配的 Session 数据的字符串版本。从这个自定义的驱动中获取或存储 Session 数据不需要做任何序列化或其它编码，因为 Laravel 已经为我们做了序列化。 - `'write'` 将与 `'$sessionId'` 关联的特定 `'$data'` 字符串，写入到持久化存储系统，如 MongoDB、Dynamo 等等。再次重申，你不需要做任何序列化或其它编码，因为 Laravel 会自动处理这些事情。 - `'destroy'` 方法从持久化存储中移除 `'$sessionId'` 对应的数据。 - `'gc'` 方法能销毁 `'$lifetime'` 之前的所有数据，`'$lifetime'` 是一个 UNIX 的时间戳。对本身拥有过期机制的系统如 Memcached 和 Redis 而言，该方法可以留空。

注册驱动

在 Session 驱动实现了 `SessionHandlerInterface` 接口后，你还需要在框架中注册该驱动，将该扩展驱动添加到 Laravel Session 后端。你可以使用 `Session Facade` 的 `extend` 方法。在 [服务提供者](#) 的 `boot` 方法内调用 `extend` 方法。你可用使用已经存在的 `AppServiceProvider` 或者创建一个新的提供者。

```
<?php

namespace App\Providers;

use App\Extensions\MongoSessionStore;
use Illuminate\Support\Facades\Session;
use Illuminate\Support\ServiceProvider;

class SessionServiceProvider extends ServiceProvider
{
    /**
     * 提供注册后运行的服务。
     *
     * @return void
     */
    public function boot()
    {
        Session::extend('mongo', function ($app) {
            // Return implementation of SessionHandlerInterface...
            return new MongoSessionStore();
        });
    }

    /**
     * 在容器中注册绑定。
     *

```

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

一旦 Session 驱动被注册，则必须在 config/session.php 的配置文件内使用 Mongo 驱动。

Laravel 的表单验证机制详解

- 简介
- 快速上手
 - 定义路由
 - 创建控制器
 - 编写验证逻辑
 - 显示验证错误
 - 有关可选字段的注意事项
- 表单请求验证
 - 创建表单请求
 - 授权表单请求
 - 自定义错误格式
 - 自定义错误消息
- 手动创建表单验证
 - 自动重定向
 - 命名错误包
 - 验证后钩子
- 处理错误消息
 - 自定义错误消息
- 可用的验证规则
- 按条件增加规则
- 验证数组
- 自定义验证规则

Introduction

Laravel 提供了多种不同的验证方法来对应用程序传入的数据进行验证。默认情况下，Laravel 的基类控制器使用 `ValidatesRequests Trait`，它提供了方便的方法使用各种强大的验证规则来验证传入的 HTTP 请求数据。

快速上手

为了了解 Laravel 强大验证特性，我们先来看看一个完整的表单验证并返回错误消息的示例。

定义路由

首先，我们假定在 `routes/web.php` 文件中定义了以下路由：

```
Route::get('post/create', 'PostController@create');

Route::post('post', 'PostController@store');
```

`GET` 路由会显示一个用于创建新博客文章的表单，`POST` 路由则会将新的博客文章保存到数据库。

创建控制器

下一步，我们来看一个处理这些路由的简单的控制器。我们将 `store` 方法置空：

```
<?php

namespace App\Http\Controllers;

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

class PostController extends Controller
{
    /**
     * 显示创建博客文章的表单。
     *
     * @return Response
     */
    public function create()
    {
        return view('post.create');
    }

    /**
     * 保存一个新的博客文章。
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        // 验证以及保存博客发表文章...
    }
}
```

编写验证逻辑

现在我们准备开始编写 `store` 逻辑方法来验证我们博客发布的新文章。检查应用程序的基底控制器 (`App\Http\Controllers\Controller`) 类你会看到这个类使用了 `ValidatesRequests Trait`。这个 Trait 在你所有的控制器里提供了方便的 `validate` 验证方法。

`validate` 方法会接收 HTTP 传入的请求以及验证的规则。如果验证通过，你的代码就可以正常的运行。若验证失败，则会抛出异常错误消息并自动将其返回给用户。在一般的 HTTP 请求下，都会生成一个重定向响应，而对于 AJAX 请求则会发送 JSON 响应。

让我们接着回到 `store` 方法来深入理解 `validate` 方法：

```

/**
 * 保存一篇新的博客文章。
 *
 * @param Request $request
 * @return Response
 */
public function store(Request $request)
{
    $this->validate($request, [
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ]);

    // 文章内容是符合规则的，存入数据库
}

```

如你所见，我们将本次 HTTP 请求及所需的验证规则传递至 `validate` 方法中。另外再提醒一次，如果验证失败，将会自动生成一个对应的响应。如果验证通过，那我们的控制器将会继续正常运行。

在第一次验证失败后停止

有时，你希望在某个属性第一次验证失败后停止运行验证规则。为了达到这个目的，附加 `bail` 规则到该属性：

```

$this->validate($request, [
    'title' => 'bail|required|unique:posts|max:255',
    'body' => 'required',
]);

```

在这个例子里，如果 `title` 字段没有通过 `required` 的验证规则，那么 `unique` 这个规则将不会被检测了。将按规则被分配的顺序来验证规则。

嵌套属性的注解

如果你的 HTTP 请求包含一个「嵌套的」参数，你可以在验证规则中通过「点」语法来指定这些参数。

```

$this->validate($request, [
    'title' => 'required|unique:posts|max:255',
    'author.name' => 'required',
    'author.description' => 'required',
]);

```

显示验证错误

如果本次请求的参数未通过我们指定的验证规则呢？正如前面所提到的，Laravel 会自动把用户重定向到先前的位置。另外，所有的验证错误会被自动 [闪存至 session](#)。

再者，请注意在 `GET` 路由中，我们无需显式的将错误信息和视图绑定起来。这是因为 Laravel 会检查在 `Session` 数据中的错误信息，然后如果对应的视图存在的话，自动将它们绑定起来。变量 `$errors` 会成为 `Illuminate\Support\MessageBag` 的一个实例对象。要获取关于这个对象的更多信息，请[查阅这个文档](#)。

{tip} `$errors` 变量被 `Illuminate\View\Middleware\ShareErrorsFromSession` 中间件绑定到视图，该中间件由 `web` 中间件组提供。当这个中间件被应用后，在你的视图中就可以获取到 `$error` 变量，可以使你方便的假定 `$errors` 变量总是已经被定义好并且可以安全的使用。

所以，在我们的例子中，当验证失败的时候，用户将会被重定向到 `create` 方法，让我们在视图中显示错误信息：

```
<!-- /resources/views/post/create.blade.php -->

<h1>创建文章</h1>

@if (count($errors) > 0)
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

<!-- 创建文章表单 -->
```

有关可选字段的注意事项

默认情况下，Laravel 会在你的应用中的全局中间件栈中包含 `TrimStrings` 和 `ConvertEmptyStringsToNull` 中间件。这些中间件在 `App\Http\Kernel` 类中。因此，如果您不希望验证程序将「null」值视为无效的，您通常需要将「可选」的请求字段标记为 `nullable`。

```
$this->validate($request, [
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
    'publish_at' => 'nullable|date',
]);
```

在这个例子里，我们指定 `publish_at` 字段可以为 `null` 或者一个有效的日期格式。如果 `nullable` 的修饰词没有添加到规则定义中，验证器会认为 `null` 是一个无效的日期格式。

自定义闪存的错误消息格式

当验证失败时，如果你想要在闪存上自定义验证的错误格式，则需在控制器中重写 `formatValidationErrors`。别忘了将 `Illuminate\Contracts\Validation\Validator` 类引入到文件上方：

```
<?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 $validator)
    {
        return $validator->errors()->all();
    }
}
```

AJAX 请求验证

在这个例子中，我们使用一种传统的方式来将数据发送到应用程序上。当我们在 AJAX 的请求中使用 `validate` 方法时，Laravel 并不会生成一个重定向响应，而是会生成一个包含所有错误验证的 JSON 响应。这个 JSON 响应会发送一个 422 HTTP 状态码。

表单请求验证

创建表单请求

在更复杂的验证情境中，你可能会想要创建一个「表单请求（form request）」。表单请求是一个自定义的请求类，里面包含着验证逻辑。要创建一个表单请求类，可使用 Artisan 命令行命令

`make:request` :

```
php artisan make:request StoreBlogPost
```

新生成的类保存在 `app/Http/Requests` 目录下。如果这个目录不存在，那么将会在你运行 `make:request` 命令时创建出来。让我们添加一些验证规则到 `rules` 方法中：

```
/**
```

```

 * 获取适用于请求的验证规则。
 *
 * @return array
 */
public function rules()
{
    return [
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ];
}

```

怎样才能较好的运行验证规则呢？你所需要做的就是在控制器方法中利用类型提示传入请求。传入的请求会在控制器方法被调用前进行验证，意思就是说你不会因为验证逻辑而把控制器弄得一团糟：

```

/**
 * 保存传入的博客文章。
 *
 * @param StoreBlogPost $request
 * @return Response
 */
public function store(StoreBlogPost $request)
{
    // The incoming request is valid...
}

```

如果验证失败，就会生成一个重定向响应把用户返回到先前的位置。这些错误会被闪存到 Session，所以这些错误都可以被显示。如果进来的是 AJAX 请求的话，则会传回一个 HTTP 响应，其中包含了 422 状态码和验证错误的 JSON 数据。

添加表单请求后钩子

如果你想在表单请求「之后」添加钩子，你可以使用 `withValidator` 方法。这个方法接收一个完整的验证类，允许你在实际判断验证规则调之前调用验证类的所有方法：

```

/**
 * @param \Illuminate\Validation\Validator $validator
 * @return void
 */
public function withValidator($validator)
{
    $validator->after(function ($validator) {
        if ($this->somethingElseIsInvalid()) {
            $validator->errors()->add('field', 'Something is wrong with this field!');
        }
    });
}

```

授权表单请求

表单的请求类内包含了 `authorize` 方法。在这个方法中，你可以确认用户是否真的通过了授权，以便更新指定数据。比方说，有一个用户想试图去更新一篇文章的评论，你能保证他确实是这篇评论的拥有者吗？具体代码如下：

```
/**
 * 判断用户是否有权限做出此请求。
 *
 * @return bool
 */
public function authorize()
{
    $comment = Comment::find($this->route('comment'));

    return $comment && $this->user()->can('update', $comment);
}
```

由于所有的表单请求都是扩展于基础的 Laravel 请求类，所以我们可以使用 `user` 方法去获取当前认证登录的用户。同时请注意上述例子中对 `route` 方法的调用。这个方法授权你获取调用的路由规则中的 URI 参数，譬如下面例子中的 `{comment}` 参数：

```
Route::post('comment/{comment}');
```

如果 `authorize` 方法返回 `false`，则会自动返回一个 HTTP 响应，其中包含 403 状态码，而你的控制器方法也将不会被运行。

如果你打算在应用程序的其它部分处理授权逻辑，只需从 `authorize` 方法返回 `true`：

```
/**
 * 判断用户是否有权限做出此请求。
 *
 * @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闪存到 Session 的验证错误格式，可在你的基底请求 (`App\Http\Requests\Request`) 中重写 `formatErrors`。别忘了文件上方引入 `Illuminate\Contracts\Validation\Validator` 类：

```
/**
```

```

 * {@inheritDoc}
 */
protected function formatErrors(Validator $validator)
{
    return $validator->errors()->all();
}

```

自定义错误消息

你可以通过重写表单请求的 `messages` 方法来自定义错误消息。此方法必须返回一个数组，其中含有成对的属性或规则以及对应的错误消息：

```

/**
 * 获取已定义验证规则的错误消息。
 *
 * @return array
 */
public function messages()
{
    return [
        'title.required' => 'A title is required',
        'body.required'  => 'A message is required',
    ];
}

```

手动创建验证请求

如果你不想要使用 `ValidatesRequests` Trait 的 `validate` 方法，你可以手动创建一个 `validator` 实例并通过 `validator::make` 方法在 [Facade](#) 生成一个新的 `validator` 实例：

```

<?php

namespace App\Http\Controllers;

use Validator;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PostController extends Controller
{
    /**
     * 保存一篇新的博客文章。
     *
     * @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();
    }

    // 保存文章
}
}

```

第一个传给 `make` 方法的参数是验证数据。第二个参数则是数据的验证规则。

如果请求没有通过验证，则可以使用 `withErrors` 方法把错误消息闪存到 Session。在进行重定向之后，`$errors` 变量可以在视图中自动共用，让你可以轻松地显示这些消息并返回给用户。`withErrors` 方法接收 validator、MessageBag，或 PHP array。

自动重定向

如果你想手动创建一个验证器实例，但希望继续享用 `ValidatesRequest` 特性提供的自动跳转功能，那么你可以调用一个现存的验证器实例中的 `validate` 方法。如果验证失败了，用户会被自动重定向，或者在 AJAX 请求中，一个 JSON 格式的响应将会被返回：

```

Validator::make($request->all(), [
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
])->validate();

```

命名错误包

如果你在一个页面中有多个表单，你也许会希望命名错误信息包 `MessageBag`，错误信息包允许你从指定的表单中接收错误信息。简单的给 `withErrors` 方法传递第二个参数作为一个名字：

```

return redirect('register')
    ->withErrors($validator, 'login');

```

然后你能从 `$errors` 变量中获取到 `MessageBag` 实例：

```

{{ $errors->login->first('email') }}

```

验证后钩子

验证器允许你在验证完成之后附加回调函数。这使得你可以容易的执行进一步验证，甚至可以在消息集合中添加更多的错误信息。使用它只需在验证实例中使用 `after` 方法：

```
$validator = Validator::make(...);

$validator->after(function ($validator) {
    if ($this->somethingElseIsInvalid()) {
        $validator->errors()->add('field', 'Something is wrong with this field!');
    }
});

if ($validator->fails()) {
    //
}
```

处理错误消息

调用 `Validator` 实例的 `errors` 方法，会得到一个 `Illuminate\Support\MessageBag` 的实例，里面有许多可让你操作错误消息的便利方法。`$errors` 值可以自动的被所有的视图获取，并且是一个 `MessageBag` 类的实例。自动对所有视图可用的 `$errors` 变量也是 `MessageBag` 类的一个实例。

查看特定字段的第一个错误消息

如果要查看特定字段的第一个错误消息，可以使用 `first` 方法：

```
$errors = $validator->errors();

echo $errors->first('email');
```

查看特定字段的所有错误消息

如果你想通过指定字段来简单的获取所有消息中的一个数组，则可以使用 `get` 方法：

```
foreach ($errors->get('email') as $message) {
    //
}
```

如果你正在验证的一个表单字段类型是数组，你可以使用 `*` 来获取每个元素的所有错误信息：

```
foreach ($errors->get('attachments.*') as $message) {
    //
}
```

查看所有字段的所有错误消息

如果你想要得到所有字段的消息数组，则可以使用 `all` 方法：

```
foreach ($errors->all() as $message) {
    //
}
```

判断特定字段是否含有错误消息

可以使用 `has` 方法来检测一个给定的字段是否存在错误信息：

```
if ($errors->has('email')) {
    //
}
```

自定义错误消息

如果有需要的话，你也可以自定义错误的验证消息来取代默认的验证消息。有几种方法可以指定自定义消息。首先，你需要先通过传递三个参数到 `Validator::make` 方法来自定义验证消息：

```
$messages = [
    'required' => 'The :attribute field is required.',
];

$validator = Validator::make($input, $rules, $messages);
```

在这个例子中，`:attribute` 占位符会被通过验证的字段实际名称所取代。除此之外，你还可以使用其它默认字段的验证提示消息。例如：

```
$messages = [
    'same'      => 'The :attribute and :other must match.',
    'size'      => 'The :attribute must be exactly :size.',
    'between'   => 'The :attribute must be between :min - :max.',
    'in'        => 'The :attribute must be one of the following types: :values',
];
```

指定自定义消息到特定的属性

有时候你可能想要对特定的字段来自定义错误消息。只需在属性名称后加上「.」符号和指定验证的规则即可：

```
$messages = [
    'email.required' => 'We need to know your e-mail address!',
];
```

在语言文件中指定自定义的消息提示

多数情况下，你会在语言文件中指定自定义的消息提示，而不是将定制的消息传递给 `validator`。实现它需要在语言文件 `resources/lang/xx/validation.php` 中，将定制的消息添加到 `custom` 数组。

```
'custom' => [
    'email' => [
        'required' => 'We need to know your e-mail address!',
    ],
],
```

在语言文件中自定义属性

如果希望将验证消息的 `:attribute` 部分替换为自定义属性名称，则可以在 `resources/lang/xx/validation.php` 语言文件的 `attributes` 数组中指定自定义名称：

```
'attributes' => [
    'email' => 'email address',
],
```

可用的验证规则

以下是所有可用的验证规则清单与功能：

Accepted Active URL After (Date) After Or Equal (Date) Alpha Alpha Dash Alpha Numeric Array
Before (Date) Before Or Equal (Date) Between Boolean Confirmed Date Date Format Different Digits
Digits Between Dimensions (Image Files) Distinct E-Mail Exists (Database) File Filled Image (File) In
In Array Integer IP Address JSON Max MIME Types MIME Type By File Extension Min Nullable Not
In Numeric Present Regular Expression Required Required If Required Unless Required With
Required With All Required Without Required Without All Same Size String Timezone Unique
(Database) URL

`accepted`

验证字段值是否为 `yes`、`on`、`1`、或 `true`。这在确认「服务条款」是否同意时相当有用。

`active_url`

根据 PHP 函数 `dns_get_record`，判断要验证的字段必须具有有效的 A 或 AAAA 记录。

`after:date`

验证字段是否是在指定日期之后。这个日期将会通过 `strtotime` 函数来验证。

```
'start_date' => 'required|date|after:tomorrow'
```

作为替换 `strtotime` 传递的日期字符串，你可以指定其它的字段来比较日期：

```
'finish_date' => 'required|date|after:start_date'
```

after_or_equal:date

验证字段必需是等于指定日期或在指定日期之后。更多信息请参见 [after](#) 规则。

alpha

验证字段值是否仅包含字母字符。

alpha_dash

验证字段值是否仅包含字母、数字、破折号（-）以及下划线（_）。

alpha_num

验证字段值是否仅包含字母、数字。

array

验证字段必须是一个 PHP 数组。

before:date

验证字段是否是在指定日期之前。这个日期将会通过 `strtotime` 函数来验证。

before_or_equal:date

验证字段是否是在指定日期之前。这个日期将会使用 PHP `strtotime` 函数来验证。

between:min,max

验证字段值的大小是否介于指定的 `min` 和 `max` 之间。字符串、数值或是文件大小的计算方式和 [size](#) 规则相同。

boolean

验证字段值是否能够转换为布尔值。可接受的参数为 `true`、`false`、`1`、`0`、`"1"` 以及 `"0"`。

confirmed

验证字段值必须和 `foo_confirmation` 的字段值一致。例如，如果要验证的字段是 `password`，就必须和输入数据里的 `password_confirmation` 的值保持一致。

date

验证字段值是否为有效日期，会根据 PHP 的 `strtotime` 函数来做验证。

`dateformat:_format`

验证字段值是否符合指定的日期格式 (*format*)。你应该只使用 `date` 或 `date_format` 当中的其中一个用于验证，而不应该同时使用两者。

`different:field`

验证字段值是否和指定的字段 (*field*) 有所不同。

`digits:value`

验证字段值是否为 *numeric* 且长度为 *value*。

`digitsbetween:_min,max`

验证字段值的长度是否在 *min* 和 *max* 之间。

`dimensions`

验证的文件必须是图片并且图片比例必须符合规则：

```
'avatar' => 'dimensions:min_width=100,min_height=200'
```

可用的规则为： *min_width*, *max_width*, *min_height*, *max_height*, *width*, *height*, *ratio*。

比例应该使用宽度除以高度的方式出现。能够使用 `3/2` 这样的形式设置，也可以使用 `1.5` 这样的浮点方式：

```
'avatar' => 'dimensions:ratio=3/2'
```

Since this rule requires several arguments, you may use the method to fluently construct the rule:

由于此规则需要多个参数，因此您可以 `Rule::dimensions` 方法来构造规则：

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'avatar' => [
        'required',
        Rule::dimensions()->maxWidth(1000)->maxLength(500)->ratio(3 / 2),
    ],
]);
```

`distinct`

当你在验证数组的时候，你可以指定某个值必须是唯一的：

```
'foo.*.id' => 'distinct'
```

email

验证字段值是否符合 e-mail 格式。

exists:table,column

验证字段值是否存在指定的数据表中。

Exists 规则的基本使用方法

```
'state' => 'exists:states'
```

指定一个特定的字段名称

```
'state' => 'exists:states,abbreviation'
```

有时，您可能需要指定要用于 `exists` 查询的特定数据库连接。你可以使用点「.」语法将数据库连接名称添加到数据表前面来实现这个目的：

```
'email' => 'exists:connection.staff,email'
```

如果您想自定义由验证规则执行的查询，您可以使用 `Rule` 类流畅地定义规则。在这个例子中，我们还将使用数组指定验证规则，而不是使用 `|` 字符来分隔它们：

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'email' => [
        'required',
        Rule::exists('staff')->where(function ($query) {
            $query->where('account_id', 1);
        }),
    ],
]);
```

file

必须是成功上传的文件。

filled

验证的字段必须带有内容。

image

验证字段文件必须为图片格式（jpeg、png、bmp、gif、或svg）。

in:*foo,bar,...*

验证字段值是否有在指定的列表里面。因为这个规则通常需要你 `implode` 一个数组，`Rule::in` 方法可以用来流利地构造规则：

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'zones' => [
        'required',
        Rule::in(['first-zone', 'second-zone']),
    ],
]);
```

inarray:_*anotherfield*

验证的字段必须存在于 *anotherfield* 的值中。

integer

验证字段值是否是整数。

ip

验证字段值是否符合 IP address 的格式。

ipv4

验证字段值是否符合 IPv4 的格式。

ipv6

验证字段值是否符合 IPv6 的格式。

json

验证字段是否是一个有效的 JSON 字符串。

max:*value*

字段值必须小于或等于 *value*。字符串、数值或是文件大小的计算方式和 `size` 规则相同。

mimetypes:*text/plain,...*

验证的文件必须是这些 MIME 类型中的一个：

```
'video' => 'mimetypes:video/avi,video/mpeg,video/quicktime'
```

要确定上传文件的MIME类型，会读取文件的内容，并且框架将尝试猜测 MIME 类型，这可能与客户端提供的 MIME 类型不同。

mimes:*foo,bar,...*

验证字段文件的 MIME 类型是否符合列表中指定的格式。

MIME 规则基本用法

```
'photo' => 'mimes:jpeg,bmp,png'
```

即使你可能只需要验证指定扩展名，但此规则实际上会验证文件的 MIME 类型，其通过读取文件的内容以猜测它的 MIME 类型。

完整的 MIME 类型及对应的扩展名列表可以在下方链接找到：<https://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types>

min:*value*

字段值必须大于或等于 *value*。字符串、数值或是文件大小的计算方式和 **size** 规则相同。

nullable

验证的字段可以为 `null`。这在验证基本数据类型，如字符串和整型这些能包含 `null` 值的数据类型中特别有用。

notin:*_foo,bar,...*

验证字段值是否不在指定的列表里。

numeric

验证字段值是否为数值。

present

验证的字段必须出现，并且数据可以为空。

regex:*pattern*

验证字段值是否符合指定的正则表达式。

Note: 当使用 **regex** 规则时，你必须使用数组，而不是使用管道分隔规则，特别是当正则表达式含有管道符号时。

required

验证字段必须存在输入数据，且不为空。字段符合下方任一条件时即为「空」：

- 该值为 `null`。
- 该值为空字符串。
- 该值为空数组或空的‘可数’对象。
- 该值为没有路径的上传文件。

`requiredif:_anotherfield,value,...`

如果指定的其它字段（`anotherfield`）等于任何一个 `value` 时，此字段为必填。

`requiredunless:_anotherfield,value,...`

如果指定的其它字段（`anotherfield`）等于任何一个 `value` 时，此字段为不必填。

`requiredwith:_foo,bar,...`

如果指定的字段中的任意一个有值且不为空，则此字段为必填。

`requiredwith_all:_foo,bar,...`

如果指定的所有字段都有值，则此字段为必填。

`requiredwithout:_foo,bar,...`

如果缺少任意一个指定的字段，则此字段为必填。

`requiredwithout_all:_foo,bar,...`

如果所有指定的字段都没有值，则此字段为必填。

`same:field`

验证字段值和指定的字段（`field`）值是否相同。

`size:value`

验证字段值的大小是否符合指定的 `value` 值。对于字符串来说，`value` 为字符数。对于数字来说，`value` 为某个整数值。对文件来说，`size` 对应的是文件大小（单位 kb）。

`string`

验证字段值的类型是否为字符串。如果你允许字段的值为 `null`，那么你应该将 `nullable` 规则附加到字段中。

`timezone`

验证字段值是否是有效的时区，会根据 PHP 的 `timezone_identifiers_list` 函数来判断。

`unique:table,column,except,idColumn`

在指定的数据表中，验证字段必须是唯一的。如果没有指定 `column`，将会使用字段本身的名称。

指定一个特定的字段名称:

```
'email' => 'unique:users,email_address'
```

自定义数据库连接

有时，您可能需要为验证程序所做的数据库查询设置自定义连接。如上面所示，如上所示，将 `unique: users` 设置为验证规则将使用默认数据库连接来查询数据库。如果要修改数据库连接，请使用「点」语法指定连接和表名：

```
'email' => 'unique:connection.users,email_address'
```

强迫 Unique 规则忽略指定 ID:

有时候，你希望在进行字段唯一性验证时对指定 ID 进行忽略。例如，在「更新个人资料」页面会包含用户名、邮箱等字段。这时你会想要验证更新的 E-mail 值是否为唯一的。如果用户仅更改了名称字段而没有改 E-mail 字段，就不需要抛出验证错误，因为此用户已经是这个 E-mail 的拥有者了。若要用指定规则来忽略用户 ID，则应该把要发送的 ID 当作第三个参数：

为了指示验证器忽略用户的ID，我们将使用 `Rule` 类流畅地定义规则。在这个例子中，我们还将通过数组来指定验证规则，而不是使用 `|` 字符来分隔：

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'email' => [
        'required',
        Rule::unique('users')->ignore($user->id),
    ],
]);
```

如果你的数据表使用的主键名称不是 `id`，则可以在调用 `ignore` 方法时指定列的名称：

```
'email' => Rule::unique('users')->ignore($user->id, 'user_id')
```

增加额外的 Where 语句:

你也可以通过 `where` 方法指定额外的查询约束条件。例如，我们添加 `account_id` 为 `1` 约束条件：

```
'email' => Rule::unique('users')->where(function ($query) {
    $query->where('account_id', 1);
})
```

url

验证字段必需是有效的 URL 格式。

按条件增加规则

当字段存在的时候进行验证

在某些情况下，你可能只想在输入数据中有此字段时才进行验证。可通过增加 `sometimes` 规则到规则列表来实现：

```
$v = Validator::make($data, [
    'email' => 'sometimes|required|email',
]);
```

在上面的例子中，`email` 字段的验证只会在 `$data` 数组有此字段时才会进行。

复杂的条件验证

有时候你可能希望增加更复杂的验证条件，例如，你可以希望某个指定字段在另一个字段的值超过 100 时才为必填。或者当某个指定字段有值时，另外两个字段要拥有符合的特定值。增加这样的验证条件并不难。首先，利用你熟悉的 `static rules` 来创建一个 `Validator` 实例：

```
$v = Validator::make($data, [
    'email' => 'required|email',
    'games' => 'required|numeric',
]);
```

假设我们有一个专为游戏收藏家所设计的网页应用程序。如果游戏收藏家收藏超过一百款游戏，我们会希望他们来说明下为什么他们会拥有这么多游戏。比如说他们有可能经营了一家二手游戏商店，或者只是为了享受收集的乐趣。为了在特定条件下加入此验证需求，可以在 `Validator` 实例中使用 `sometimes` 方法。

```
$v->sometimes('reason', 'required|max:500', function ($input) {
    return $input->games >= 100;
});
```

传入 `sometimes` 方法的第一个参数是我们要用条件认证的字段名称。第二个参数是我们想使用的验证规则。闭包作为第三个参数传入，如果其返回 `true`，则额外的规则就会被加入。这个方法可以轻松的创建复杂的条件式验证。你甚至可以一次对多个字段增加条件式验证：

```
$v->sometimes(['reason', 'cost'], 'required', function ($input) {
    return $input->games >= 100;
});
```

{tip} 传入 闭包 的 `$input` 参数是 `Illuminate\Support\Fluent` 实例，可用来访问你的输入或文件对象。

验证数组

验证基于数组的表单输入字段并不一定是一件痛苦的事情。要验证指定数组输入字段中的每一个 email 是否唯一，可以这么做：

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

同理，你在语言文件定义验证信息的时候可以使用星号 * 字符，可以更加容易的在基于数组格式的字段中使用相同的验证信息：

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

自定义验证规则

Laravel 提供了许多有用的验证规则。但你可能想自定义一些规则。注册自定义验证规则的方法之一，就是使用 `Validator Facade` 中的 `extend` 方法，让我们在 [服务提供者](#) 中使用这个方法来注册自定义的验证规则：

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\Validator;

class AppServiceProvider extends ServiceProvider
{
    /**
     * 启动任意应用程序服务。
     *
     * @return void
     */
    public function boot()
    {
        Validator::extend('foo', function ($attribute, $value, $parameters, $validator) {
            return $value == 'foo';
        });
    }
}
```

```

    /**
     * 注册服务容器。
     *
     * @return void
     */
    public function register()
    {
        //
    }
}

```

自定义的验证闭包接收四个参数：要被验证的属性名称 `$attribute`，属性的值 `$value`，传入验证规则的参数数组 `$parameters`，及 `Validator` 实例。

除了使用闭包，你也可以传入类和方法到 `extend` 方法中：

```
Validator::extend('foo', 'FooValidator@validate');
```

自定义错误消息

另外你可能还需要为自定义规则来定义一个错误消息。这可以通过使用自定义内联消息数组或是在验证语言包中加入新的规则来实现。此消息应该被放在数组的第一级，而不是被放在 `custom` 数组内，这是仅针对特定属性的错误消息：

```

"foo" => "你的输入是无效的！",

"accepted" => ":attribute 必须被接受。",

// 其余的验证错误消息...

```

当你在创建自定义验证规则时，你可能需要定义占位符来取代错误消息。你可以像上面所描述的那样通过 `Validator Facade` 来使用 `replacer` 方法创建一个自定义验证器。通过 [服务提供者](#) 中的 `boot` 方法可以实现：

```

    /**
     * 启动任意应用程序服务。
     *
     * @return void
     */
    public function boot()
    {
        Validator::extend(...);

        Validator::replacer('foo', function ($message, $attribute, $rule, $parameters)
        {
            return str_replace(...);
        });
    }
}

```

```
}
```

隐式扩展功能

默认情况下，若有一个类似 `required` 这样的规则，当此规则被验证的属性不存在或包含空值时，其一般的验证规则（包括自定扩展功能）都将不会被运行。例如，当 `integer` 规则的值为 `null` 时 `unique` 将不会被运行：

```
$rules = ['name' => 'unique'];

$input = ['name' => null];

Validator::make($input, $rules)->passes(); // true
```

如果要在属性为空时依然运行此规则，则此规则必须暗示该属性为必填。要创建一个「隐式」扩展功能，可以使用 `Validator::extendImplicit()` 方法：

```
Validator::extendImplicit('foo', function ($attribute, $value, $parameters, $validator) {
    return $value == 'foo';
});
```

{note} 一个「隐式」扩展功能只会 暗示 该属性为必填。它的实际属性是否为无效属性或空属性主要取决于你。

6 前端

Laravel 的 Blade 模板引擎

- 简介
- 模板继承
 - 定义页面布局
 - 继承页面布局
- 组件 & Slots
- 显示数据
 - Blade & JavaScript 框架
- 控制结构
 - If 语句
 - 循环
 - 循环变量
 - 注释
 - PHP
- 引入子视图
 - 为集合渲染视图
- 堆栈
- 服务注入
- 扩充 Blade

简介

Blade 是 Laravel 提供的一个既简单又强大的模板引擎。和其他流行的 PHP 模板引擎不一样，Blade 并不限制你在视图中使用原生 PHP 代码。所有 Blade 视图文件都将被编译成原生的 PHP 代码并缓存起来，除非它被修改，否则不会重新编译，这就意味着 Blade 基本上不会给你的应用增加任何额外负担。Blade 视图文件使用 `.blade.php` 扩展名，一般被存放在 `resources/views` 目录。

模板继承

定义页面布局

Blade 的两个主要优点是 模板继承 和 区块 。

为方便开始，让我们先通过一个简单的例子来上手。首先，我们需要确认一个 "master" 的页面布局。因为大多数 web 应用是在不同的页面中使用相同的布局方式，我们可以很方便的定义这个 Blade 布局视图：

```
<!-- 文件保存于 resources/views/layouts/app.blade.php -->

<html>
  <head>
    <title>应用程序名称 - @yield('title')</title>
```

```

</head>
<body>
    @section('sidebar')
        这是 master 的侧边栏。
    @show

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

```

如你所见，该文件包含了典型的 HTML 语法。不过，请注意 `@section` 和 `@yield` 命令。

`@section` 命令正如其名字所暗示的一样是用来定义一个视图区块的，而 `@yield` 指令是用来显示指定区块的内容的。

现在，我们已经定义好了这个应用程序的布局，让我们接着来定义一个继承此布局的子页面。

继承页面布局

当定义子页面时，你可以使用 Blade 提供的 `@extends` 命令来为子页面指定其所「继承」的页面布局。当子页面继承布局之后，即可使用 `@section` 命令将内容注入于布局的 `@section` 区块中。切记，在上面的例子里，布局中使用 `@yield` 的地方将会显示这些区块中的内容：

```

<!-- Stored in resources/views/child.blade.php -->

@extends('layouts.app')

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

@section('sidebar')
    @@parent

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

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

```

在上面的例子里，`sidebar` 区块利用了 `@@parent` 命令追加布局中的 `sidebar` 区块中的内容，如果不使用则会覆盖掉布局中的这部分内容。`@@parent` 命令会在视图被渲染时替换为布局中的内容。

当然，可以通过在路由中使用全局辅助函数 `view` 来返回 Blade 视图：

```

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

```

组件 & Slots

组件和 slots 能提供类似于区块和布局的好处；不过，一些人可能发现组件和 slots 更容易理解。首先，让我们假设一个会在我们应用中重复使用的「警告」组件：

```
<!-- /resources/views/alert.blade.php -->

<div class="alert alert-danger">
    {{ $slot }}
</div>
```

`{% raw %}{{ $slot }} {% endraw %}` 变量将包含我们希望注入到组件的内容。现在，我们可以使用 `@component` 指令来构造这个组件：

```
@component('alert')
    <strong>哇！</strong> 出现了一些问题！
@endcomponent
```

有些时候它对于定义组件的多个 slots 是非常有帮助的。让我们修改我们的警告组件，让它支持注入一个「标题」。已命名的 slots 将显示「相对应」名称的变量的值：

```
<!-- /resources/views/alert.blade.php -->

<div class="alert alert-danger">
    <div class="alert-title">{{ $title }}</div>

    {{ $slot }}
</div>
```

现在，我们可以使用 `@slot` 指令注入内容到已命名的 slot 中，任何没有被 `@slot` 指令包裹住的内容将传递给组件中的 `$slot` 变量：

```
@component('alert')
    @slot('title')
        拒绝
    @endslot

    你没有权限访问这个资源！
@endcomponent
```

传递额外的数据给组件

有时候你可能需要传递额外的数据给组件。为了解决这个问题，你可以传递一个数组作为第二个参数传递给 `@component` 指令。所有的数据都将以变量的形式传递给组件模版：

```
@component('alert', ['foo' => 'bar'])
...
@endcomponent
```

显示数据

你可以使用「中括号」包住变量以显示传递至 Blade 视图的数据。如下面的路由设置：

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

你可以像这样显示 `name` 变量的内容：

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

当然也不是说一定只能显示传递至视图的变量内容。你也可以显示 PHP 函数的结果。事实上，你可以在 Blade 中显示任意的 PHP 代码：

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

{note} Blade `{{ }}` 语法则会自动调用 PHP `htmlspecialchars` 函数来避免 XSS 攻击。

当数据存在时输出

有时候你可能想要输出一个变量，但是你并不确定这个变量是否已经被定义，我们可以用像这样的冗长 PHP 代码表达：

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

事实上，Blade 提供了更便捷的方式来代替这种三元运算符表达式：

```
{% raw %}{{ $name or 'Default' }} {{ endraw %}}
```

在这个例子中，如果 `$name` 变量存在，它的值将被显示出来。但是，如果它不存在，则会显示 `Default`。

显示未转义过的数据

在默认情况下，Blade 模板中的 `{% raw %}{{ }} {{ endraw %}}` 表达式将会自动调用 PHP `htmlspecialchars` 函数来转义数据以避免 XSS 的攻击。如果你不想你的数据被转义，你可以使用下面的语法：

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

{note} 要非常小心处理用户输入的数据时，你应该总是使用 `{}{}` 语法来转义内容中的任何的 HTML 元素，以避免 XSS 攻击。

Blade & JavaScript 框架

由于很多 JavaScript 框架都使用花括号来表明所提供的表达式，所以你可以使用 `@` 符号来告知 Blade 渲染引擎你需要保留这个表达式原始形态，例如：

```
<h1>Laravel</h1>

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

在这个例子里，`@` 符号最终会被 Blade 引擎剔除，并且 `% raw {{ name }} %` 表达式会被原样的保留下，这样就允许你的 JavaScript 框架来使用它了。

`@verbatim` 指令

如果你需要在页面中大片区块中展示 JavaScript 变量，你可以使用 `@verbatim` 指令来包裹 HTML 内容，这样你就不需要为每个需要解析的变量增加 `@` 符号前缀了：

```
@verbatim
<div class="container">
    Hello, % raw {{ name }} %.
</div>
@endverbatim
```

控制结构

除了模板继承与数据显示的功能以外，Blade 也给一般的 PHP 结构控制语句提供了方便的缩写，比如条件表达式和循环语句。这些缩写提供了更为清晰简明的方式来使用 PHP 的控制结构，而且还保持与 PHP 语句的相似性。

If 语句

你可以通过 `@if`，`@elseif`，`@else` 及 `@endif` 指令构建 `if` 表达式。这些命令的功能等同于在 PHP 中的语法：

```
@if (count($records) === 1)
    我有一条记录!
@elseif (count($records) > 1)
    我有多条记录!
@else
    我没有任何记录!
@endif
```

为了方便，Blade 也提供了一个 `@unless` 命令：

```
@unless (Auth::check())
    你尚未登录。
@endunless
```

循环

除了条件表达式外，Blade 也支持 PHP 的循环结构，这些命令的功能等同于在 PHP 中的语法：

```
@for ($i = 0; $i < 10; $i++)
    目前的值为 {{ raw }}{{ $i }} {{ endraw }}
@endfor

@foreach ($users as $user)
    <p>此用户为 {{ $user->id }} </p>
@endforeach

@forelse ($users as $user)
    <li>{{ $user->name }} </li>
@empty
    <p>没有用户</p>
@endforelse

@while (true)
    <p>我永远都在跑循环。</p>
@endwhile
```

{tip} 当循环时，你可以使用 [循环变量](#) 来获取循环中有价值的信息，比如循环中的首次或最后的迭代。

当使用循环时，你可能也需要一些结束循环或者跳出当前循环的命令：

```
@foreach ($users as $user)
@if ($user->type == 1)
    @continue
@endif

<li>{{ $user->name }} </li>

@if ($user->number == 5)
    @break
@endif
@endforeach
```

你也可以使用命令声明包含条件的方式在一条语句中达到中断：

```

@foreach ($users as $user)
@continue($user->type == 1)

<li>{{ $user->name }} </li>

@break($user->number == 5)
@endforeach

```

循环变量

当循环时，你可以在循环内访问 `$loop` 变量。这个变量可以提供一些有用的信息，比如当前循环的索引，当前循环是不是首次迭代，又或者当前循环是不是最后一次迭代：

```

@foreach ($users as $user)
@if ($loop->first)
    This is the first iteration.
@endif

@if ($loop->last)
    This is the last iteration.
@endif

<p>This is user {{ $user->id }} </p>
@endforeach

```

如果你是在一个嵌套的循环中，你可以通过使用 `$loop` 变量的 `parent` 属性来获取父循环中的 `$loop` 变量：

```

@foreach ($users as $user)
@foreach ($user->posts as $post)
@if ($loop->parent->first)
    This is first iteration of the parent loop.
@endif
@endforeach
@endforeach

```

`$loop` 变量也包含了其它各种有用的属性：

属性	描述
<code>\$loop->index</code>	当前循环所迭代的索引，起始为 0。
<code>\$loop->iteration</code>	当前迭代数，起始为 1。
<code>\$loop->remaining</code>	循环中迭代剩余的数量。
<code>\$loop->count</code>	被迭代项的总数量。
<code>\$loop->first</code>	当前迭代是否是循环中的首次迭代。

<code>\$loop->last</code>	当前迭代是否是循环中的最后一次迭代。
<code>\$loop->depth</code>	当前循环的嵌套深度。
<code>\$loop->parent</code>	当在嵌套的循环内时，可以访问到父循环中的 <code>\$loop</code> 变量。

注释

Blade 也允许在页面中定义注释，然而，跟 HTML 的注释不同的是，Blade 注释不会被包含在应用程序返回的 HTML 内：

```
{% raw %}{{-- 此注释将不会出现在渲染后的 HTML --}} {% endraw %}
```

PHP

在某些情况下，它对于你在视图文件中嵌入 php 代码是非常有帮助的。你可以在你的模版中使用 Blade 提供的 `@php` 指令来执行一段纯 PHP 代码：

```
@php
//
@endphp
```

{tip} 虽然 Blade 提供了这个功能，但频繁地使用也同时意味着你在你的模版中嵌入了太多的逻辑了。

引入子视图

你可以使用 Blade 的 `@include` 命令来引入一个已存在的视图，所有在父视图的可用变量在被引入的视图中都是可用的。

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

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

尽管被引入的视图会继承父视图中的所有数据，你也可以通过传递额外的数组数据至被引入的页面：

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

当然，如果你尝试使用 `@include` 去引用一个不存在的视图，Laravel 会抛出错误。如果你想引入一个视图，而你又无法确认这个视图存在与否，你可以使用 `@includeIf` 指令：

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

{note} 请避免在 Blade 视图中使用 `__DIR__` 及 `__FILE__` 常量，因为他们会引用视图被缓存的位置。

为集合渲染视图

你可以使用 Blade 的 `@each` 命令将循环及引入结合成一行代码：

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

第一个参数为每个元素要渲染的子视图，第二个参数是你要迭代的数组或集合，而第三个参数为迭代时被分配至子视图中的变量名称。举个例子，如果你需要迭代一个 `jobs` 数组，通常子视图会使用 `job` 作为变量来访问 `job` 信息。子视图使用 `key` 变量作为当前迭代的键名。

你也可以传递第四个参数到 `@each` 命令。当需要迭代的数组为空时，将会使用这个参数提供的视图来渲染。

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

堆栈

Blade 也允许你在其它视图或布局中为已经命名的堆栈中压入数据，这在子视图中引入必备的 JavaScript 类库时尤其有用：

```
@push('scripts')
    <script src="/example.js"></script>
@endpush
```

你可以根据需要多次压入堆栈，通过 `@stack` 命令中键入堆栈的名字来渲染整个堆栈：

```
<head>
    <!-- Head Contents -->

    @stack('scripts')
</head>
```

服务注入

你可以使用 `@inject` 命令来从 Laravel service container 中取出服务。传递给 `@inject` 的第一个参数为置放该服务的变量名称，而第二个参数为你想要解析的服务的类或是接口的名称：

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

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

拓展 Blade

Blade 甚至允许你使用 `directive` 方法来注册自己的命令。当 Blade 编译器遇到该命令时，它将会带参数调用提供的回调函数。

以下例子会创建一个把指定的 `$var` 格式化的 `@datetime($var)` 命令：

```
<?php

namespace App\Providers;

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

class AppServiceProvider extends ServiceProvider
{
    /**
     * 运行服务注册后的启动进程。
     *
     * @return void
     */
    public function boot()
    {
        Blade::directive('datetime', function ($expression) {
            return "<?php echo ($expression)->format('m/d/Y H:i'); ?>";
        });
    }

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

如你所见，我们可以使用链式调用 `format` 方法的表述方式传递到指令。所以，在这个例子里，最终该指令生成了的 PHP 代码如下：

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

{note} 在更新 Blade 指令的逻辑后，你将需要删除所有已缓存的 Blade 视图，使用 `view:clear` Artisan 命令来清除被缓存的视图。

Laravel 的本地化功能

- 简介
- 定义翻译语句
 - 使用短键
 - 使用翻译语句作为键
- 提取翻译语句
 - 翻译语句中的参数替换
 - 复数
- 重写扩展包的语言包

简介

Laravel 的本地化功能提供方便的方法来获取多语言的字符串，让你的网站可以简单的支持多语言。

语言包存放在 `resources/lang` 目录下的文件里。在此目录中应该有应用对应支持的语言并将其对应到每一个子目录：

```
/resources
  /lang
    /en
      messages.php
    /es
      messages.php
```

语言包简单地返回键值和字符串数组，例如：

```
<?php

return [
    'welcome' => 'Welcome to our application'
];
```

切换语言

应用的默认语言保存在 `config/app.php` 配置文件中。当然，你可以根据需求自由的修改当前设置，可以使用 `App facade` 的 `setLocale` 方法动态地更改现有语言：

```
Route::get('welcome/{locale}', function ($locale) {
    App::setLocale($locale);

    //
});
```

你也可以设置「备用语言」，它将会在当现有语言没有指定语句时被使用。就像默认语言那样，备用语言也可以在 `config/app.php` 配置文件设置：

```
'fallback_locale' => 'en',
```

指定当前语言

你可以使用 `App facade` 的 `getLocale` 及 `isLocale` 方法指定当前的语言环境或者检验当前语言是否是给定的值：

```
$locale = App::getLocale();

if (App::isLocale('en')) {
    //
}
```

定义翻译语句

使用短键

通常，语言包存放在 `resources/lang` 目录下的文件里。在此目录中应该有应用对应支持的语言并将其对应到每一个子目录：

```
/resources
  /lang
    /en
      messages.php
    /es
      messages.php
```

语言包简单地返回键值和字符串数组，例如：

```
<?php

// resources/lang/en/messages.php

return [
    'welcome' => 'Welcome to our application'
];
```

使用翻译语句作为键

对于有大量翻译需求的应用，如果每一句翻译语句都使用「短键」来定义，那么当你在视图中尝试去引用这些「短键」的时候，很容易变得混乱，分不清哪个对应哪个。因此，Laravel 也提供支持使用「默认语言」的翻译语句作为键，来定义其他语言的翻译语句。

使用翻译语句作为键的语言包需要在 `resources/lang` 目录下保存为 JSON 文件。例如，如果你的应用中有西班牙语的语言包，你应该新建一个 `resources/lang/es.json` 文件：

```
{
    "I love programming.": "Me encanta la programación."
}
```

获取翻译语句

你可以使用 `__` 辅助函数来获取翻译语句，`__` 方法接受文件名和键值作为其第一个参数。例如，让我们提取 `resources/lang/messages.php` 中的 `welcome`：

```
echo __('messages.welcome');

echo __('I love programming.');
```

当然，如果你使用 [Blade 模板引擎](#)，那么你可以在视图文件中使用 `{{ }}` 语法或者使用 `@lang` 指令来输出语句：

```
{{ __('messages.welcome') }}

@lang('messages.welcome')
```

如果指定的语句不存在，`__` 方法则会简单的返回这个键名。所以，如果上述示例中的键不存在，那么 `__` 方法则会返回 `messages.welcome`。

翻译语句中的参数替换

如果需要，你也可以在翻译语句中定义占位符。所有的占位符都使用的 `:` 开头。例如，你可以自定义一则欢迎消息的占位符：

```
'welcome' => 'Welcome, :name',
```

你可以在 `__` 方法中传递一个数组作为第二个参数，它会将数组的值替换到语言内容的占位符中：

```
echo ___('messages.welcome', ['name' => 'dayle']);
```

如果你的占位符中包含了首字母大写或者全体大写，翻译过来的内容也会相应的做相应的处理：

```
'welcome' => 'Welcome, :NAME', // Welcome, DAYLE
'goodbye' => 'Goodbye, :Name', // Goodbye, Dayle
```

复数

复数是个复杂的问题，不同语言对于复数有不同的规则。使用管道符 `|`，可以区分单复数字符串格式：

```
'apples' => 'There is one apple|There are many apples',
```

你甚至可以创建使用更复杂的复数规则，例如根据数量的范围不同来指定不同翻译语句：

```
'apples' => '{0} There are none|[1,19] There are some|[20,*] There are many',
```

当你定义完复数语句条件的时候，你可以使用 `trans_choice` 方法来设置「总数」以获取符合对应条件的复数翻译语句。例如，在这个例子中，设置「总数」为 10，符合数量范围 1 至 19，所以会得到 `There are some` 这条复数语句：

```
echo trans_choice('messages.apples', 10);
```

重写扩展包的语言包

部分扩展包有自己的语言包，你可以通过在 `resources/lang/vendor/{package}/{locale}` 放置文件来重写它们，而不是直接修改扩展包的核心文件。

例如，你需要重写 `skyrim/hearthfire` 扩展包的英文语言包 `messages.php`，则需要把文件放置在 `resources/lang/vendor/hearthfire/en/messages.php`。在这个文件中定义你想要重写的翻译语句，所有没有重写的语句将会加载扩展包的语言包中原来的语句。

Laravel 的前端资源处理 JavaScript 和 CSS 构建

- 简介
- 编写 CSS
- 编写 JavaScript
 - 编写 Vue 组件

简介

Laravel 并没有规定你使用哪个 JavaScript 或 CSS 预处理器，不过默认提供了对大多数应用都很适用的 [Bootstrap](#) 和 [Vue](#) 来作为起点。 Laravel 默认使用 [NPM](#) 安装这些前端的依赖。

CSS

[Laravel Mix](#) 提供了一个简洁、富有表现力的 API 来编译 SASS 或 Less，这些 CSS 预处理语言扩充了 CSS 语言，增加了诸如变量、混合（mixins）及其他一些强大的功能让编写 CSS 代码变得更加有趣。

尽管我们会在这里简要的讨论 CSS 编译的内容，但是，你应该访问 [Laravel Mix 文档](#) 获取更多关于编译 SASS 或 Less 的信息。

JavaScript

Laravel 并不需要你使用特定的 JavaScript 框架或者库来构建应用程序。事实上，你也可以完全不用 JavaScript。不过， Laravel 自带了用 [Vue](#) 实现的基本脚手架代码来帮你更轻松的开始现代化 JavaScript 编码。Vue 提供了强大的组件化 API 用来构建健壮的 JavaScript 应用程序。

编写 CSS

在 Laravel 的根目录中的 `Package.json` 文件引入了 `bootstrap-sass` 依赖包可以帮助你使用 Bootstrap 制作应用程序的前端原型。不过，你可以根据自己应用程序的需要在 `package.json` 灵活的添加或者移除依赖包。使用 Bootstrap 框架来构建你的 Laravel 应用程序并不是必选项，它只是给那些想用它的人提供一个很好的起点。

编译 CSS 代码之前，需要你先使用 NPM 来安装前端的依赖：

```
npm install
```

使用 `npm install` 成功安装依赖之后，你就可以使用 [Laravel Mix](#) 来将 SASS 文件编译为纯 CSS。`. npm run dev` 命令会执行处理 `webpack.mix.js` 文件中的指令。通常情况下，编译好的 CSS 代码会被放置在 `public/css` 目录：

```
npm run dev
```

默认情况下，`webpack.mix.js` 会编译 `resources/assets/sass/app.scss` SASS 文件。`app.scss` 文件导入了一个包含 SASS 变量的文件，加载了 Bootstrap 框架，这对大多数程序来说是一个很好的出发点。你也可以根据自己的需要去定制 `app.scss` 文件的内容，甚至使用完全不同的预处理器，详细配置见[配置 Laravel Mix](#)。

编写 JavaScript

在项目根目录中的 `package.json` 可以找到应用程序的所有 JavaScript 依赖。它和 `composer.json` 文件类似，不同的是它指定的是 JavaScript 的依赖而不是 PHP 的依赖。使用[Node 包管理器 \(NPM\)](#) 来安装这些依赖包：

```
npm install
```

Laravel `package.json` 文件默认会包含一些依赖包来帮助你开始建立 JavaScript 应用程序，例如 `vue` 和 `axios`。你可以根据自己程序的需要在 `package.json` 中添加或者移除依赖。

成功安装依赖之后，你就可以使用 `npm run dev` 命令来[编译资源文件](#)。Webpack 是一个为现代 JavaScript 应用而生的模块构建工具。当你运行 `npm run dev` 命令时，Webpack 会执行 `webpack.mix.js` 文件中的指令：

```
npm run dev
```

默认情况下，`webpack.mix.js` 会编译 SASS 文件和 `resources/assets/js/app.js` 文件。你可以在 `app.js` 文件中注册你的 Vue 组件，或者如果你更喜欢其他的框架，也可以在这里进行配置。编译好的 JavaScript 文件通常会放置在 `public/js` 目录。

{tip} `app.js` 会加载 `resources/assets/js/bootstrap.js` 文件来启动、配置 Vue，Vue Resource，jQuery，以及其他 JavaScript 依赖。如果你有额外的 JavaScript 依赖需要去配置，你也可以在这个文件中完成。

编写 Vue 组件

全新安装的 Laravel 程序默认会在 `resources/assets/js/components` 中包含一个 `Example.vue` 的 Vue 组件。`Example.vue` 文件是一个[单文件 Vue 组件](#)的示例，单文件 Vue 组件允许我们在同一个文件中编写 JavaScript 和 HTML 模板，它提供了一种非常方便的方式去构建 JavaScript 驱动的应用程序。这个示例组件注册在 `app.js` 文件。

```
Vue.component('example', require('./components/Example.vue'));
```

在应用程序中使用示例组件，你只需要简单的将其放到你的 HTML 模板之中。例如，在你运行 `make:auth` Artisan 命令去生成应用的用户认证和注册的脚手架页面后，你可以把组件放到 `home.blade.php` Blade 模板：

```
@extends('layouts.app')

@section('content')
    <example></example>
@endsection
```

{tip} 谨记，你需要在每次修改 Vue 组件后都需要运行 `npm run dev` 命令。或者，你可以使用 `npm run watch` 命令来监控并在每次文件被修改时自动重新编译组件。

当然，如果你对学习更多编写 Vue 组件的内容感兴趣，你可以读一下 [Vue 官方引导文档](#)，它提供了一个透彻、易懂的文档让你一览 Vue 框架的概貌。

Laravel 的资源任务编译器 Laravel Mix

- 简介
- 安装与配置
- 运行 Mix
- 使用样式
 - Less
 - Sass
 - Stylus
 - PostCSS
 - 纯 CSS
 - URL 处理
 - 资源地图
- 使用脚本
 - 提取 Vendor
 - React
 - 原生 JS
 - 自定义 Webpack 配置
- 复制文件与目录
- 版本与缓存清除
- Browsersync 自动加载刷新
- 通知

简介

Laravel Mix 提供了简洁流畅的 API，让你能够为你的 Laravel 应用定义 Webpack 的编译任务。Mix 支持许多常见的 CSS 与 JavaScript 预处理器，通过简单的方法，你可以轻松的管理资源。例如：

```
mix.js('resources/assets/js/app.js', 'public/js')
    .sass('resources/assets/sass/app.scss', 'public/css');
```

如果你曾经对于使用 Webpack 及编译资源感到困惑，那么你绝对会爱上 Laravel Mix。当然，在 Laravel 应用开发中使用 Mix 并不是必须的，你也可以选择任何你喜欢的资源编译工具，或者不使用任何工具。

安装

安装 Node

在开始使用 Mix 之前，你必须先确定你的开发环境上有安装 Node.js 和 NPM。

```
node -v
```

```
npm -v
```

默认情况下, Laravel Homestead 会包含你所需的一切。当然, 如果你没有使用 Vagrant, 那么你可以浏览 [nodejs](#) 下载可视化的安装工具来安装最新版的 Node 和 NPM.

Laravel Mix

剩下的只需要安装 Laravel Mix! 随着新安装的 Laravel, 你会发现根目录下有个名为 `package.json` 的文件。就如同 `composer.json` 文件, 只不过 `package.json` 文件定义的是 Node 的依赖, 而不是 PHP。你可以使用以下的命令安装依赖扩展包:

```
npm install
```

如果你使用的是 Windows 系统或运行在 Windows 系统上的 VM, 你需要在运行 `npm install` 命令时将 `--no-bin-links` 开启:

```
npm install --no-bin-links
```

使用

Mix 基于 [Webpack](#) 的配置, 所以运行定义于 `package.json` 文件中的 NPM 脚本即可执行 Mix 的编译任务:

```
// 运行所有 Mix 任务...
npm run dev

// 运行所有 Mix 任务和压缩资源输出
npm run production
```

监控资源文件修改

`npm run watch` 会在你的终端里持续运行, 监控资源文件是否有发生改变。在 `watch` 命令运行的情况下, 一旦资源文件发生变化, Webpack 会自动重新编译:

```
npm run watch
```

你可能会发现, 在某些环境下, 当你改变了你的文件的时候而 Webpack 并没有同步更新。如果在你的系统中出现了这个问题, 你可以考虑使用 `watch-poll` 命令:

```
npm run watch-poll
```

使用样式

项目根目录的 `webpack.mix.js` 文件是资源编译的入口。可以把它看作是 Webpack 的配置文件。Mix 任务可以使用链式调用的写法来定义你的资源文件该如何进行编译。

Less

`less` 方法可以让你将 [Less](#) 编译为 CSS。下面的命令可以把 `app.less` 编译为 `public/css/app.css`。

```
mix.less('resources/assets/less/app.less', 'public/css');
```

多次调用 `less` 方法可以编译多个文件:

```
mix.less('resources/assets/less/app.less', 'public/css')
    .less('resources/assets/less/admin.less', 'public/css');
```

如果你想自定义编译后的 CSS 文件名, 你可以传递一个完整的路径到 `less` 方法的第二个参数:

```
mix.less('resources/assets/less/app.less', 'public/stylesheets/styles.css');
```

如果你需要重写 [底层 Less 插件选项](#), 你可以传递一个对象到 `mix.less()` 的第三个参数:

```
mix.less('resources/assets/less/app.less', 'public/css', {
    strictMath: true
});
```

Sass

`sass` 方法可以让你将 [Sass](#) 编译为 CSS。你可以使用此方法:

```
mix.sass('resources/assets/sass/app.scss', 'public/css');
```

同样的, 如同 `less` 方法, 你可以将多个 Sass 文件编译为多个 CSS 文件, 甚至可以自定义生成的 CSS 的输出目录:

```
mix.sass('resources/assets/sass/app.sass', 'public/css')
    .sass('resources/assets/sass/admin.sass', 'public/css/admin');
```

另外 [Node-Sass 插件选项](#) 可以通过传递第三个参数来重写:

```
mix.sass('resources/assets/sass/app.sass', 'public/css', {
    precision: 5
});
```

Stylus

类似于 Less 和 Sass， `stylus` 方法允许你编译 Stylus 为 CSS:

```
mix.stylus('resources/assets/stylus/app.styl', 'public/css');
```

你也可以安装其他的 Stylus 插件，例如 [Rupture](#)。首先，通过 NPM(`npm install rupture`) 来安装插件，然后在调用 `mix.stylus()` 的时候引用插件:

```
mix.stylus('resources/assets/stylus/app.styl', 'public/css', {
    use: [
        require('rupture')()
    ]
});
```

PostCSS

[PostCSS](#)，一个用来转换 CSS 的强大工具，已经包含在 Laravel Mix 中。默认，Mix 利用了流行的 Autoprefixer 插件来自动添加所需要的 CSS3 供应商前缀。不过，你也可以自由添加任何适合你应用程序的插件。首先，通过 NPM 来安装想要的插件，然后在你的 `webpack.mix.js` 文件中引用:

```
mix.sass('resources/assets/sass/app.scss', 'public/css')
.options({
    postCss: [
        require('postcss-css-variables')()
    ]
});
```

纯 CSS

如果你只是想将一些纯 CSS 样式合并成单个的文件，你可以使用 `styles` 方法。

```
mix.styles([
    'public/css/vendor/normalize.css',
    'public/css/vendor/videojs.css'
], 'public/css/all.css');
```

URL 处理

由于 Laravel Mix 是建立在 Webpack 之上，所以了解一些 Webpack 概念就非常有必要。编译 CSS 的时候，Webpack 会重写和优化那些你样式表中调用 `url()` 的地方。虽然可能一开始听起来觉得奇怪，不过这确实是一个强大的功能。试想一下我们编译一个包含相对路径图片的 Sass 文件:

```
.example {
```

```
background: url('../images/example.png');
}
```

{note} `url()` 方法会在 URL 重写中排除绝对路径。例如 `url('/images/thing.png')` 或者 `url('http://example.com/images/thing.png')` 不会被修改。

Laravel Mix 和 Webpack 默认会找到 `example.png`，把它复制到你的 `public/images` 目录下，然后在你生成的样式表中重写 `url()`。这样，你编译之后的 CSS 会变成：

```
.example {
  background: url(/images/example.png?d41d8cd98f00b204e9800998ecf8427e);
}
```

与此功能相同，可能你的现在的文件夹结构已经按照你喜欢的方式来配置。如果是这种情况，你可以像这样来禁用 `url()` 重写：

```
mix.sass('resources/assets/app/app.scss', 'public/css')
  .options({
    processCssUrls: false
 });
```

如果在你的 `webpack.mix.js` 文件这样配置之后，Mix 将不再匹配 `url()` 或者复制 assets 到你的 `public` 目录。换句话来说，编译后的 CSS 跟你原来输入的看起来一样：

```
.example {
  background: url("../images/thing.png");
}
```

资源地图

`source maps` 默认状态下是禁用的，你可以通过在 `webpack.mix.js` 文件中调用 `mix.sourceMaps()` 方法来开启。它会带来一些编译成本，但在使用编译后的资源文件时可以更方便的在浏览器中进行调试：

```
mix.js('resources/assets/js/app.js', 'public/js')
  .sourceMaps();
```

使用脚本

Mix 也提供了一些函数来帮助你使用 JavaScript 文件，像是编译 ECMAScript 2015、模块编译、压缩、及简单的串联纯 JavaScript 文件。更棒的是，这些都不需要自定义的配置：

```
mix.js('resources/assets/js/app.js', 'public/js');
```

这一行简单的代码，支持：

- ECMAScript 2015 语法. - Modules - 编译 `vue` 文件. - 针对生产环境压缩代码.

Vendor Extraction

将应用程序的 JavaScript 与依赖库捆绑在一起的一个潜在缺点是，使得长期缓存更加困难。如，对应用程序代码的单独更新将强制浏览器重新下载所有依赖库，即使它们没有更改。

如果你打算频繁更新应用程序的 JavaScript，应该考虑将所有的依赖库提取到单独文件中。这样，对应用程序代码的更改不会影响 vendor.js 文件的缓存。Mix 的 extract 方法可以轻松做到：

```
mix.js('resources/assets/js/app.js', 'public/js')
    .extract(['vue'])
```

extract 方法接受你希望提取到 vendor.js 文件中的所有的依赖库或模块的数组。使用以上代码片段作为示例，Mix 将生成以下文件：

- `public/js/manifest.js`: *Webpack 显示运行时* - `public/js/vendor.js`: *依赖库* - `public/js/app.js`: *应用代码*

为了避免 JavaScript 错误，请务必按正确的顺序加载这些文件：

```
<script src="/js/manifest.js"></script>
<script src="/js/vendor.js"></script>
<script src="/js/app.js"></script>
```

React

Mix 可以自动安装 Babel 插件来支持 React。你只需要替换你的 mix.js() 变成 mix.react() 即可：

```
mix.react('resources/assets/js/app.jsx', 'public/js');
```

在背后，React 会自动下载，并且自动下载适当的 babel-preset-react Babel 插件。

原生 JS

类似使用 mix.styles() 来组合多个样式表一样，你也可以使用 scripts() 方法来合并并且压缩多个 JavaScript 文件：

```
mix.scripts([
    'public/js/admin.js',
    'public/js/dashboard.js'
], 'public/js/all.js');
```

这个选项对于那些没有使用 Webpack 的历史项目非常有用。

{tip} `mix.babel()` 和 `mix.scripts()` 有点稍微不一样。`babel` 方法用法和 `scripts` 一样；不过，这些文件会经过 Babel 编译，把所有 ES2015 的代码转换为原生 JavaScript，这样所有浏览器都能识别。

自定义 Webpack 配置

Laravel Mix 默认引用了一个预先配置的 `webpack.config.js` 文件，以便尽快启动和运行。有时，你可能需要手动修改此文件。例如，你可能有一个特殊的加载器或插件需要被引用，或者也许你喜欢使用 Stylus 而不是 Sass。在这种情况下，你有两个选择：

合并

Mix 提供了一个有用的 `webpackConfig` 方法，允许合并任何 `Webpack` 配置以覆盖默认配置。这是一个非常好的选择，你不需要复制和维护 `webpack.config.js` 文件。`webpackConfig` 方法接受一个对象，该对象应包含要应用的任何 [Webpack 配置项](#)：

```
mix.webpackConfig({
  resolve: {
    modules: [
      path.resolve(__dirname, 'vendor/laravel/spark/resources/assets/js')
    ]
  }
});
```

复制文件与目录

`copy` 方法可以复制文件与目录至新位置。当 `node_modules` 目录中的特定资源需要复制到 `public` 文件夹时会很有用。

```
mix.copy('node_modules/foo/bar.css', 'public/css/bar.css');
```

版本与缓存清除

许多的开发者会在它们编译后的资源文件中加上时间戳或是唯一的 token，强迫浏览器加载全新的资源文件以取代提供的旧版本代码副本。你可以使用 `version` 方法让 Mix 处理它们。

`version` 方法为你的文件名称加上唯一的哈希值，以防止文件被缓存：

```
mix.js('resources/assets/js/app.js', 'public/js')
  .version();
```

在为文件生成版本之后，你将不知道确切的文件名。因此，你应该在你的视图中使用 Laravel 的全局 `mix` PHP 辅助函数来正确加载名称被哈希后的文件。`mix` 函数会自动判断被哈希的文件名称：

```
<link rel="stylesheet" href="{{ mix('/css/app.css') }}>
```

在开发中通常是不需要版本化，你可能希望仅在运行 `npm run production` 的时候进行版本化：

```
mix.js('resources/assets/js/app.js', 'public/js');

if (mix.inProduction()) {
    mix.version();
}
```

Browsersync 自动加载刷新

[BrowserSync](#) 可以监控你的文件变化，并且无需手动刷新就可以把你的变化注入到浏览器中。你可以通过调用 `mix.browserSync()` 方法来启用这个功能支持：

```
mix.browserSync('my-domain.dev');

// 或者...

// https://browsersync.io/docs/options
mix.browserSync({
    proxy: 'my-domain.dev'
});
```

你可以通过传递一个字符串（代理）或者一个对象（BrowserSync 设置）给这个方法。接着，使用 `npm run watch` 命令来开启 Webpack 的开发服务器。现在，当你修改一个脚本或者 PHP 文件，看着浏览器立即刷新出来的页面来反馈你的改变。

通知

在可用的时候，Mix 会将每个包的编译是否成功以系统通知的方式反馈给你。如果你希望停用这些通知，可以通过 `disableNotifications` 方法实现：

```
mix.disableNotifications();
```

7 安全

Laravel 的用户认证系统

- 简介
 - 数据库注意事项
- 认证快速入门
 - 路由
 - 视图
 - 认证
 - 获取已认证的用户信息
 - 限制路由访问
 - 登入限流
- 手动认证用户
 - 记住用户
 - 其它认证方法
- HTTP 基础认证
 - 无状态 HTTP 基础认证
- 社交认证
- 增加自定义 Guard
- 增加自定义用户 Provider
 - 用户 Provider Contract
 - 用户认证 Contract
- 事件

简介

{tip} 想要快速起步？在一个全新的 Laravel 应用中运行 `php artisan make:auth` 和 `php artisan migrate` 命令，然后可以用浏览器访问 `http://your-app.dev/register` 或者你在程序中定义的其他 url。这个两个简单的命令就可以搭建好整个认证系统的脚手架。

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、标注好注释的选项配置。

在其核心代码中，Laravel 的认证组件由 `guards` 和 `providers` 组成，Guard 定义了用户在每个请求中如何实现认证，例如，Laravel 通过 `session guard` 来维护 Session 存储的状态和 Cookie。

Provider 定义了如何从持久化存储中获取用户信息，Laravel 底层支持通过 Eloquent 和数据库查询构建器两种方式来获取用户，如果需要的话，你还可以定义额外的 Provider。

如果看到这些名词觉得很困惑，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。

数据库注意事项

默认的 Laravel 在 `app` 文件夹中会含有 `App\User Eloquent 模型`。这个模型将使用默认的 Eloquent 认证来驱动。如果你的应用程序没有使用 Eloquent，请选择使用 Laravel 查询构造器的 `database` 认证驱动。

为 `App\User` 模型创建数据库表结构时，确认密码字段最少必须 60 字符长。保持字段原定的 255 字符长是个好选择。

`users` 数据表中必须含有 nullable、100 字符长的 `remember_token` 字段。当用户登录应用并勾选「记住我」时，这个字段将会被用来保存「记住我」 session 的令牌。

认证快速入门

Laravel 带有几个预设的认证控制器，它们被放置在 `App\Http\Controllers\Auth` 命名空间内，`RegisterController` 处理用户注册，`LoginController` 处理用户认证，`ForgotPasswordController` 处理重置密码的 e-mail 链接，`ResetPasswordController` 包含重置密码的逻辑。这些控制器使用了 trait 来包含所需要的方法，对于大多数的应用程序而言，你并不需要修改这些控制器。

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
```

该命令应该在新安装的应用下使用，它会生成 `layout` 布局视图，注册和登录视图，以及所有的认证路由，同时生成 `HomeController`，用来处理登录成功后会跳转到该控制器下的请求。

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。所有这些视图都基于 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，认证控制器已经（通过 trait）包含了注册及登录逻辑。

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 `/home`，你可以通过在 `LoginController`，`RegisterController` 和 `ResetPasswordController` 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 = '/';
```

如果跳转路径需要自定义逻辑来生成，你可以定义 `redirectTo` 方法来代替 `redirectTo` 属性：

```
protected function redirectTo()
{
    return '/path';
}
```

{tip} `redirectTo` 方法优先于 `redirectTo` 属性。

自定义用户名

Laravel 默认使用 `email` 字段来认证。如果你想用其他字段认证，可以在 `LoginController` 里面定义一个 `username` 方法

```
public function username()
{
    return 'username';
}
```

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 `LoginController`，`RegisterController` 和 `ResetPasswordController` 中定义 `guard` 方法，该方法需要返回一个 guard 实例：

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

protected function guard()
{
    return Auth::guard('guard-name');
}
```

自定义验证 / 存储

要修改新用户注册所必需的表单字段，或者自定义新用户字段如何存储到数据库，你可以修改 `RegisterController` 类。该类负责为应用验证输入参数和创建新用户。

`RegisterController` 的 `validator` 方法包含了新用户的验证规则，你可以按需要自定义该方法。

`RegisterController` 的 `create` 方法负责使用 [Eloquent ORM](#) 在数据库中创建新的 `App\User` 记录。当然，你也可以基于自己的需求自定义该方法。

获取已认证的用户信息

可以通过 `Auth facade` 来访问认证的用户。

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

// 获取当前已通过认证的用户...
$user = Auth::user();

// 获取当前已通过认证的用户id...
$id = Auth::id();
```

也有另外一种方法可以访问认证过的用户，就是通过 `Illuminate\Http\Request` 实例，请注意类型提示的类会被自动注入：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class ProfileController extends Controller
{
    /**
     * Update the user's profile.
     *
     * @param Request $request
     * @return Response
     */
    public function update(Request $request)
    {
        // $request->user() 返回认证过的用户的实例...
    }
}
```

检查用户是否登录

使用 `Auth facade` 的 `check` 方法来检查用户是否登录，如果已经登录，将会返回 `true`：

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

if (Auth::check()) {
    // 这个用户已经登录...
}
```

{tip} 尽管可以使用 `check` 方法来检查用户是否登录，在允许该用户访问特定的路由或控制器之前，可以使用中间件来检查用户是否认证过。要想得到更多信息，请阅读 [限制路由访问](#) 的文档。

限制路由访问

[路由中间件](#) 用于限定认证过的用户访问指定的路由，Laravel 提供了 `auth` 中间件来达到这个目的，而这个中间件被定义在 `Illuminate\Auth\Middleware\Authenticate` 中。因为这个中间件已经在 HTTP kernel 中注册了，只需要将它应用到路由定义中即可使用：

```
Route::get('profile', function () {
    // 只有认证过的用户能进来这里...
})->middleware('auth');
```

如果使用 [控制器类](#)，可以在构造器中调用 `middleware` 方法，来代替在路由中直接定义：

```
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。指定的 guard 对应配置文件 `auth.php` 中 `guards` 数组的某个键：

```
public function __construct()
{
    $this->middleware('auth:api');
}
```

登录限流

Laravel 内置的 `LoginController` 类提供 `Illuminate\Foundation\Auth\ThrottlesLogins` trait 允许你在应用程序中限制登录次数。默认情况下，如果用户在进行几次尝试后仍不能提供正确的凭证，将在一分钟内无法进行登录。这个限制会特别针对用户的用户名 / 邮件地址和他们的 IP 地址。

手动认证用户

当然，不一定要使用 Laravel 内置的认证控制器。如果选择删除这些控制器，可以直接调用 Laravel 的认证类来实现用户认证管理。不用担心，很简单。

我们可以利用 `Auth facade` 来访问 Laravel 的认证服务，因此需要确认在类的顶部导入 `Auth facade`。接下来让我们看一下 `Auth` 的 `attempt` 方法：

```
<?php

namespace App\Http\Controllers;
```

```

use Illuminate\Support\Facades\Auth;

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

```

`attempt` 方法会接受一个数组来作为第一个参数，这个数组的值可用来寻找数据库里的用户数据，所以在上面的例子中，用户通过 `email` 字段被取出，如果用户被找到了，数据库里经过哈希的密码将会与数组中哈希的 `password` 值比对，如果两个值一样的话就会开启一个通过认证的 session 给用户。

如果认证成功，`attempt` 方法将会返回 `true`，反之则为 `false`。

重定向器上的 `intended` 方法将会重定向用户回原本想要进入的页面，也可以传入一个回退 URI 至这个方法，以避免要转回的页面不可使用。

指定额外条件

可以加入除用户的邮箱及密码外的额外条件进行认证查找。例如，我们要确认用户是否被标示为 `active`：

```

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

```

{note} 在这些例子中，`email` 不是一个一定要有的选项，它仅仅是被用来当作例子，你可以用任何字段，只要它在数据库的意义等同于「用户名」。

访问指定 Guard 实例

可以通过 `Auth facade` 的 `guard` 方法来指定使用特定的 guard 实例。这样可以实现在应用不同部分管理用户认证时使用完全不同的认证模型或者用户表。

传递给 `guard` 方法 `guard` 名称必须是 `auth.php` 配置文件中 `guards` 的值之一：

```

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

```

```
//  
}
```

注销用户

要想让用户注销，你可以使用 `Auth facade` 的 `logout` 方法。这个方法会清除所有认证后加入到用户 `session` 的数据：

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

记住用户

如果你想要提供「记住我」的功能，你需要传入一个布尔值到 `attempt` 方法的第二个参数，在用户注销前 `session` 值都会被一直保存。`users` 数据表一定要包含一个 `remember_token` 字段，这是用来保存「记住我」令牌的。

```
if (Auth::attempt(['email' => $email, 'password' => $password], $remember)) {  
    // 这个用户被记住了...  
}
```

{tip} 如果使用 Laravel 内置的 `LoginController`，合适的「记住我」逻辑已经通过 traits 实现。

可以使用 `viaRemember` 方法来检查这个用户是否使用「记住我」 cookie 来做认证：

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

其它认证方法

用「用户实例」做认证

如果你需要使用存在的用户实例来登录，你需要调用 `login` 方法，并传入使用实例，这个对象必须是由 `Illuminate\Contracts\Auth\Authenticatable` contract 所实现。当然，`App/User` 模型已经实现了这个接口：

```
Auth::login($user);  
  
// 登录并且「记住」用户  
Auth::login($user, true);
```

你也可以指定 guard 实例：

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

通过用户 ID 做认证

使用 `loginUsingId` 方法来登录指定 ID 用户，这个方法接受要登录用户的主键：

```
Auth::loginUsingId(1);

// 登录并且「记住」用户
Auth::loginUsingId(1, true);
```

仅在本次认证用户

可以使用 `once` 方法来针对一次性认证用户，没有任何的 session 或 cookie 会被使用，这个对于构建无状态的 API 非常的有用，`once` 方法跟 `attempt` 方法拥有同样的传入参数：

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

HTTP 基础认证

[HTTP 基础认证](#) 提供一个快速的方法来认证用户，不需要任何「登录」页面。开始之前，先增加 `auth.basic` 中间件到你的路由，`auth.basic` 中间件已经被包含在 Laravel 框架中，所以你不需要定义它：

```
Route::get('profile', function () {
    // 只有认证过的用户可进入...
})->middleware('auth.basic');
```

一旦中间件被增加到路由上，当使用浏览器进入这个路由时，将自动的被提示需要提供凭证。默认情况下，`auth.basic` 中间件将会使用用户的 `email` 字段当作「用户名」。

FastCGI 的注意事项

如果是正在使用 FastCGI，则 HTTP 的基础认证可能无法正常运作，你需要将下面这几行加入你 `.htaccess` 文件中：

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

无状态 HTTP 基础认证

你可以使用 HTTP 基础认证而不用在 `session` 中设置用户认证用的 cookie，这个功能对 API 认证来说非常有用。为了达到这个目的，[定义一个中间件](#) 并调用 `onceBasic` 方法。如果从 `onceBasic` 方法没有返回任何响应的话，这个请求会直接传进应用程序中：

```
<?php

namespace Illuminate\Auth\Middleware;

use Illuminate\Support\Facades\Auth;

class AuthenticateOnceWithBasicAuth
{
    /**
     * Handle an incoming request.
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure $next
     * @return mixed
     */
    public function handle($request, $next)
    {
        return Auth::onceBasic() ?: $next($request);
    }
}
```

接着，[注册这个路由中间件](#)，然后将它增加在一个路由上：

```
Route::get('api/user', function () {
    // 只有认证过的用户可以进入...
})->middleware('auth.basic.once');
```

增加自定义的 Guard

你可以使用 `Auth` 的 `extend` 方法来自定义认证 Guard，你需要在 [服务提供者](#) 中放置此代码调用。因为 Laravel 已经提供 `AuthServiceProvider`，可以把代码放入其中：

```
<?php

namespace App\Providers;

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

class AuthServiceProvider extends ServiceProvider
{
```

```

    /**
     * Register any application authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        Auth::extend('jwt', function ($app, $name, array $config) {
            // Return an instance of Illuminate\Contracts\Auth\Guard...

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

```

正如上面的代码所示，`extend` 方法传参进去的回调需要返回 `Illuminate\Contracts\Auth\Guard` 的实现，这个接口类有几个方法你需要实现。定制好 Guard 以后，你需要在配置信息中开启使用：

```

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

```

添加自定义用户提供者

如果你没有使用传统的关系型数据库存储用户信息，则需要使用自己的认证用户提供者来扩展 Laravel。我们使用 `Auth facade` 上的 `provider` 方法定义自定义该提供者：

```

<?php

namespace App\Providers;

use Illuminate\Support\Facades\Auth;
use App\Extensions\RiakUserProvider;
use Illuminate\Support\ServiceProvider;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * Register any application authentication / authorization services.
     *
     * @return void
     */
    public function boot()

```

```

{
    $this->registerPolicies();

    Auth::provider('riak', function ($app, array $config) {
        // Return an instance of Illuminate\Contracts\Auth\UserProvider...

        return new RiakUserProvider($app->make('riak.connection'));
    });
}
}

```

通过 `provider` 方法注册用户提供者后，你可以在配置文件 `config/auth.php` 中切换到新的用户提供者。首先，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` 的实现只负责获取 `Illuminate\Contracts\Auth\Authenticatable` 的实现，且不受限于永久保存系统，例如 MySQL, Riak 等等。这两个接口允许 Laravel 认证机制继续作用，而不管用户如何保存或是使用什么样类型的类实现它。

让我们来看看 `Illuminate\Contracts\Auth\UserProvider` contract:

```

<?php

namespace Illuminate\Contracts\Auth;

interface UserProvider {

    public function retrieveById($identifier);
    public function retrieveByToken($identifier, $token);
    public function updateRememberToken(Authenticatable $user, $token);
}

```

```

    public function retrieveByCredentials(array $credentials);
    public function validateCredentials(Authenticatable $user, array $credentials);

}

```

`retrieveById` 函数通常获取一个代表用户的值，例如 MySQL 中自增的 ID。`Authenticatable` 的实现通过 ID 匹配的方法来取出和返回。

`retrieveByToken` 函数借助用户唯一的 `$identifier` 和「记住我」 `$token` 来获取用户。如同之前的方法，`Authenticatable` 的实现应该被返回。

`updateRememberToken` 方法使用新的 `$token` 更新了 `$user` 的 `remember_token` 字段。这个新的令牌可以是全新的令牌（当使用「记住我」尝试登录成功时），或是 `null`（当用户注销时）。

`retrieveByCredentials` 方法获取了从 `Auth::attempt` 方法发送过来的凭证数组（当想要登录时）。这个方法应该要「查找」所使用的持久化存储系统来匹配这些凭证。通常，这个方法会运行一个带着「where」 `$credentials['username']` 条件的查找。这个方法接着需要返回一个 `UserInterface` 的实现。此方法不应该企图做任何密码验证或认证操作。

`validateCredentials` 方法需要比较 `$user` 和 `$credentials` 来认证这个用户。例如，这个方法可能会使用 `Hash::check` 比较 `$user->getAuthPassword()` 字符串及 `$credentials['password']`。这个方法通过返回一个布尔值来验证密码是否正确。

用户认证 Contract

现在我们已经介绍了 `UserProvider` 的每个方法，让我们看一下 `Authenticate contract`。这个提供者需要 `retrieveById` 和 `retrieveByCredentials` 方法来返回这个接口的实现：

```

<?php

namespace Illuminate\Contracts\Auth;

interface Authenticatable {

    public function getAuthIdentifierName();
    public function getAuthIdentifier();
    public function getAuthPassword();
    public function getRememberToken();
    public function setRememberToken($value);
    public function getRememberTokenName();

}

```

这个接口很简单。`getAuthIdentifierName` 方法需要返回「主键名字」。`getAuthIdentifier` 方法需要返回用户的「主键」。在 MySQL 中，这个主键是指自动增加的主键。而 `getAuthPassword` 应该要返回用户哈希后的密码。这个接口允许认证系统和任何用户类运作，不用管你在使用何种 ORM

或存储抽象层。默认情况下，Laravel 的 `app` 文件夹中会包含 `User` 类来实现此接口，所以你可以观察这个类以作为实现的例子。

事件

Laravel 提供了在认证过程中的各种 [事件](#)。你可以在 `EventServiceProvider` 中对这些事件做监听：

```
/**
 * 为应用程序的事件监听器的映射
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Auth\Events\Registered' => [
        'App\Listeners\LogRegisteredUser',
    ],

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

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

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

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

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

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

Laravel 的 API 认证系统 Passport

- 介绍
- 安装
 - 前端快速上手
- 配置
 - 令牌的使用期限
- 发放访问令牌
 - 管理客户端
 - 请求令牌
 - 刷新令牌
- 密码授权令牌
 - 创建密码授权客户端
 - 请求密码授权令牌
 - 请求所有作用域
- 简化授权令牌
- 客户端授权令牌
- 私人访问令牌
 - 创建私人访问令牌的客户端
 - 管理私人访问令牌
- 路由保护
 - 通过中间件
 - 传递访问令牌
- 令牌作用域
 - 定义作用域
 - 给令牌分派作用域
 - 检查作用域
- 使用 JavaScript 接入 API
- 事件
- 测试

介绍

在 Laravel 中，实现基于传统表单的登陆和授权已经非常简单，但是如何满足 API 场景下的授权需求呢？在 API 场景里通常通过令牌来实现用户授权，而非维护请求之间的 Session 状态。现在 Laravel 项目中可以使用 Passport 轻而易举地实现 API 授权过程，通过 Passport 可以在几分钟之内为你的应用程序添加完整的 OAuth2 服务端实现。Passport 基于 [League OAuth2 server](#) 实现，该项目的维护人是 [Alex Bilbie](#)。

{note} 本文档假定你已熟悉 OAuth2。如果你并不了解 OAuth2，阅读之前请先熟悉下 OAuth2 的常用术语和基本特征。

安装

使用 Composer 依赖包管理器安装 Passport：

```
composer require laravel/passport
```

接下来，将 Passport 的服务提供者注册到配置文件 `config/app.php` 的 `providers` 数组中：

```
Laravel\Passport\PassportServiceProvider::class,
```

Passport 使用服务提供者注册内部的数据库迁移脚本目录，所以上一步完成后，你需要更新你的数据库结构。Passport 的迁移脚本会自动创建应用程序需要的客户端数据表和令牌数据表：

```
php artisan migrate
```

{note} 如果你不打算使用 Passport 的默认迁移，你应该在 `AppServiceProvider` 的 `register` 方法中调用 `Passport :: ignoreMigrations` 方法。你可以导出这个默认迁移用 `php artisan vendor:publish --tag=passport-migrations` 命令。

接下来，你需要运行 `passport:install` 命令来创建生成安全访问令牌时用到的加密密钥，同时，这条命令也会创建「私人访问」客户端和「密码授权」客户端：

```
php artisan passport:install
```

上面命令执行后，请将 `Laravel\Passport\HasApiTokens` Trait 添加到 `App\User` 模型中，这个 Trait 会给你的模型提供一些辅助函数，用于检查已认证用户的令牌和使用作用域：

```
<?php

namespace App;

use Laravel\Passport\HasApiTokens;
use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;
}
```

接下来，需要在 `AuthServiceProvider` 的 `boot` 方法中调用 `Passport::routes` 函数。这个函数会注册一些在访问令牌、客户端、私人访问令牌的发放和吊销过程中会用到的必要路由：

```
<?php
```

```

namespace App\Providers;

use Laravel\Passport\Passport;
use Illuminate\Support\Facades\Gate;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        'App\Model' => 'App\Policies\ModelPolicy',
    ];

    /**
     * Register any authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        Passport::routes();
    }
}

```

最后，需要将配置文件 `config/auth.php` 中 `api` 部分的授权保护项（`driver`）改为 `passport`。此调整会让你的应用程序在接收到 API 的授权请求时使用 Passport 的 `TokenGuard` 来处理：

```

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

```

前端快速上手

{note} 如果想要使用 Passport 的 Vue 组件，那么你必须使用 [Vue Javascript 框架](#)，另外这些组件还用到了 Bootstrap CSS 框架。当然你也可以不使用上面的任何工具，但在实现你自己的前端部分时，Passport 的 Vue 组件仍旧有很高的参考价值。

Passport 配备了一些可以让你的用户自行创建客户端和私人访问令牌的 JSON API。所以，你可以自己花费时间来编写一些前端代码来使用这些 API。当然在 Passport 中也已经预制了一些 [Vue 组件](#)，你可以直接使用这些示例代码，也可以基于这些代码实现自己的前端部分。

使用 Artisan 命令 `vendor:publish` 来发布 Passport 的 Vue 组件：

```
php artisan vendor:publish --tag=passport-components
```

已发布的组件将被放置在 `resources/assets/js/components` 目录中，可以在 `resources/assets/js/app.js` 文件中注册这些已发布的组件：

```
Vue.component(
  'passport-clients',
  require('./components/passport/Clients.vue')
);

Vue.component(
  'passport-authorized-clients',
  require('./components/passport/AuthorizedClients.vue')
);

Vue.component(
  'passport-personal-access-tokens',
  require('./components/passport/PersonalAccessTokens.vue')
);
```

这些组件注册后，你可以直接将这些组件直接放入应用程序的模板中，用于创建客户端和私人访问令牌：

```
<passport-clients></passport-clients>
<passport-authorized-clients></passport-authorized-clients>
<passport-personal-access-tokens></passport-personal-access-tokens>
```

配置

令牌的有效期

默认情况下，Passport 发放的访问令牌是永久有效的，不需要刷新。但是如果你想给访问令牌配置一个短一些的有效期，那你就需要用到 `tokensExpireIn` 和 `refreshTokensExpireIn` 方法了，上述两个方法同样需要在 `AuthServiceProvider` 的 `boot` 方法中调用：

```
use Carbon\Carbon;
```

```

/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::tokensExpireIn(Carbon::now()->addDays(15));

    Passport::refreshTokensExpireIn(Carbon::now()->addDays(30));
}

```

发放访问令牌

熟悉 OAuth2 的开发者一定知道， OAuth2 中必不可少的部分就是授权码。在获取授权码时，接入应用会重定向一个用户到你的服务端，用户可以选择允许或拒绝向这个客户端发放访问令牌。

管理客户端

首先，接入应用如果想要与你应用的 API 进行交互，必须先在你的应用程序中注册一个「客户端」。一般来说，这个注册过程需要开发者提供两部分信息：接入应用名称和用户授权后的跳转链接。

命令 `passport:client`

创建客户端最简单的方式是使用 Artisan 命令 `passport:client`，你可以使用此命令创建自己的客户端，用于测试 OAuth2 的功能。在你执行 `client` 命令时，Passport 会提示输入更多关于你的客户端的信息，最终会提供给你生成的客户端的 ID 和密钥：

```
php artisan passport:client
```

JSON API

考虑到你的用户们并没有办法使用 `client` 命令，Passport 同时提供了用户创建客户端的 JSON API。这样你就不用再花时间编码来实现客户端创建、更新和删除的相关控制器逻辑了。

然而，你仍旧需要基于 Passport 的 JSON API 开发一套前端界面，方便你的用户管理他们授权的客户端。下面我们会列出所有用于管理客户端的 API，方便起见，我们使用 [Vue](#) 展示对 API 的 HTTP 请求。

{tip} 如果你不自己重写整个客户端管理的前端界面，可以根据 [前端快速上手](#) 在几分钟内组建一套功能完备的前端界面。

GET /oauth/clients

此接口会返回当前认证用户的所有客户端。主要用途是列出当前用户所有客户端，方便用户修改或删除：

```
this.$http.get('/oauth/clients')
  .then(response => {
    console.log(response.data);
  });

```

POST /oauth/clients

此接口用户创建新的客户端。它需要两部分数据：客户端的名称、客户端的 `redirect` 链接。当用户允许或拒绝授权请求后，用户都会被重定向到这个 `redirect` 链接。

当客户端创建完成后，会生成此客户端的 ID 和密钥，客户端可以使用这两个值从你的应用程序请求访问令牌。此接口会返回新建客户端实例的信息：

```
const data = {
  name: 'Client Name',
  redirect: 'http://example.com/callback'
};

this.$http.post('/oauth/clients', data)
  .then(response => {
    console.log(response.data);
  })
  .catch (response => {
    // List errors on response...
  });

```

PUT /oauth/clients/{client-id}

此接口用于更新客户端信息。它需要两部分数据：客户端的名称和 `redirect` 链接。当用户允许或拒绝授权请求后，用户都会被重定向到这个 `redirect` 链接。此接口会返回被更新客户端实例的信息：

```
const data = {
  name: 'New Client Name',
  redirect: 'http://example.com/callback'
};

this.$http.put('/oauth/clients/' + clientId, data)
  .then(response => {
    console.log(response.data);
  })
  .catch (response => {
    // List errors on response...
  });

```

```
DELETE /oauth/clients/{client-id}
```

此接口用于删除客户端：

```
axios.delete('/oauth/clients/' + clientId)
  .then(response => {
    //
  });

```

请求令牌

授权时的重定向

客户端创建之后，开发者会使用此客户端的 ID 和密钥向你的应用程序请求一个授权码和访问令牌。首先，接入应用会将用户重定向到你应用程序的 `/oauth/authorize` 路由上，示例如下：

```
Route::get('/redirect', function () {
  $query = http_build_query([
    'client_id' => 'client-id',
    'redirect_uri' => 'http://example.com/callback',
    'response_type' => 'code',
    'scope' => '',
  ]);

  return redirect('http://your-app.com/oauth/authorize?'.$query);
});
```

{tip} 注意，路由 `/oauth/authorize` 已经在 `Passport::routes` 方法中定义，所以无需再次定义。

确认授权请求

接收到授权请求时，Passport 会显示默认的授权确认页面，用户可以允许或拒绝本次授权请求。用户确认后会被重定向回接入应用程序请求中指定的 `redirect_uri` 链接。`redirect_uri` 必须和客户端创建时提供的 `redirect` 完全一致。

如果你想自定义授权确认页面，可以使用 Artisan 命令 `vendor:publish` 发布 Passport 的视图文件。发布后的视图文件存放路径为 `resources/views/vendor/passport`：

```
php artisan vendor:publish --tag=passport-views
```

将授权码转换为访问令牌

用户允许授权请求后，用户将会被重定向会接入应用程序，然后接入应用将通过 `POST` 请求向你的应用程序申请访问令牌，此次请求需要携带用户允许授权时产生的授权码。在下面的例子中，我们使用 Guzzle HTTP 库来实现这次 `POST` 请求：

```
Route::get('/callback', function (Request $request) {
    $http = new GuzzleHttp\Client;

    $response = $http->post('http://your-app.com/oauth/token', [
        'form_params' => [
            'grant_type' => 'authorization_code',
            'client_id' => 'client-id',
            'client_secret' => 'client-secret',
            'redirect_uri' => 'http://example.com/callback',
            'code' => $request->code,
        ],
    ]);
    return json_decode((string) $response->getBody(), true);
});
```

接口 `/oauth/token` 的 JSON 相应中会包含 `access_token`、`refresh_token` 和 `expires_in` 属性。`expires_in` 的值即当前访问令牌的有效期（单位：秒）。

{tip} 如上 `/oauth/authorize` 路由，`/oauth/token` 已经在 `Passport::routes` 方法中定义，所以无需再次定义。

刷新令牌

如果你的应用程序发放了短期访问令牌，用户需要刷新访问令牌时，需要提供与访问令牌同时发放的刷新令牌。在下面的例子中，我们使用 Guzzle HTTP 库来刷新令牌：

```
$http = new GuzzleHttp\Client;

$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'refresh_token',
        'refresh_token' => 'the-refresh-token',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'scope' => '',
    ],
]);
return json_decode((string) $response->getBody(), true);
```

接口 `/oauth/token` 会返回一个 JSON 响应，会包含 `access_token`、`refresh_token` 和 `expires_in` 属性。`expires_in` 属性值即当前访问令牌的有效时间（单位：秒）。

密码授权令牌

OAuth2 密码授权机制可以让自有应用基于邮箱地址（用户名）和密码获取访问令牌，自有应用比如你的手机客户端。这样就允许自由应用无需跳转步骤即可通过整个 OAuth2 的授权过程。

创建密码授权客户端

如果想要通过密码授权机制来发布令牌，首先你需要创建一个密码授权客户端。你可以使用带有 `--password` 参数的 `passport:client` 命令。如果你已经运行了 `passport:install` 命令，那无需再单独运行此命令：

```
php artisan passport:client --password
```

请求密码授权令牌

当你创建密码授权客户端后，你可以向 `/oauth/token` 接口发起 `POST` 请求来获取访问令牌，请求时需要带有用户的邮箱地址和密码信息。注意，该接口已经在 `Passport::routes` 方法中定义，所以无需再次手动定义。请求成功后，服务端返回的 JSON 响应数据中会带有 `access_token` 和 `refresh_token` 属性：

```
$http = new GuzzleHttp\Client;

$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'password',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'username' => 'taylor@laravel.com',
        'password' => 'my-password',
        'scope' => '',
    ],
]);

return json_decode((string) $response->getBody(), true);
```

{tip} 注意：访问令牌默认是永久有效的。但是如果需要你可以 [配置你应用程序的访问令牌有效时间](#)。

请求所有作用域

使用密码授权机制时，你可以通过请求作用域 `*` 让你的令牌获取应用程序中定义的所有作用域。在处理使用此令牌发起的请求时，`can` 函数会始终返回 `true`，这种作用域的授权最好只应用在使用 `password` 授权时发放的令牌中：

```
$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'password',
```

```

    'client_id' => 'client-id',
    'client_secret' => 'client-secret',
    'username' => 'taylor@laravel.com',
    'password' => 'my-password',
    'scope' => '*',
],
]);

```

简化授权令牌

简化授权和通过授权码授权相似; 区别是, 不需要通过授权码去获取令牌而是把令牌直接返回客户端. 主要用在无法安全存储证书场景中, 这种授权在 JavaScript 和 移动应用 是最常用的. 开启授权, 在 `AuthServiceProvider` 中调用 `enableImplicitGrant` 方法:

```

/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::enableImplicitGrant();
}

```

调用上面方法开启授权后, 开发者可以通过自己的应用把 client ID 当做参数去请求一个令牌. 在你的应用程序 `/oauth/authorize` 的接口中应该有一个重定向请求像下面这样:

```

Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'token',
        'scope' => '',
    ]);

    return redirect('http://your-app.com/oauth/authorize?'.$query);
});

```

{tip} 记住, 这个 `/oauth/authorize` 接口已经定义在 `Passport::routes` 中. 所以无需再次手动定义.

客户端证书授权令牌

客户端证书授权适用于机器对机器认证，例如，你可以在通过API执行脚本任务中使用此授权。要获取令牌，向 `oauth/token` 接口发出请求：

```
$guzzle = new GuzzleHttp\Client;

$response = $guzzle->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'client_credentials',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'scope' => 'your-scope',
    ],
]);

echo json_decode((string) $response->getBody(), true);
```

私人访问令牌

有些时候你的用户可能想发布一个访问令牌自己使用，又不想经历通常的授权跳转流程，这时候如果能让用户在你的应用程序中自行发放访问令牌，也是一个不错的解决方案。

{note} 私人访问令牌总是永久有效的，`tokensExpireIn` 和 `refreshTokensExpireIn` 方法不会影响它的有效期。

创建私人访问客户端

发布私人访问令牌之前，你需要先创建对应的客户端。你可以使用带 `--personal` 参数的 `passport:client` 命令来创建，如果你已经运行了 `passport:install` 命令，那无需再运行此命令：

```
php artisan passport:client --personal
```

管理私人访问令牌

创建私人访问客户端后，你可以使用 `User` 模型实例上的 `createToken` 方法来为给定用户发布令牌，`createToken` 方法的第一个参数为令牌名称，第二个参数（可选）是 [作用域](#) 列表：

```
$user = App\User::find(1);

// Creating a token without scopes...
$token = $user->createToken('Token Name')->accessToken;

// Creating a token with scopes...
$token = $user->createToken('My Token', ['place-orders'])->accessToken;
```

JSON API

Passport 中也有用来管理私人访问令牌的 JSON API，你可以基于这些 API 开发一套前端操作界面供给用户管理自己的私人访问令牌。下面我们会列出所有管理私人访问令牌的 API。方便起见，我们使用 [Vue](#) 展示对 API 的 HTTP 请求。

{tip} 如果你不想自己重写整个私人访问令牌管理的前端界面，可以根据 [前端快速上手](#) 在几分钟内组建一套功能完备的前端界面。

GET /oauth/scopes

此接口会返回应用程序中定义的所有 [作用域](#)。你可以使用此接口将所有的作用域展示给用户，方便他们授权给需要的私人访问令牌：

```
this.$http.get('/oauth/scopes')
  .then(response => {
    console.log(response.data);
  });

```

GET /oauth/personal-access-tokens

此接口返回当前授权用户创建的所有私人访问令牌。主要用途是列出当前用户所有客户端，方便用户修改或删除：

```
this.$http.get('/oauth/personal-access-tokens')
  .then(response => {
    console.log(response.data);
  });

```

POST /oauth/personal-access-tokens

此接口用来创建私人访问令牌。需要提供两部分数据：令牌的名称（`name`）作用域（`scopes`）：

```
const data = {
  name: 'Token Name',
  scopes: []
};

this.$http.post('/oauth/personal-access-tokens', data)
  .then(response => {
    console.log(response.data.accessToken);
  })
  .catch (response => {
    // List errors on response...
  });

```

DELETE /oauth/personal-access-tokens/{token-id}

此接口用于删除私人访问令牌：

```
this.$http.delete('/oauth/personal-access-tokens/' + tokenId);
```

路由保护

通过中间件

Passport 包含一个 [验证保护机制](#) 可以验证请求中的访问令牌。前面将 `api` 中的保护机制改为为 `passport` 后，你只要给需要验证访问令牌的路由添加 `auth:api` 中间件，该机制将发挥作用：

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

传递访问令牌

接入应用在调用 Passport 保护下的路由时，需要将访问令牌作为 `Bearer` 令牌放在请求头 `Authorization` 中。在下面的例子中，我们使用 Guzzle HTTP 库来实现这次 `POST` 请求：

```
$response = $client->request('GET', '/api/user', [
    'headers' => [
        'Accept' => 'application/json',
        'Authorization' => 'Bearer ' . $accessToken,
    ],
]);
```

令牌作用域

定义作用域

当 API 客户端接入特定用户时，可以通过作用域来限定其访问权限。例如在你编写的电子商务应用中，一些接入应用可以获取订单的发货状态而不能创建订单。换言之，作用域能够让你的用户限制第三方应用的行为，从而保障自身的利益。

你可以使用 `Passport::tokensCan` 方法来定义 API 的作用域，定义代码需要放置在 `AuthServiceProvider` 的 `boot` 方法中。`tokensCan` 方法接受一个包含作用域名称、描述的数组作为参数。作用域描述将会在授权确认页中直接展示给用户，你可以将其定义为任何你需要的内容：

```
use Laravel\Passport\Passport;

Passport::tokensCan([
    'place-orders' => 'Place orders',
    'check-status' => 'Check order status',
```

```
]);
```

给令牌分派作用域

授权码机制

使用授权码机制申请访问令牌时，接入应用可以通过 `scope` 字符串参数指定他们需要的作用域。`scope` 包含多个作用域名称时，名称之间使用空格分隔：

```
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'code',
        'scope' => 'place-orders check-status',
    ]);

    return redirect('http://your-app.com/oauth/authorize?'.$query);
});
```

私人访问令牌

使用 `User` 模型的 `createToken` 方法发放访问令牌时，你可以将需要的作用域数组作为第二个参数传给此方法：

```
$token = $user->createToken('My Token', ['place-orders'])->accessToken;
```

检查作用域

Passport 包含两个检查作用域的中间件，通过访问令牌请求时将会使用这两个中间件来检查是否授予了特定作用域。使用之前，需要将下面的中间件添加到 `app/Http/Kernel.php` 文件的 `$routeMiddleware` 属性中：

```
'scopes' => \Laravel\Passport\Http\Middleware\CheckScopes::class,
'scope' => \Laravel\Passport\Http\Middleware\CheckForAnyScope::class,
```

检查所有作用域

路由可以使用 `scopes` 中间件来检查当前请求是否拥有指定的所有作用域：

```
Route::get('/orders', function () {
    // Access token has both "check-status" and "place-orders" scopes...
})->middleware('scopes:check-status,place-orders');
```

检查任意作用域

路由可以使用 `scope` 中间件来检查当前请求是否拥有指定的任意作用域：

```
Route::get('/orders', function () {
    // Access token has either "check-status" or "place-orders" scope...
})->middleware('scope:check-status,place-orders');
```

检查特定令牌实例的作用域

接入应用使用访问令牌通过你应用程序的验证后，你仍然可以使用当前授权 `user` 实例上的 `tokenCan` 方法来验证此令牌是否拥有指定的作用域：

```
use Illuminate\Http\Request;

Route::get('/orders', function (Request $request) {
    if ($request->user()->tokenCan('place-orders')) {
        //
    }
});
```

使用 JavaScript 接入 API

在构建 API 时，如果能通过 JavaScript 应用接入自己的 API 将会给开发过程带来极大的便利。这样你可以与所用人一样使用你自己的应用程序的 API，同样的 API 可以被你自己的 web 应用、移动应用、第三方应用以及你发布到各个包管理平台的 SDK 共同使用。

通常，在你通过 JavaScript 接入你的 API 时，每次请求你的应用程序时都需要手动传递访问令牌，然而，Passport 其中一个中间件可以帮你做这件事，你需要做的仅仅是将 `CreateFreshApiToken` 中间件添加到你的 `web` 中间件组中：

```
'web' => [
    // Other middleware...
    \Laravel\Passport\Http\Middleware\CreateFreshApiToken::class,
],
```

Passport 的这个中间件将会在你所有的对外请求中添加一个 `laravel_token` cookie，该 cookie 将包含一个加密后的 [JWT](#)，Passport 可以根据此数据判断你 JavaScript 应用的授权状态。至此，你可以无需传递访问令牌直接请求应用程序的 API 了：

```
axios.get('/user')
    .then(response => {
        console.log(response.data);
   });
```

当使用上面方法授权时，Axios 会自动带上 `X-CSRF-TOKEN` 请求头传递。另外，默认的 Laravel JavaScript 也会带上 `X-Requested-With` 请求头：

```
window.axios.defaults.headers.common = {
  'X-Requested-With': 'XMLHttpRequest',
};
```

{note} 如果你用了其他 JavaScript 框架，需要确保每次对外请求都会带有 `X-CSRF-TOKEN` 和 `X-Requested-With` 请求头。

事件

Passport 在访问令牌和刷新令牌时触发事件。你可以通过触发这些事件来修改或删除数据库中的其他访问令牌。你可以在应用程序的 `EventServiceProvider` 中为这些事件附加监听器：

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Laravel\Passport\Events\AccessTokenCreated' => [
        'App\Listeners\RevokeOldTokens',
    ],
    'Laravel\Passport\Events\RefreshTokenCreated' => [
        'App\Listeners\PruneOldTokens',
    ],
];
```

测试

Passport 的 `actingAs` 方法可以用于指定当前认证的用户及其授权范围。`actingAs` 方法第一个参数是一个对象，第二个参数是数组表示申请的授权范围：

```
public function testServerCreation()
{
    Passport::actingAs(
        factory(User::class)->create(),
        ['create-servers']
    );

    $response = $this->post('/api/create-server');

    $response->assertStatus(200);
}
```


Laravel 的用户授权系统

- 简介
- Gates
 - 编写 Gates
 - 通过 Gates 授权动作
- 创建策略
 - 生成策略
 - 注册策略
- 编写策略
 - 策略方法
 - 不使用模型方法
 - 策略过滤器
- 使用策略授权动作
 - 通过用户模型
 - 通过中间件
 - 通过控制器辅助函数
 - 通过 Blade 模板

简介

除了内置提供的 [用户认证](#) 服务外，Laravel 还提供一种更简单的方式来处理用户授权动作。类似用户认证，Laravel 有 2 种主要方式来实现用户授权：gates 和策略。

可以把 gates 和策略类比于路由和控制器。Gates 提供了一个简单、基于闭包的方式来授权认证。策略则和控制器类似，在特定的模型或者资源中通过分组来实现授权认证的逻辑。我们先来看看 gates，然后再看策略。

在你的应用中，不要将 gates 和策略当作相互排斥的方式。大部分应用很可能同时包含 gates 和策略，并且能很好的工作。Gates 大部分应用在模型和资源无关的地方，比如查看管理员的面板。与此相比，策略应该用在特定的模型或者资源中。

Gates

编写 Gates

Gates 是用来决定用户是否授权访问给定的动作的闭包函数，并且典型的做法是在 `App\Providers\AuthServiceProvider` 类中使用 `Gate facade` 定义。Gates 接受一个用户实例作为第一个参数，接受可选参数，比如相关的 Eloquent 模型：

```
/**
 * 注册任意用户认证、用户授权服务。
 */
```

```

 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Gate::define('update-post', function ($user, $post) {
        return $user->id == $post->user_id;
    });
}

```

Gates 也可以使用 `class@method` 形式作为回调字符串，比如控制器

```

/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Gate::define('update-post', 'PostPolicy@update');
}

```

Resource Gates

你还可以使用 `resource` 方法一次性定义多个 `Gate` 能力

```
Gate::resource('posts', 'PostPolicy');
```

这与手动定义以下Gate定义相同：

```

Gate::define('posts.view', 'PostPolicy@view');
Gate::define('posts.create', 'PostPolicy@create');
Gate::define('posts.update', 'PostPolicy@update');
Gate::define('posts.delete', 'PostPolicy@delete');

```

默认情况下，`view`，`create`，`update`，和`delete`能力是被定义过的。你也可以通过将数组作为第三个参数传递给 `resource` 方法来定义其他功能。数组的键定义了该能力的名称，而该值定义了方法名称：

```

Gate::resource('posts', 'PostPolicy', [
    'posts.photo' => 'updatePhoto',
    'posts.image' => 'updateImage',
]);

```

使用 gates 授权动作

使用 gates 来授权动作时，应当使用 `allows` 方法。注意你并不需要传递当前认证通过的用户给 `allows` 方法。Laravel 会自动处理好传入的用户，然后传递给 gate 闭包函数：

```
if (Gate::allows('update-post', $post)) {
    // 当前用户可以更新 post...
}

if (Gate::denies('update-post', $post)) {
    // 当前用户不能更新 post...
}
```

如果需要指定一个特定的用户可以访问某个动作，可以使用 `Gate facade` 中的 `forUser` 方法：

```
if (Gate::forUser($user)->allows('update-post', $post)) {
    // 指定用户可以更新 post...
}

if (Gate::forUser($user)->denies('update-post', $post)) {
    // 指定用户不能更新 post...
}
```

创建策略

生成策略

策略是在特定模型或者资源中组织授权逻辑的类。例如，如果应用是一个博客，会有一个 `Post` 模型和一个相应的 `PostPolicy` 来授权用户动作，比如创建或者更新博客。

可以使用 `make:policy` [artisan 命令](#) 来生成策略。生成的策略将放置在 `app/Policies` 目录。如果你的应用中不存在这个目录，那么 Laravel 会自动创建：

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

`make:policy` 会生成空的策略类。如果希望生成的类包含基本的「CRUD」策略方法，可以在使用命令时指定 `--model` 选项：

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

{tip} 所有授权策略会通过 Laravel [服务容器](#) 解析，意指你可以在授权策略的构造器对任何需要的依赖使用类型提示，它们将会被自动注入。

注册策略

一旦该授权策略存在，需要将它进行注册。`AuthServiceProvider` 包含了一个 `policies` 属性，可将各种模型对应至管理它们的授权策略。注册一个策略将引导 Laravel 在授权动作访问给定模型时使用何种策略：

```
<?php

namespace App\Providers;

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

class AuthServiceProvider extends ServiceProvider
{
    /**
     * 应用的策略映射。
     *
     * @var array
     */
    protected $policies = [
        Post::class => PostPolicy::class,
    ];

    /**
     * 注册任意用户认证、用户授权服务。
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        //
    }
}
```

编写策略

策略方法

一旦授权策略被生成且注册，我们就可以为每个权限的授权增加方法。例如，让我们在 `PostPolicy` 中定义一个 `update` 方法，它会判断指定的 `User` 是否可以「更新」一条 `Post`。

`update` 方法接受 `User` 和 `Post` 实例作为参数，并且应当返回 `true` 或 `false` 来指明用户是否授权更新给定的 `Post`。因此，这个例子中，我们判断用户的 `id` 是否和 `post` 中的 `user_id` 匹配：

```
<?php

namespace App\Policies;

use App\User;
use App\Post;

class PostPolicy
{
    /**
     * 判断给定博客能否被用户更新
     *
     * @param \App\User $user
     * @param \App\Post $post
     * @return bool
     */
    public function update(User $user, Post $post)
    {
        return $user->id === $post->user_id;
    }
}
```

你可以接着在此授权策略定义额外的方法，作为各种权限需要的授权。例如，你可以定义 `view` 或 `delete` 方法来授权 `Post` 的多种行为。可以为自定义策略方法使用自己喜欢的名字。

{tip} 如果在 Artisan 控制台生成策略使用 `--model` 选项，会自动包含 `view`、`create`、`update` 和 `delete` 动作。

不包含模型方法

一些策略方法只接受当前认证通过的用户作为参数，而不用传入授权相关的模型实例。最普遍的应用场景就是授权 `create` 动作。例如，如果正在创建一篇博客，你可能希望检查一下当前用户是否授权创建博客。

当定义一个不需要传入模型实例的策略方法时，比如 `create` 方法，你需要定义这个方法只接受已授权的用户作为参数：

```
/**
 * 判断给定用户是否可以创建博客。
 *
 * @param \App\User $user
 * @return bool
 */
public function create(User $user)
{
    //
}
```

策略过滤器

对特定用户，你可能希望通过指定的策略授权所有动作。要达到这个目的，可以在策略中定义一个 `before` 方法。`before` 方法会在策略中其他所有方法之前执行，这样提供了一种方式来授权动作而不是指定的策略方法来执行判断。这个功能最常见的场景是授权应用的管理员可以访问所有动作：

```
public function before($user, $ability)
{
    if ($user->isSuperAdmin()) {
        return true;
    }
}
```

如果你想拒绝用户所有的授权，你应该在 `before` 方法中返回 `false`。如果返回的是 `null`，则通过其他的策略方法来决定授权与否。

使用策略授权动作

通过用户模型

Laravel 应用内置的 `User` 模型包含 2 个有用的方法来授权动作：`can` 和 `cant`。`can` 方法指定需要授权的动作和相关的模型。例如，判定一个用户是否授权更新给定的 `Post` 模型：

```
if ($user->can('update', $post)) {
    //
}
```

如果给定模型的 [策略已被注册](#)，`can` 方法会自动调用核实的策略方法并且返回 `boolean` 值。如果没有策略注册到这个模型，`can` 方法会尝试调用和动作名相匹配的基于闭包的 Gate。

不需要指定模型的动作

一些动作，比如 `create`，并不需要指定模型实例。在这种情况下，可传递一个类名给 `can` 方法。当授权动作时，这个类名将被用来判断使用哪个策略：

```
use App\Post;

if ($user->can('create', Post::class)) {
    // 执行相关策略中的「create」方法...
}
```

通过中间件

Laravel 包含一个可以在请求到达路由或控制器之前就进行动作授权的中间件。默认，`Illuminate\Auth\Middleware\Authorize` 中间件被指定到 `App\Http\Kernel` 类中 `can` 键上。我们用一个授权用户更新博客的例子来讲解 `can` 中间件的使用：

```
use App\Post;

Route::put('/post/{post}', function (Post $post) {
    // 当前用户可以更新博客...
})->middleware('can:update,post');
```

在这个例子中，我们传递给 `can` 中间件 2 个参数。第一个是需要授权的动作的名称，第二个是我们希望传递给策略方法的路由参数。这里因为使用了 [隐式模型绑定](#)，一个 `Post` 会被传递给策略方法。如果用户不被授权访问指定的动作，这个中间件会生成带有 `403` 状态码的 HTTP 响应。

不需要指定模型的动作

同样的，一些动作，比如 `create`，并不需要指定模型实例。在这种情况下，可传递一个类名给中间件。当授权动作时，这个类名将被用来判断使用哪个策略：

```
Route::post('/post', function () {
    // 当前用户可以创建博客...
})->middleware('can:create,App\Post');
```

通过控制器辅助函数

除了在 `User` 模型中提供辅助方法外，Laravel 也为所有继承了 `App\Http\Controllers\Controller` 基类的控制器提供了一个有用的 `authorize` 方法。和 `can` 方法类似，这个方法接收需要授权的动作和相关的模型作为参数。如果动作不被授权，`authorize` 方法会抛出 `Illuminate\Auth\Access\AuthorizationException` 异常，然后被 Laravel 默认的异常处理器转化为带有 `403` 状态码的 HTTP 响应：

```
<?php

namespace App\Http\Controllers;

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

class PostController extends Controller
{
    /**
     * 更新给定博客
     *
     * @param Request $request
     * @param Post $post

```

```

    * @return Response
    */
public function update(Request $request, Post $post)
{
    $this->authorize('update', $post);

    // 当前用户可以更新博客...
}
}

```

不需要指定模型的动作

和之前讨论的一样，一些动作，比如 `create`，并不需要指定模型实例。在这种情况下，可传递一个类名给 `authorize` 方法。当授权动作时，这个类名将被用来判断使用哪个策略：

```

/**
 * 新建博客
 *
 * @param Request $request
 * @return Response
 */
public function create(Request $request)
{
    $this->authorize('create', Post::class);

    // 当前用户可以新建博客...
}

```

通过 Blade 模板

当编写 Blade 模板时，你可能希望页面的指定部分只展示给允许授权访问给定动作的用户。例如，你可能希望只展示更新表单给有权更新博客的用户。这种情况下，你可以直接使用 `@can` 和 `@cannot` 指令。

```

@can('update', $post)
    <!-- 当前用户可以更新博客 -->
@elsecan('create', $post)
    <!-- 当前用户可以新建博客 -->
@endcan

@cannot('update', $post)
    <!-- 当前用户不可以更新博客 -->
@elsecannot('create', $post)
    <!-- 当前用户不可以新建博客 -->
@endcannot

```

这些指令在编写 `@if` 和 `@unless` 时提供了方便的缩写。`@can` 和 `@cannot` 各自转化为如下声明：

```
@if (Auth::user()->can('update', $post))
    <!-- 当前用户可以更新博客 -->
@endif

@unless (Auth::user()->can('update', $post))
    <!-- 当前用户不可以更新博客 -->
@endunless
```

不需要指定模型的动作

和大部分其他的授权方法类似，当动作不需要模型实例时，你可以传递一个类名给 `@can` 和 `@cannot` 指令：

```
@can('create', Post::class)
    <!-- 当前用户可以新建博客 -->
@endcan

@cannot('create', Post::class)
    <!-- 当前用户不可以新建博客 -->
@endcannot
```

Laravel 的加密解密机制

- [介绍](#)
- [设置](#)
- [基本用法](#)

介绍

Laravel 是利用 OpenSSL 去提供 AES-256 和 AES-128 的加密。强烈建议您使用 Laravel 自己的加密机制，而不是尝试自己的「自制」加密算法。Laravel 所有加密之后的结果都会使用消息认证码 (MAC) 去签署，所以一旦被加密就无法再改变。

设置

在使用 Laravel 加密之前，你必须先设置 `config/app.php` 配置文件中的 `key` 选项。由于 Artisan 控制台会使用 PHP 的安全机制为你随机生成 `key`，你可以直接使用 `php artisan key:generate` 命令去生成 `key`。如果没有适当地设置这个值，所有被 Laravel 加密的值都将是不安全的。

基本用法

加密一个值

你可以借助 `encrypt` 辅助函数来加密一个值。这些值都会使用 OpenSSL 与 `AES-256-CBC` 来进行加密。此外，所有加密过后的值都会被签署文件消息验证码 (MAC)，以检测加密字符串是否被篡改过：

```
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller
{
    /**
     * 存储用户保密信息
     *
     * @param Request $request
     * @param int $id
     * @return Response
     */
    public function storeSecret(Request $request, $id)
```

```

{
    $user = User::findOrFail($id);

    $user->fill([
        'secret' => encrypt($request->secret)
    ])->save();
}
}

```

不进行序列化的加密解密方法

加密值在加密期间通过 `serialize` 传递，这也就允许对对象和数组进行加密。由此，非PHP客户端接收到加密值将需要 `unserialize` 数据。如果您希望在不进行序列化的情况下加密和解密值，可以使用 `Crypt facade` 的 `encryptString` 和 `decryptString` 方法：

```

use Illuminate\Support\Facades\Crypt;

$encrypted = Crypt::encryptString('Hello world.');

$decrypted = Crypt::decryptString($encrypted);

```

解密一个值

你可以借助 `decrypt` 辅助函数来解密一个值。如果值不能被正确解密，例如当 MAC 无效时，将抛出 `Illuminate\Contracts\Encryption\DecryptException` 异常：

```

use Illuminate\Contracts\Encryption\DecryptException;

try {
    $decrypted = decrypt($encryptedValue);
} catch (DecryptException $e) {
    //
}

```

Laravel 的哈希加密

- 简介
- 基本用法

简介

Laravel `Hash Facade` 提供安全的 Bcrypt 哈希保存用户密码。如果应用程序中使用了 Laravel 内置的 `LoginController` 和 `RegisterController` 类，它们将自动使用 Bcrypt 进行注册和身份验证。

{tip} Bcrypt 是哈希密码的理想选择，因为它的「加密系数」可以任意调整，这意味着生成哈希所需的时间可以随着硬件功率的增加而增加。

基本用法

你可以通过调用 `Hash Facade` 的 `make` 方法来填写密码：

```
<?php

namespace App\Http\Controllers;

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

class UpdatePasswordController extends Controller
{
    /**
     * 更新用户密码
     *
     * @param Request $request
     * @return Response
     */
    public function update(Request $request)
    {
        // Validate the new password length...

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

`make` 方法还能使用 `rounds` 选项来管理 bcrypt 哈希算法的加密系数。然而，大多数应用程序还是能接受默认值的：

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

根据哈希值验证密码

`check` 方法可以验证给定的纯文本字符串对应于给定的散列。如果使用 [Laravel 内置的 LoginController](#)，则不需要直接使用该方法，因为该控制器会自动调用此方法：

```
if (Hash::check('plain-text', $hashedPassword)) {  
    // 密码对比...  
}
```

检查密码是否需要重新加密

`needsRehash` 函数允许你检查已加密的密码所使用的加密系数是否被修改：

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

Laravel 的密码重设功能

- [简介](#)
- [数据库注意事项](#)
- [路由](#)
- [视图](#)
- [重置密码后](#)
- [密码自定义](#)

简介

{tip} 想要快速上手此功能？首先在 Laravel 应用下运行 `php artisan make:auth` 命令，然后使用浏览器打开 `http://your-app.dev/register`，或者任意一个在应用中分配的 URL。这个命令将会生成包括密码重置在内的整个认证系统。

大部分的 web 应用都为用户提供重置密码的功能。Laravel 提供了一种非常方便的方法用于发送密码重置邮件来完成密码重置，而不需要在每个应用中重新实现。

{note} 在使用 Laravel 密码重置功能之前，你必须 `use Illuminate\Notifications\Notifiable trait`。

数据库注意事项

开始之前，请先确认你的 `App\User` 模型是否实现了

`Illuminate\Contracts\Auth\CanResetPassword` 协议。当然，在 Laravel 框架中 `App\User` 模型已经实现了这个接口，并使用 `Illuminate\Auth\Passwords\CanResetPassword trait` 来实现该接口包含的方法。

生成重置令牌表迁移

接下来，我们必须先创建用来存储密码重置令牌的数据表。由于 Laravel 已经自带了这张表的迁移，就存放在 `database/migrations` 目录，因此，你仅仅只需要运行下面的命令即可完成数据表的创建：

```
php artisan migrate
```

路由

Laravel 在 `Auth\ForgotPasswordController` 和 `Auth\ResetPasswordController` 两个类中包含了电子邮件密码重置链接和重置用户密码的必要逻辑。你只需要使用 Artisan 命令 `make:auth` 命令即可生成密码重置所需要的路由：

```
php artisan make:auth
```

视图

和路由一样Laravel 在执行 `make:auth` 命令时同时会在 `resources/views/auth/passwords` 生成密码重置必要视图文件。当然，你可以随意对它们进行修改。

重置密码后

一旦您定义了重置用户密码的路由和视图后，您可以在浏览器中访问 `/password/reset` 。框架自带的 `ForgotPasswordController` 已经包含了发送密码重置链接邮件的逻辑，`ResetPasswordController` 包含了重置用户密码的逻辑。

在密码重置之后，用户将会自动登录并重定向到 `/home` 。你可以定义 `ResetPasswordController` 控制器的 `redirectTo` 属性来定制密码重置成功后的自定义跳转链接：

```
protected $redirectTo = '/dashboard';
```

{note} 默认情况下，密码重置令牌在一小时内有效。你可以通过修改 `config/auth.php` 的 `expire` 选项来修改此项配置。

自定义

自定义认证 Guard

在配置文件 `auth.php` 中，你可以配置多个「guards」参数，可以用来多用户表的独立认证。你可以通过重写 `ResetPasswordController` 控制器的 `guard` 方法来实现自定义认证 `guard`，同时别忘了，这个方法需要返回一个 `guard` 实例：

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

protected function guard()
{
    return Auth::guard('guard-name');
}
```

自定义密码 Broker

在配置文件 `auth.php` 中，可以配置多个密码「brokers」，它可用于多个用户表重置密码。你可以通过重写 `ForgotPasswordController` 和 `ResetPasswordController` 控制器的 `broker` 方法来实现你的自定义 `broker`：

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

```
/**  
 * 获取在密码重置期间使用的 broker  
 *  
 * @return PasswordBroker  
 */  
protected function broker()  
{  
    return Password::broker('name');  
}
```

自定义重置邮箱

你可以方便的修改通知类用以给用户发送密码重置链接。在开始之前，重写 `User` 模型中 `sendPasswordResetNotification` 方法。在这个方法中，你可以使用你选择的任意通知类发送通知。这个方法的第一个参数为密码重置令牌 `$token`：

```
/**  
 * 发送密码重置通知  
 *  
 * @param string $token  
 * @return void  
 */  
public function sendPasswordResetNotification($token)  
{  
    $this->notify(new ResetPasswordNotification($token));  
}
```

8 综合话题

Laravel 的 Artisan 命令行工具

- [介绍](#)
- [编写命令](#)
 - [生成命令](#)
 - [命令结构](#)
 - [闭包命令](#)
- [定义预期的输入](#)
 - [参数](#)
 - [选项](#)
 - [输入数组](#)
 - [输入说明](#)
- [I/O 命令](#)
 - [获取输入](#)
 - [输入提示](#)
 - [编写输出](#)
- [注册命令](#)
- [程序内部调用命令](#)
 - [命令中调用其它命令](#)

介绍

`Artisan` 是 Laravel 的命令行接口的名称，它提供了许多实用的命令来帮助你开发 Laravel 应用，要查看所有的 `Artisan` 命令列表，可以使用 `list` 命令：

```
php artisan list
```

每个命令也包含了「帮助」界面，它会显示并概述命令可使的参数及选项。只需要在命令前面加上 `help` 即可显示命令帮助界面：

```
php artisan help migrate
```

Laravel REPL

所有的 Laravel 应用都包括 Tinker，一个基于 `PsySH` 开发的 REPL 包。Tinker 让你可以在命令行中与你整个的 Laravel 应用进行交互，包括 Eloquent ORM，任务，事件等等。运行 `tinker` 命令进入 Tinker 环境：

```
php artisan tinker
```

编写命令

除了 `Artisan` 提供的命令之外，还可以创建自定义命令。自定义命令默认存储在 `app\Console\Commands` 目录，当然，你也可以修改 `composer.json` 配置来指定你想要存放的目录。

生成命令

要创建一个新的命令，可以使用 `make:command` 命令。这个命令会创建一个命令类并存放在 `app\Console\Commands` 目录。不必担心不存在这个目录，运行 `make:command` 命令时会首先创建这个目录。生成的命令将会包括所有默认存在的属性和方法：

```
php artisan make:command SendEmails
```

接下来，你需要在 Artisan CLI 里执行之前[注册命令](#)。

命令结构

命令生成以后，应先填写类的 `signature` 和 `description` 属性，之后可以在使用 `list` 命令是显示出来。执行命令是调用 `handle` 方法，可以把你的命令逻辑放到这个方法中。

{tip} 大部分的代码复用，保持你的代码轻量并让它们延迟到应用服务中完成任务是个不错的实践。在下面这个例子中，我们注入了一个服务类去执行发送邮件的繁重任务。

让我们看这个简单的命令例子，`Command` 类构造器允许注入需要的依赖，Laravel 的[服务容器](#) 将会自动注入构造函数中所有带类型约束的依赖：

```
<?php

namespace App\Console\Commands;

use App\User;
use App\DripEmailer;
use Illuminate\Console\Command;

class SendEmails extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'email:send {user}';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Send drip e-mails to a user';
```

```

/**
 * The drip e-mail service.
 *
 * @var DripEmailer
 */
protected $drip;

/**
 * Create a new command instance.
 *
 * @param DripEmailer $drip
 * @return void
 */
public function __construct(DripEmailer $drip)
{
    parent::__construct();

    $this->drip = $drip;
}

/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
    $this->drip->send(User::find($this->argument('user')));
}
}

```

闭包命令

闭包命令提供一个替代定义命令方法的类。同样的路由闭包是控制器的一种替代方法，这种命令闭包可以替换命令类。使用 `app\Console\Kernel.php` 文件的 `commands` 方法，需要 Laravel 在 `routes/console.php` 注册：

```

/**
 * Register the Closure based commands for the application.
 *
 * @return void
 */
protected function commands()
{
    require base_path('routes/console.php');
}

```

虽然这个文件没有定义 HTTP 路由，它定义了基于控制台的入口点（路由）到你的应用中，在这个文件中，你可以使用 `Artisan::command` 方法定义所有基于路由的闭包，`command` 方法接收两个参数：[命令签名](#) 和一个接收命令参数和选项的闭包：

```
Artisan::command('build {project}', function ($project) {
    $this->info("Building {$project}!");
});
```

闭包绑定下面的命令实例，因此你可以访问所有的辅助方法，您也可以访问一个完整的命令类。

类型提示依赖

除了接收命令的参数和选项外，命令闭包也可以使用类型提示来指定 [服务容器](#) 之外的额外依赖：

```
use App\User;
use App\DripEmailer;

Artisan::command('email:send {user}', function (DripEmailer $drip, $user) {
    $drip->send(User::find($user));
});
```

闭包命令描述

当定义一个基于命令的闭包时，你可以使用 `describe` 方法来为命令添加描述。这个描述将会在你执行 `php artisan list` 或 `php artisan help` 命令时显示：

```
Artisan::command('build {project}', function ($project) {
    $this->info("Building {$project}!");
})->describe('Build the project');
```

定义预期的输入

在你编写控制台命令时，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，你可以在命令里使用 `signature` 属性。`signature` 属性通过一个类似路由风格的语法让用户为命令定义名称，参数和选项。

参数

所有用户提供的参数及选项都包在大括号中。如以下例子，此命令会定义一个 必须 的参数：`user`：

```
/**
 * The name and signature of the console command.
 *
 * @var string
```

```
 */
protected $signature = 'email:send {user}';
```

您也可以创建可选参数，并定义参数的默认值：

```
// Optional argument...
email:send {user?}

// Optional argument with default value...
email:send {user=foo}
```

选项

选项，和参数一样，也是用户输入的一种格式，不过当使用选项时，需要在命令前加两个连字符（`- -`）的前缀，有两种类型的选项：接收一个值和不接受值。选项不接收一个值作为布尔值的 `switch`。让我们看一个选项的例子：

```
/**
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'email:send {user} {--queue}';
```

在这个例子中，当调用 `Artisan` 命令时，`--queue` 这个选项可以被明确的指定。如果 `--queue` 被当成输入时，这个选项的值为 `true`，否则这个值为 `false`：

```
php artisan email:send 1 --queue
```

有值的选项

接下来，我们看下有值的选项。如果用户必须为选项指定一个值，会在选项的后面加一个 `=` 的后缀：

```
/**
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'email:send {user} {--queue=}';
```

在这个例子中，用户可以像下面这个例子传递一个值：

```
php artisan email:send 1 --queue=default
```

您可以通过指定选项名称后的默认值将默认值分配给选项。如果用户没有输入一个值，将会采用默认的值：

```
email:send {user} {--queue=default}
```

选项快捷键

当定义一个定义选项时，可以分配一个快捷键。你可以在选项前使用一个 `|` 分隔符将简写和完整选项名分开：

```
email:send {user} {--Q|queue}
```

数组输入

如果你想使用数组输入方式定义参数或选项，你可以使用 `*` 符号。首先，我们先看一个数组输入的实例：

```
email:send {user*}
```

调用此方法时，`user` 参数通过命令行输入。例如，下面这个命令将会为 `user` 设置 `['foo', 'bar']`：

```
php artisan email:send foo bar
```

在定义一个使用数组输入时，每个输入选项值的命令都要在选项名称前加前缀：

```
email:send {user} {--id=*}

php artisan email:send --id=1 --id=2
```

输入描述

您可以通过在使用一个冒号分离参数来分配输入参数和选项的说明。如果你需要一个大的额外的空间来定义你的命令，可以多行定义你的扩展：

```
/**
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'email:send
    {user : The ID of the user}
    {--queue= : Whether the job should be queued}';
```

I/O 命令

获取输入

在命令执行的时，你可以像下面这样使用 `argument` 和 `option` 方法获取参数和选项：

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
    $userId = $this->argument('user');

    //
}
```

如果您需要获取所有参数作为一个 `array`，调用 `arguments` 方法：

```
$arguments = $this->arguments();
```

选项可以像参数一样使用 `option` 方法检索， 获取所有的选项作为一个 `array`，调用 `options` 方法：

```
// Retrieve a specific option...
$queueName = $this->option('queue');

// Retrieve all options...
$options = $this->options();
```

如果参数或选项不存在，将会返回 `null`。

交互式输入

除了显示输出，您还可以要求用户在您的命令执行过程中提供输入。`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：

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
    $name = $this->ask('What is your name?');
```

```
}
```

`secret` 方法和 `ask` 方法类似，但是用户输入在终端中是不可以见的，这个方法在需要用户输入像密码这样的敏感信息是很有用的：

```
$password = $this->secret('What is the password?');
```

请求确认

如果你要用户提供确认信息，你可以使用 `confirm` 方法，默认情况下，该方法返回 `false`，当然，如果你输入 `y` 这个方法将会返回 `true`。

```
if ($this->confirm('Do you wish to continue?')) {
    //
}
```

自动完成

`anticipate` 方法可用于为可能的选项提供自动完成功能，用户仍然可以选择，忽略这个提示：

```
$name = $this->anticipate('What is your name?', ['Taylor', 'Dayle']);
```

多重选择

如果你要给用户一组预设选择，可以使用 `choice` 方法，你可以设置默认选项当用户没有选择时：

```
$name = $this->choice('What is your name?', ['Taylor', 'Dayle'], $default);
```

编写输出

使用 `line`、`info`、`comment`、`question` 和 `error` 方法来发送输出到终端。每个方法都有适当的 ANSI 颜色来作为他们的标识。例如，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：

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
    $this->info('Display this on the screen');
}
```

显示错误信息，使用 `error` 方法。错误信息显示红色：

```
$this->error('Something went wrong!');
```

使用 `line` 方法可以像平常一样，没有颜色输出。

```
$this->line('Display this on the screen');
```

数据表布局

`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();

$bar = $this->output->createProgressBar(count($users));

foreach ($users as $user) {
    $this->performTask($user);

    $bar->advance();
}

$bar->finish();
```

更多信息请查阅[Symfony Progress Bar 组件的文档](#).

注册命令

命令编写完成后，需要注册 Artisan 后才能使用。注册文件为 `app\Console\Kernel.php`。在这个文件中，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 启动时，所有罗列在这个属性的命令，都会被 [服务容器](#) 解析并向 Artisan 注册：

```
protected $commands = [
    Commands\SendEmails::class
];
```

程序内部调用命令

有时候你可能希望在 CLI 之外执行 Artisan 命令，例如，你可能希望在路由或控制器中触发 Artisan 命令，你可以使用 `Artisan facade` 上的 `call` 方法来完成。`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'
    ]);

    //
});

});
```

在 `Artisan facade` 使用 `queue` 方法，可以将 Artisan 命令给后台的 [队列服务器](#) 运行，在这之前，确保您已配置了您的队列，并正在运行队列：

```
Route::get('/foo', function () {
    Artisan::queue('email:send', [
        'user' => 1, '--queue' => 'default'
    ]);

    //
});

});
```

如果需要指定非接收字符串选项的值，如 `migrate:refresh` 命令的 `--force` 值，你可以传递一个 `true` 或者 `false`：

```
$exitCode = Artisan::call('migrate:refresh', [
    '--force' => true,
]);
```

命令中调用其它命令

有时候你希望从一个已存在的 Artisan 命令中调用其它命令。你可以使用 `call` 方法，`call` 方法接受命令名称和命令参数的数组：

```
/**
 * Execute the console command.
 *
 * @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` 命令。`callSilent` 和 `call` 方法用法一样：

```
$this->callSilent('email:send', [
    'user' => 1, '--queue' => 'default'
]);
```

Laravel 的事件广播系统

- [简介](#)
 - [配置](#)
 - [对驱动器的要求](#)
- [概念综述](#)
 - [使用示例程序](#)
- [定义广播事件](#)
 - [广播名称](#)
 - [广播数据](#)
 - [广播队列](#)
- [频道授权](#)
 - [定义授权路由](#)
 - [定义授权回调](#)
- [对事件进行广播](#)
 - [只广播给他人](#)
- [接收广播](#)
 - [安装 Laravel Echo](#)
 - [对事件进行监听](#)
 - [退出频道](#)
 - [命名空间](#)
- [Presence 频道](#)
 - [授权 Presence 频道](#)
 - [加入 Presence 频道](#)
 - [广播到 Presence 频道](#)
- [客户端事件](#)
- [消息通知](#)

简介

在现代的 web 应用程序中，WebSockets 被用来实现需要实时、即时更新的接口。当服务器上的数据被更新后，更新信息将通过 WebSocket 连接发送到客户端等待处理。相比于不停地轮询应用程序，WebSocket 是一种更加可靠和高效的选择。

为了帮助你建立这类应用，Laravel 将通过 WebSocket 连接来使「广播」[事件](#) 变得更加轻松。广播事件允许你在服务端代码和客户端 JavaScript 应用之间共享相同的事件名。

{tip} 在深入了解事件广播之前，请确认你已阅读所有关于 [Laravel 事件和侦听器](#) 的文档。

配置

所有关于事件广播的配置都被保存在 `config/broadcasting.php` 文件中。Laravel 自带了几个广播驱动器：[Pusher](#)、[Redis](#)，和一个用于本地开发与调试的 `log` 驱动器。另外，还有一个 `null` 驱动器可以让你完全关闭广播功能。每一个驱动的示例配置都可以在 `config/broadcasting.php` 文件中被找到。

广播服务提供者

在对事件进行广播之前，你必须先注册 `App\Providers\BroadcastServiceProvider`。对于一个全新安装的 Laravel 应用程序，你只需在 `config/app.php` 配置文件的 `providers` 数组中取消对该提供者的注释即可。该提供者将允许你注册广播授权路由和回调。

CSRF 令牌

[Laravel Echo](#) 会需要访问当前会话的 CSRF 令牌。如果可用，Echo 会从 `Laravel.csrfToken` JavaScript 对象中获取该令牌。如果你运行了 `make:auth` Artisan 命令，该对象会在 `resources/views/layouts/app.blade.php` 布局文件中被定义。如果你未使用该布局文件，可以在应用程序的 `head` HTML 元素中定义一个 `meta` 标签：

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

对驱动器的要求

Pusher

如果你使用 [Pusher](#) 对事件进行广播，请用 Composer 包管理器来安装 Pusher PHP SDK：

```
composer require pusher/pusher-php-server
```

然后，你需要在 `config/broadcasting.php` 配置文件中填写你的 Pusher 证书。该文件中已经包含了一个 Pusher 示例配置，你只需指定 Pusher key、secret 和 application ID 即可。`config/broadcasting.php` 中的 `pusher` 配置项同时也允许你指定 Pusher 支持的 `options`，例如 `cluster`：

```
'options' => [
    'cluster' => 'eu',
    'encrypted' => true
],
```

当把 Pusher 与 [Laravel Echo](#) 一起使用时，你应该在实例化 Echo 对象时指定 `broadcaster` 为 `pusher`：

```
import Echo from "laravel-echo"

window.Echo = new Echo({
    broadcaster: 'pusher',
```

```
    key: 'your-pusher-key'
});
```

Redis

如果你使用 Redis 广播器， 请安装 Predis 库：

```
composer require predis/predis
```

Redis 广播器会使用 Redis 的「生产者/消费者」特性来广播消息；尽管如此，你仍需将它与 WebSocket 服务器一起使用。WebSocket 服务器会从 Redis 接收消息，然后再将消息广播到你的 WebSocket 频道上去。

当 Redis 广播器发布一个事件时，该事件会被发布到它指定的频道上去，传输的数据是一个采用 JSON 编码的字符串。该字符串包含了事件名、`data` 数据和生成该事件套接字 ID 的用户（如果可用的话）。

Socket.IO

如果你想把 Redis 广播器和 Socket.IO 服务器一起使用，你需要将 Socket.IO 客户端库文件包含到应用程序的 `head` HTML 元素中。当 Socket.IO 服务启动的时候，它会自动把 Socket.IO JavaScript 客户端库暴露在一个标准的 URL 中。例如，如果你的应用和 Socket.IO 服务器运行在同域名下，你可以像下面这样来访问你的 Socket.IO JavaScript 客户端库：

```
<script src="//{{ Request::getHost() }}:6001/socket.io/socket.io.js"></script>
```

接着，你需要在实例化 Echo 时指定 `socket.io` 连接器和 `host` 。

```
import Echo from "laravel-echo"

window.Echo = new Echo({
    broadcaster: 'socket.io',
    host: window.location.hostname + ':6001'
});
```

最后，你需要运行一个与 Laravel 兼容的 Socket.IO 服务器。Laravel 官方并没有实现 Socket.IO 服务器；不过，可以选择一个由社区驱动维护的项目 [tlaivedure/laravel-echo-server](#)，目前托管在 GitHub。

对队列的要求

在开始广播事件之前，你还需要配置和运行 [队列侦听器](#)。所有的事件广播都是通过队列任务来完成的，因此应用程序的响应时间不会受到明显影响。

概念综述

Laravel 的事件广播允许你使用基于驱动的 WebSockets 将服务端的 Laravel 事件广播到客户端的 JavaScript 应用程序。当前的 Laravel 自带了 Pusher 和 Redis 驱动。通过使用 Laravel Echo 的 Javascript 包，我们可以很方便地在客户端消费事件。

事件通过「频道」来广播，这些频道可以被指定为公开的或私有的。任何访客都可以订阅一个不需要认证和授权的公开频道；然而，如果想订阅一个私有频道，那么该用户必须通过认证，并获得该频道的授权。

使用示例程序

让我们先用一个电子商务网站作为例子来概览一下事件广播。我们不会讨论如何配置 Pusher 和 Laravel Echo 的细节，因为这些会在本文档的其他章节被详细介绍。

在我们的应用程序中，让我们假设有一个允许用户查看订单配送状态的页面。有一个 `ShippingStatusUpdated` 事件会在配送状态更新时被触发：

```
event(new ShippingStatusUpdated($update));
```

ShouldBroadcast 接口

当用户在查看自己的订单时，我们不希望他们必须通过刷新页面才能看到状态更新。我们希望一旦有更新时就主动将更新信息广播到客户端。所以，我们必须让 `ShippingStatusUpdated` 事件实现 `ShouldBroadcast` 接口。这会让 Laravel 在事件被触发时广播该事件：

```
<?php

namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class ShippingStatusUpdated implements ShouldBroadcast
{
    /**
     * Information about the shipping status update.
     *
     * @var string
     */
    public $update;
}
```

`ShouldBroadcast` 接口要求事件实现 `broadcastOn` 方法。该方法负责指定事件被广播到哪些频道。在通过 Artisan 命令生成的事件类中，一个空的 `broadcastOn` 方法已经被预定义好了，所以我们要做的仅仅是指定频道。我们希望只有订单的创建者能够看到状态更新，所以我们要把该事件广播到与这个订单绑定的私有频道上去：

```
/**
 * Get the channels the event should broadcast on.
 *
 * @return array
 */
public function broadcastOn()
{
    return new PrivateChannel('order.'.$this->update->order_id);
}
```

频道授权

记住，用户只有在被授权后才能监听私有频道。我们可以在 `routes/channels.php` 文件中定义频道的授权规则。在本例中，我们需要对试图监听私有 `order.1` 频道的所有用户进行验证，确保只有订单的创建者才能进行监听：

```
Broadcast::channel('order.{orderId}', function ($user, $orderId) {
    return $user->id === Order::findOrNew($orderId)->user_id;
});
```

`channel` 方法接收两个参数：频道名称和一个回调函数，该回调通过返回 `true` 或 `false` 来表示用户是否被授权监听该频道。

所有的授权回调接收当前被认证的用户作为第一个参数，任何额外的通配符参数作为后续参数。在本例中，我们使用 `{orderId}` 占位符来表示频道名称的「ID」部分是通配符。

对事件广播进行监听

接下来，就只剩下在 JavaScript 应用程序中监听事件了。我们可以使用 Laravel Echo 来实现。首先，使用 `private` 方法来订阅私有频道。然后，使用 `listen` 方法来监听 `ShippingStatusUpdated` 事件。默认情况下，事件的所有公有属性会被包括在广播事件中：

```
Echo.private(`order.${orderId}`)
    .listen('ShippingStatusUpdated', (e) => {
        console.log(e.update);
   });
```

定义广播事件

要告知 Laravel 一个给定的事件是广播类型，只需在事件类中实现 `Illuminate\Contracts\Broadcasting\ShouldBroadcast` 接口即可。该接口已经被导入到所有由框架生成的事件类中，所以你可以很方便地将它添加到你自己的事件中。

`ShouldBroadcast` 接口要求你实现一个方法：`broadcastOn`。`broadcastOn` 方法返回一个频道或一个频道数组，事件会被广播到这些频道。频道必须是 `Channel`、`PrivateChannel` 或 `PresenceChannel` 的实例。`Channel` 实例表示任何用户都可以订阅的公开频道，而 `PrivateChannels` 和 `PresenceChannels` 则表示需要 [频道授权](#) 的私有频道：

```
<?php

namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class ServerCreated implements ShouldBroadcast
{
    use SerializesModels;

    public $user;

    /**
     * 创建一个新的事件实例
     *
     * @return void
     */
    public function __construct(User $user)
    {
        $this->user = $user;
    }

    /**
     * 指定事件在哪些频道上进行广播
     *
     * @return Channel|array
     */
    public function broadcastOn()
    {
        return new PrivateChannel('user.'.$this->user->id);
    }
}
```

然后，你只需要像你平时那样 [触发事件](#)。一旦事件被触发，一个 [队列任务](#) 会自动广播事件到你指定的广播驱动器上。

广播名称

Laravel 默认会使用事件的类名作为广播名称来广播事件。不过，你也可以在事件类中通过定义一个 `broadcastAs` 方法来自定义广播名称：

```
/**
 * 事件的广播名称。
 *
 * @return string
 */
public function broadcastAs()
{
    return 'server.created';
}
```

广播数据

当一个事件被广播时，它所有的 `public` 属性会自动被序列化为广播数据，这允许你在你的 JavaScript 应用中访问事件的公有数据。因此，举个例子，如果你的事件有一个公有的 `$user` 属性，它包含了一个 Elouqent 模型，那么事件的广播数据会是：

```
{
    "user": {
        "id": 1,
        "name": "Patrick Stewart"
        ...
    }
}
```

然而，如果你想更细粒度地控制你的广播数据，你可以添加一个 `broadcastWith` 方法到你的事件中。这个方法应该返回一个数组，该数组中的数据会被添加到广播数据中：

```
/**
 * 指定广播数据
 *
 * @return array
 */
public function broadcastWith()
{
    return ['id' => $this->user->id];
}
```

广播队列

默认情况下，每一个广播事件都被添加到默认的队列上，默认的队列连接在 `queue.php` 配置文件中指定。你可以通过在事件类中定义一个 `broadcastQueue` 属性来自定义广播器使用的队列。该属性用于指定广播使用的队列名称：

```
/**
 * 指定事件被放置在哪个队列上
 *
 * @var string
 */
public $broadcastQueue = 'your-queue-name';
```

频道授权

对于私有频道，用户只有被授权后才能监听。实现过程是用户向你的 Laravel 应用程序发起一个携带频道名称的 HTTP 请求，你的应用程序判断该用户是否能够监听该频道。在使用 [Laravel Echo](#) 时，上述 HTTP 请求会被自动发送；尽管如此，你仍然需要定义适当的路由来响应这些请求。

定义授权路由

幸运的是，我们可以在 Laravel 里很容易地定义路由来响应频道授权请求。在 `BroadcastServiceProvider` 中，你会看到一个对 `Broadcast::routes` 方法的调用。该方法会注册 `/broadcasting/auth` 路由来处理授权请求：

```
Broadcast::routes();
```

`Broadcast::routes` 方法会自动把它的路由放进 `web` 中间件组中；另外，如果你想对一些属性自定义，可以向该方法传递一个包含路由属性的数组：

```
Broadcast::routes($attributes);
```

定义授权回调

接下来，我们需要定义真正用于处理频道授权的逻辑。这是在 `routes/channels.php` 文件中完成。在该文件中，你可以用 `Broadcast::channel` 方法来注册频道授权回调函数：

```
Broadcast::channel('order.{orderId}', function ($user, $orderId) {
    return $user->id === Order::findOrNew($orderId)->user_id;
});
```

`channel` 方法接收两个参数：频道名称和一个回调函数，该回调通过返回 `true` 或 `false` 来表示用户是否被授权监听该频道。

所有的授权回调接收当前被认证的用户作为第一个参数，任何额外的通配符参数作为后续参数。在本例中，我们使用 `{orderId}` 占位符来表示频道名称的「ID」部分是通配符。

授权回调模型绑定

就像 HTTP 路由一样，频道路由也可以利用显式或隐式[路由模型绑定](#)。例如，相比于接收一个字符串或数字类型的 order ID，你也可以请求一个真正的 `Order` 模型实例：

```
use App\Order;

Broadcast::channel('order.{order}', function ($user, Order $order) {
    return $user->id === $order->user_id;
});
```

对事件进行广播

一旦你已经定义好了一个事件并实现了 `ShouldBroadcast` 接口，剩下的就是使用 `event` 函数来触发该事件。事件分发器会识别出实现了 `ShouldBroadcast` 接口的事件并将它们加入到队列进行广播：

```
event(new ShippingStatusUpdated($update));
```

只广播给他人

当创建一个使用到事件广播的应用程序时，你可以用 `broadcast` 函数来替代 `event` 函数。和 `event` 函数一样，`broadcast` 函数将事件分发到服务端侦听器：

```
broadcast(new ShippingStatusUpdated($update));
```

不同的是 `broadcast` 函数有一个 `toOthers` 方法允许你将当前用户从广播接收者中排除：

```
broadcast(new ShippingStatusUpdated($update))->toOthers();
```

为了更好地理解什么时候使用 `toOthers` 方法，让我们假设有一个任务列表的应用程序，用户可以通过输入任务名称来新建任务。为了新建任务，你的应用程序需要发起一个请求到 `/task` 路由，该路由在接收到请求并成功创建新任务后会触发一个任务被新建的事件广播，并返回新任务的 JSON 响应。当你的 JavaScript 应用程序接收到来自该路由的响应时，它会直接将新任务插入到任务列表，就像这样：

```
axios.post('/task', task)
    .then((response) => {
        this.tasks.push(response.data);
   });
```

然而，别忘了我们还将接收到一个事件广播。如果你的 JavaScript 应用程序同时监听该事件以便添加新任务到任务列表，你将会在你的列表中看到重复的任务：一份来自路由响应，另一份来自广播。

你可以通过使用 `toOthers` 方法让广播器只广播事件到其他用户来解决该问题。

配置

当你实例化 Laravel Echo 实例时，一个套接字 ID 会被指定到该连接。如果你使用 [Vue](#) 和 [Axios](#)，套接字 ID 会自动被添加到每一个请求的 `x-Socket-ID` 头中。然后，当你调用 `toOthers` 方法时，Laravel 会从头中提取出套接字 ID，并告诉广播器不要广播任何消息到该套接字 ID 的连接上。

如果你没有使用 Vue 和 Axios，则需要手动配置 JavaScript 应用程序来发送 `x-Socket-ID` 头。你可以用 `Echo.socketId` 方法来获取套接字 ID：

```
var socketId = Echo.socketId();
```

接收广播

安装 Laravel Echo

Laravel Echo 是一个 JavaScript 库，它使得订阅频道和监听由 Laravel 广播的事件变得非常容易。你可以通过 NPM 包管理器来安装 Echo。在本例中，因为我们将使用 Pusher 广播器，请安装 `pusher-js` 包：

```
npm install --save laravel-echo pusher-js
```

一旦 Echo 被安装好，你就可以在你应用程序的 JavaScript 中创建一个全新的 Echo 实例。做这件事的一个理想地方是在 `resources/assets/js/bootstrap.js` 文件的底部，Laravel 框架自带了该文件：

```
import Echo from "laravel-echo"

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: 'your-pusher-key'
});
```

当你使用 `pusher` 连接器来创建一个 Echo 实例的时候，你需要指定 `cluster` 以及指定连接是否需要加密：

```
window.Echo = new Echo({
    broadcaster: 'pusher',
    key: 'your-pusher-key',
    cluster: 'eu',
    encrypted: true
});
```

对事件进行监听

一旦你安装好并实例化了 Echo，你就可以开始监听事件广播了。首先，使用 `channel` 方法来获取一个频道实例，然后调用 `listen` 方法来监听指定的事件：

```
Echo.channel('orders')
  .listen('OrderShipped', (e) => {
    console.log(e.order.name);
  });

```

如果你想监听私有频道上的事件，请使用 `private` 方法。你可以通过链式调用 `listen` 方法来监听一个频道上的多个事件：

```
Echo.private('orders')
  .listen(...)
  .listen(...)
  .listen(...);
```

退出频道

如果想退出频道，你需要在你的 Echo 实例上调用 `leave` 方法：

```
Echo.leave('orders');
```

命名空间

你可能注意到了在上面的例子中我们没有为事件类指定完整的命名空间。这是因为 Echo 会自动认为事件在 `App\Events` 命名空间下。你可以在实例化 Echo 的时候传递一个 `namespace` 配置选项来指定根命名空间：

```
window.Echo = new Echo({
  broadcaster: 'pusher',
  key: 'your-pusher-key',
  namespace: 'App.Other.Namespace'
});
```

另外，你也可以在使用 Echo 订阅事件的时候为事件类加上 `.` 前缀。这时需要填写完全限定名称的类名：

```
Echo.channel('orders')
  .listen('.Namespace.Event.Class', (e) => {
    //
  });

```

Presence 频道

Presence 频道是在私有频道的安全性基础上，额外暴露出有哪些人订阅了该频道。这使得它可以很容易地建立强大的、协同的应用，如当有一个用户在浏览页面时，通知其他正在浏览相同页面的用户。

授权 Presence 频道

Presence 频道也是私有频道；因此，用户必须 [获得授权后才能访问他们](#)。与私有频道不同的是，在给 presence 频道定义授权回调函数时，如果一个用户已经加入了该频道，那么不应该返回 `true`，而应该返回一个关于该用户信息的数组。

由授权回调函数返回的数据能够在你的 JavaScript 应用程序中被 presence 频道事件监听器所使用。如果用户没有获得加入该 presence 频道的授权，那么你应该返回 `false` 或 `null`：

```
Broadcast::channel('chat.*', function ($user, $roomId) {
    if ($user->canJoinRoom($roomId)) {
        return ['id' => $user->id, 'name' => $user->name];
    }
});
```

加入 Presence 频道

你可以用 Echo 的 `join` 方法来加入 presence 频道。`join` 方法会返回一个实现了 `PresenceChannel` 的对象，它通过暴露 `listen` 方法，允许你订阅 `here`、`joining` 和 `leaving` 事件。

```
Echo.join(`chat.${roomId}`)
    .here((users) => {
        //
    })
    .joining((user) => {
        console.log(user.name);
    })
    .leaving((user) => {
        console.log(user.name);
    });
});
```

`here` 回调函数会在你成功加入频道后被立即执行，它接收一个包含用户信息的数组，用来告知当前订阅在该频道上的其他用户。`joining` 方法会在其他新用户加入到频道时被执行，`leaving` 会在其他用户退出频道时被执行。

广播到 Presence 频道

Presence 频道可以像公开和私有频道一样接收事件。使用一个聊天室的例子，我们要把 `NewMessage` 事件广播到聊天室的 presence 频道。要实现它，我们将从事件的 `broadcastOn` 方法中返回一个 `PresenceChannel` 实例：

```
/**
 * 指定事件在哪些频道上进行广播
 *
 * @return Channel|array
 */
public function broadcastOn()
{
    return new PresenceChannel('room.'.$this->message->room_id);
}
```

和公开或私有事件一样，presence 频道事件也能使用 `broadcast` 函数来广播。同样的，你还能用 `toOthers` 方法将当前用户从广播接收者中排除：

```
broadcast(new NewMessage($message));

broadcast(new NewMessage($message))->toOthers();
```

你可以通过 Echo 的 `listen` 方法来监听 `join` 事件：

```
Echo.join(`chat.${roomId}`)
.here(...)
.joining(...)
.leaving(...)
.listen('NewMessage', (e) => {
    //
});
```

客户端事件

有时候你可能希望广播一个事件给其他已经连接的客户端，但并不需要通知你的 Laravel 程序。试想一下当你想提醒用户，另外一个用户正在输入中的时候，显而易见客服端事件对于「输入中」之类的通知就显得非常有用了。你可以使用 Echo 的 `whisper` 方法来广播客户端事件：

```
Echo.channel('chat')
.whisper('typing', {
    name: this.user.name
});
```

你可以使用 `listenForWhisper` 方法来监听客户端事件：

```
Echo.channel('chat')
```

```
.listenForWhisper('typing', (e) => {
    console.log(e.name);
});
```

消息通知

将 [消息通知](#) 和事件广播一同使用，你的 JavaScript 应用程序可以在不刷新页面的情况下接收新的消息通知。首先，请先阅读关于如何使用 [广播消息通知频道](#) 的文档。

一旦你将一个消息通知配置为使用广播频道，你需要使用 Echo 的 `notification` 方法来监听广播事件。谨记，频道名称应该和接收消息通知的实体类名相匹配：

```
Echo.private(`App.User.${userId}`)
.notification((notification) => {
    console.log(notification.type);
});
```

在本例中，所有通过 `broadcast` 频道发送到 `App\User` 实例的消息通知都会被该回调接收到。一个针对 `App.User.{id}` 频道的授权回调函数已经被包含在 Laravel 的 `BroadcastServiceProvider` 中了。

Laravel 的缓存系统

- 配置信息
 - 驱动前提条件
- 缓存的使用
 - 获取一个缓存实例
 - 从缓存中获取项目
 - 存放项目到缓存中
 - 删 除缓存中的项目
 - Cache 帮助函数
- 缓存标签
 - 写入被标记的缓存项
 - 访问被标记的缓存项
 - 移除被标记的缓存项
- 增加自定义的缓存驱动
 - 写驱动
 - 注册驱动
- 缓存事件

配置信息

Laravel 给多种缓存系统提供丰富而统一的 API，缓存配置信息位于 `config/cache.php`，在这个文件中你可以为你的应用程序指定默认的缓存驱动，Laravel 支持当前流行的缓存系统，如非常棒的 [Memcached](#) 和 [Redis](#)。

缓存配置信息文件中也包括很多其他选项，你可以在文件中找到这些选项，请确保你看过这些选项说明。 Laravel 默认使用将序列化缓存对象保存在文件系统中的 `file` 缓存驱动，对于大型应用程序而言，推荐你使用如 Memcached 或者 Redis 这样更强大的缓存驱动。你甚至可以为一个驱动配置多个缓存配置信息。

驱动前提条件

数据库

当使用 `database` 缓存驱动时，你需要配置一个用来存放缓存项的数据库表，下面是一个 `Schema` 数据表结构声明的示例：

```
Schema::create('cache', function ($table) {
    $table->string('key')->unique();
    $table->text('value');
    $table->integer('expiration');
});
```

{tip} 你也可以使用 `php artisan cache:table` 这个 Artisan 命令生成一个有合适数据表结构的 migration。

Memcached

使用 Memcached 驱动需要安装 [Memcached PECL 扩展包](#)。你可以把所有 Memcached 服务器都列在 `config/cache.php` 这个配置信息文件中：

```
'memcached' => [
    [
        'host' => '127.0.0.1',
        'port' => 11211,
        'weight' => 100
    ],
],
```

你也可以把 `host` 选项配置到 UNIX 的 socket 路径中。如果你这样配置了，`port` 选项应该设置为 `0`：

```
'memcached' => [
    [
        'host' => '/var/run/memcached/memcached.sock',
        'port' => 0,
        'weight' => 100
    ],
],
```

Redis

在使用 Redis 作为 Laravel 的缓存驱动前，你需要通过 Composer 安装 [predis/predis](#) 扩展包 (~1.0) 或者使用 PECL 安装 PhpRedis PHP 拓展。

关于配置 Redis 的更多信息，请参考 [Laravel 文档页面](#)。

缓存的使用

获取一个缓存实例

`Illuminate\Contracts\Cache\Factory` 和 `Illuminate\Contracts\Cache\Repository` [contracts](#) 提供了访问 Laravel 缓存服务的机制。`Factory` contract 则为你的应用程序定义了访问所有缓存驱动的机制。`Repository` contract 是典型的用 `cache` 配置信息文件指定你的应用程序默认缓存驱动的实现。

然而，你也可以使用 `Cache facade`，我们将在文档的后续中介绍。`Cache facade` 提供了方便又简洁的方法访问缓存实例：

```

<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

class UserController extends Controller
{
    /**
     * Show a list of all users of the application.
     *
     * @return Response
     */
    public function index()
    {
        $value = Cache::get('key');

        //
    }
}

```

访问多个缓存仓库

使用 `Cache facade`, 可通过 `store` 方法来访问缓存仓库, 传入 `store` 方法的键应该对应一个缓存配置信息文件中的 `stores` 配置信息数组中列出的配置值:

```

$value = Cache::store('file')->get('foo');

Cache::store('redis')->put('bar', 'baz', 10);

```

从缓存中获取项目

`Cache facade` 中的 `get` 方法用来从缓存中获取缓存项, 如果缓存中不存在该缓存项, 返回 `null`。你也可以向 `get` 方法传递第二个参数, 用来指定缓存项不存在时返回的默认值:

```

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

$value = Cache::get('key', 'default');

```

你甚至可以将 `闭包` 作为默认值传递。如果指定的缓存项在缓存中不存在, `闭包` 的结果将被返回。传递一个闭包允许你延迟从数据库或外部服务中取出默认值:

```

$value = Cache::get('key', function () {
    return DB::table(...)->get();
});

```

确认项目是否存在

`has` 方法可以用来检查一个项目是否存在于缓存中:

```
if (Cache::has('key')) {
    //
}
```

递增与递减值

`递增` 和 `递减` 方法可以用来调整缓存中整数项目值。这两个方法都可以传入一个可选的第二个参数，用来指示要递增或递减多少值:

```
Cache::increment('key');
Cache::increment('key', $amount);
Cache::decrement('key');
Cache::decrement('key', $amount);
```

获取和更新

有时你可能会想从缓存中取出一个项目，但也想在取出的项目不存在时存入一个默认值，例如，你可能会想从缓存中取出所有用户，或者当用户不存在时，从数据库中将这些用户取出并放入缓存中，你可以使用 `Cache::remember` 方法实现:

```
$value = Cache::remember('users', $minutes, function () {
    return DB::table('users')->get();
});
```

如果缓存项在缓存中不存在，则返回给 `remember` 方法的闭包将会被运行，而且闭包的运行结果将会被存放在缓存中。

获取和删除

如果你需要从缓存中获取一个缓存项然后删除它，你可以使用 `pull` 方法。像 `get` 方法一样，如果缓存项在缓存中不存在，`null` 将被返回:

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

存储项目到缓存中

你可以使用 `Cache facade` 的 `put` 方法来存放缓存项到缓存中，当你在缓存中存放缓存项时，你需要使用第三个参数来设定缓存的存放时间:

```
Cache::put('key', 'value', $minutes);
```

如果要指定一个缓存项过期的分钟数，你也可以传递一个 `DateTime` 实例来表示该缓存项过期的时间点：

```
$expiresAt = Carbon::now()->addMinutes(10);

Cache::put('key', 'value', $expiresAt);
```

写入目前不存在的项目

`add` 方法只会把暂时不存在于缓存中的缓存项放入缓存，如果存放成功将返回 `true`，否则返回 `false`：

```
Cache::add('key', 'value', $minutes);
```

永久写入项目

`forever` 方法可以用来将缓存项永久存入缓存中，因为这些缓存项不会过期，所以必须通过 `forget` 方法手动删除：

```
Cache::forever('key', 'value');
```

{tip} 如果你在使用 Memcached 驱动，那么当缓存达到大小限制时，那些「永久」保存的缓存项可能被移除。

从缓存中移除项目

你可以使用 `forget` 方法从缓存中移除一个项目：

```
Cache::forget('key');
```

也可以使用 `flush` 方法清空所有缓存：

```
Cache::flush();
```

{note} 清空缓存并不会遵从缓存的前缀，并且会将缓存中所有的缓存项删除。在清除与其它应用程序共享的缓存时应谨慎考虑这一点。

Cache 帮助函数

除了可以使用 `Cache facade` 或者 `cache contract` 之外，你也可以使用全局帮助函数 `cache` 来获取和保存缓存数据。当 `cache` 只接收一个字符串参数的时候，它将会返回给定键对应的值：

```
$value = cache('key');
```

如果你传给函数一个键值对数组和过期时间，它将会把值和过期时间保存在缓存中：

```
cache(['key' => 'value'], $minutes);

cache(['key' => 'value'], Carbon::now()->addSeconds(10));
```

{tip} 如果在测试中使用全局函数 `cache`，你应该使用 `Cache::shouldReceive` 方法，就好像你在[测试 facade](#)一样。

缓存标签

{note} 缓存标签并不支持使用 `file` 或 `dababase` 的缓存驱动。此外，当在缓存使用多个标签并「永久」写入时，类似 `memcached` 的驱动性能会是最佳的，且会自动清除旧的纪录。

写入被标记的缓存项

缓存标签允许你在缓存中标记关联的项目，并清空所有已分配指定标签的缓存值。你可以通过传入一组标签名称的有序数组，以访问被标记的缓存。举例来说，让我们访问一个被标记的缓存并 `put` 值给它：

```
Cache::tags(['people', 'artists'])->put('John', $john, $minutes);

Cache::tags(['people', 'authors'])->put('Anne', $anne, $minutes);
```

访问被标记的缓存项

若要获取一个被标记的缓存项，只要传递一样的有序标签列表至 `tags` 方法，然后通过你希望获取的值对应的键来调用 `get` 方法：

```
$john = Cache::tags(['people', 'artists'])->get('John');

$anne = Cache::tags(['people', 'authors'])->get('Anne');
```

移除被标记的缓存项

你可以清空已分配的单个标签或是一组标签列表中的所有缓存项。例如，下方的语句会把被标记为 `people`、`authors`，或两者都标记了的缓存都移除。所以，`Anne` 与 `John` 都会被从缓存中移除：

```
Cache::tags(['people', 'authors'])->flush();
```

相反的，下方的语句只会删除被标示为 `authors` 的缓存，所以 `Anne` 会被移除，但 `John` 不会：

```
Cache::tags('authors')->flush();
```

增加自定义的缓存驱动

写驱动

为了创建自定义的缓存驱动，首先我们需要部署 `Illuminate\Contracts\Cache\Store` contract。所以 MongoDB 缓存实现看起来会像这样：

```
<?php

namespace App\Extensions;

use Illuminate\Contracts\Cache\Store;

class MongoStore implements Store
{
    public function get($key) {}
    public function many(array $keys);
    public function put($key, $value, $minutes) {}
    public function putMany(array $values, $minutes);
    public function increment($key, $value = 1) {}
    public function decrement($key, $value = 1) {}
    public function forever($key, $value) {}
    public function forget($key) {}
    public function flush() {}
    public function getPrefix() {}
}
```

我们只需要通过一个 MongoDB 的连接来实现这些方法。关于如何实现这些方法，可以查看框架源代码中的 `Illuminate\Cache\MemcachedStore`。一旦我们的部署完成，我们就可以完成自定义驱动的注册了。

```
Cache::extend('mongo', function ($app) {
    return Cache::repository(new MongoStore());
});
```

{tip} 如果你想知道把自定义的缓存驱动代码放置在哪里，你可以在 `app` 目录下创建一个 `Extensions` 命名空间。Laravel 没有硬性规定应用程序的结构，你可以依照你的喜好任意组织你的应用程序。

注册驱动

通过 Laravel 注册自定义缓存驱动，我们将用到 `Cache facade` 的 `extend` 方法。`Cache::extend` 的调用会在最新的 Laravel 应用程序默认的 `App\Providers\AppServiceProvider` 的 `boot` 方法中完成。或者你可以创建你自己的服务提供者来放置这些扩展 - 不要忘记在 `config/app.php` 提供者数组中注册提供者：

```
<?php

namespace App\Providers;

use App\Extensions\MongoStore;
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\ServiceProvider;

class CacheServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    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。这取决于你的 `config/cache.php` 配置信息文件的 `driver` 选项。第二个参数为一个应该返回 `Illuminate\Cache\Repository` 实例的闭包。这个闭包将传递一个 `service container` 的 `$app` 实例。

一旦你的扩展被注册，就可以轻松的更新 `config/cache.php` 配置信息文件的 `driver` 选项为你的扩展名称。

缓存事件

为了在每一次缓存操作时执行代码，你可以监听缓存触发的事件 [事件](#)。一般来说，你必须将这些事件监听器放置在 `EventServiceProvider`：

```
/**
 * 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',
    ],
];
```

Laravel 的集合 Collection

- 简介
 - 创建集合
- 可用的方法
- 高阶信息传递

简介

`Illuminate\Support\Collection` 类提供一个流畅、便利的封装来操控数组数据。如下面的示例代码，我们用 `collect` 函数从数组中创建新的集合实例，对每一个元素运行 `strtoupper` 函数，然后移除所有的空元素：

```
$collection = collect(['taylor', 'abigail', null])->map(function ($name) {
    return strtoupper($name);
})
->reject(function ($name) {
    return empty($name);
});
```

如上面的代码示例，`Collection` 类支持链式调用，一般来说，每一个 `Collection` 方法会返回一个新的 `collection` 实例，你可以放心地进行链接调用。

创建集合

如上所述，`collect` 辅助函数会利用传入的数组生成一个新的 `Illuminate\Support\Collection` 实例。所以要创建一个集合就这么简单：

```
$collection = collect([1, 2, 3]);
```

{tip} 默认 `Eloquent` 模型的查询结果总是以 `Collection` 实例返回。

可用的方法

接下来，我们将会探讨 `Collection` 类的所有方法。要记得的是，所有方法都支持链式调用，几乎所有的方法都会返回新的 `Collection` 实例，让你保留原版的集合以备不时之需。

[all] (#method-all) [avg] (#method-avg) [chunk] (#method-chunk) [collapse] (#method-collapse)
 [combine] (#method-combine) [contains] (#method-contains) [count] (#method-count) [diff] (#method-diff)
 [diffKeys] (#method-diffkeys) [each] (#method-each) [every] (#method-every) [except] (#method-except)
 [filter] (#method-filter) [first] (#method-first) [flatMap] (#method-flatmap) [flatten] (#method-flatten)
 [flip] (#method-flip) [forget] (#method-forget) [forPage] (#method-forpage) [get] (#method-get)

[groupBy](#method-groupby) [has](#method-has) [implode](#method-implode) [intersect](#method-intersect) [isEmpty](#method-isempty) [keyBy](#method-keyby) [keys](#method-keys) [last](#method-last) [map](#method-map) [mapWithKeys](#method-mapwithkeys) [max](#method-max) [merge](#method-merge) [min](#method-min) [nth](#method-nth) [only](#method-only) [partition](#method-partition) [pipe](#method-pipe) [pluck](#method-pluck) [pop](#method-pop) [prepend](#method-prepend) [pull](#method-pull) [push](#method-push) [put](#method-put) [random](#method-random) [reduce](#method-reduce) [reject](#method-reject) [reverse](#method-reverse) [search](#method-search) [shift](#method-shift) [shuffle](#method-shuffle) [slice](#method-slice) [sort](#method-sort) [sortBy](#method-sortby) [sortByDesc](#method-sortbydesc) [splice](#method-splice) [split](#method-split) [sum](#method-sum) [take](#method-take) [toArray](#method-toarray) [toJson](#method-tojson) [transform](#method-transform) [union](#method-union) [unique](#method-unique) [values](#method-values) [when](#method-when) [where](#method-where) [whereStrict](#method-wherestrict) [whereIn](#method-wherein) [whereInStrict](#method-whereinstrict) [zip](#method-zip)

方法清单

`all()`

返回该集合所代表的底层 `数组`：

```
collect([1, 2, 3])->all();
// [1, 2, 3]
```

`avg()`

返回集合中所有项目的平均值：

```
collect([1, 2, 3, 4, 5])->avg();
// 3
```

如果集合包含了嵌套数组或对象，你可以通过传递「键」来指定使用哪些值计算平均值：

```
$collection = collect([
    ['name' => 'JavaScript: The Good Parts', 'pages' => 176],
    ['name' => 'JavaScript: The Definitive Guide', 'pages' => 1096],
]);
$collection->avg('pages');

// 636
```

chunk()

将集合拆成多个指定大小的较小集合：

```
$collection = collect([1, 2, 3, 4, 5, 6, 7]);

$chunks = $collection->chunk(4);

$chunks->toArray();

// [[1, 2, 3, 4], [5, 6, 7]]
```

这个方法在适用于网格系统如 [Bootstrap](#) 的 [视图](#)。想像你有一个 [Eloquent](#) 模型的集合要显示在一个网格内：

```
@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()

将多个数组组成的集合合成单个一维数组集合：

```
$collection = collect([[1, 2, 3], [4, 5, 6], [7, 8, 9]]);

$collapsed = $collection->collapse();

$collapsed->all();

// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

combine()

将集合的值作为「键」，合并另一个数组或者集合作为「键」对应的值。

```
$collection = collect(['name', 'age']);

$combined = $collection->combine(['George', 29]);

$combined->all();

// ['name' => 'George', 'age' => 29]
```

contains()

判断集合是否含有指定项目：

```
$collection = collect(['name' => 'Desk', 'price' => 100]);  
  
$collection->contains('Desk');  
  
// true  
  
$collection->contains('New York');  
  
// false
```

你可以将一对键/值传入 `contains` 方法，用来判断该组合是否存在于集合内：

```
$collection = collect([  
    ['product' => 'Desk', 'price' => 200],  
    ['product' => 'Chair', 'price' => 100],  
]);  
  
$collection->contains('product', 'Bookcase');  
  
// false
```

最后，你也可以传入一个回调函数到 `contains` 方法内运行你自己的判断语句：

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$collection->contains(function ($value, $key) {  
    return $value > 5;  
});  
  
// false
```

count()

返回该集合内的项目总数：

```
$collection = collect([1, 2, 3, 4]);  
  
$collection->count();  
  
// 4
```

diff()

将集合与其它集合或纯 PHP 数组 进行值的比较，返回第一个集合中存在而第二个集合中不存在的值：

```
$collection = collect([1, 2, 3, 4, 5]);

$diff = $collection->diff([2, 4, 6, 8]);

$diff->all();

// [1, 3, 5]
```

diffKeys()

将集合与其它集合或纯 PHP 数组 的「键」进行比较，返回第一个集合中存在而第二个集合中不存在「键」所对应的键值对：

```
$collection = collect([
    'one' => 10,
    'two' => 20,
    'three' => 30,
    'four' => 40,
    'five' => 50,
]);

$diff = $collection->diffKeys([
    'two' => 2,
    'four' => 4,
    'six' => 6,
    'eight' => 8,
]);

$diff->all();

// ['one' => 10, 'three' => 30, 'five' => 50]
```

each()

遍历集合中的项目，并将之传入回调函数：

```
$collection = $collection->each(function ($item, $key) {
    //
});
```

回调函数中返回 `false` 以中断循环：

```
$collection = $collection->each(function ($item, $key) {
    if /* some condition */) {
        return false;
    }
});
```

every()

判断集合中每一个元素是否都符合指定条件：

```
collect([1, 2, 3, 4])->every(function ($value, $key) {
    return $value > 2;
});

// false
```

except()

返回集合中除了指定键以外的所有项目：

```
$collection = collect(['product_id' => 1, 'price' => 100, 'discount' => false]);

$filtered = $collection->except(['price', 'discount']);

$filtered->all();

// ['product_id' => 1]
```

与 `except` 相反的方法请查看 [only](#)。

filter()

使用回调函数筛选集合，只留下那些通过判断测试的项目：

```
$collection = collect([1, 2, 3, 4]);

$filtered = $collection->filter(function ($value, $key) {
    return $value > 2;
});

$filtered->all();

// [3, 4]
```

如果没有提供回调函数，集合中所有返回 `false` 的元素都会被移除：

```
$collection = collect([1, 2, 3, null, false, '', 0, []]);
$collection->filter()->all();

// [1, 2, 3]
```

与 `filter` 相反的方法可以查看 `reject`。

`first()`

返回集合第一个通过指定测试的元素：

```
collect([1, 2, 3, 4])->first(function ($value, $key) {
    return $value > 2;
});

// 3
```

你也可以不传入参数使用 `first` 方法以获取集合中第一个元素。如果集合是空的，则会返回 `null`：

```
collect([1, 2, 3, 4])->first();

// 1
```

`flatMap()`

对集合内所有子集遍历执行回调，并在最后转为一维集合：

```
$collection = collect([
    ['name' => 'Sally'],
    ['school' => 'Arkansas'],
    ['age' => 28]
]);

$flattened = $collection->flatMap(function ($values) {
    return array_map('strtoupper', $values);
});

$flattened->all();

// ['name' => 'SALLY', 'school' => 'ARKANSAS', 'age' => '28'];
```

flatten()

将多维集合转为一维集合：

```
$collection = collect(['name' => 'taylor', 'languages' => ['php', 'javascript']]);

$flattened = $collection->flatten();

$flattened->all();

// ['taylor', 'php', 'javascript'];
```

你可以选择性地传入遍历深度的参数：

```
$collection = collect([
    'Apple' => [
        ['name' => 'iPhone 6S', 'brand' => 'Apple'],
    ],
    'Samsung' => [
        ['name' => 'Galaxy S7', 'brand' => 'Samsung']
    ],
]);

$products = $collection->flatten(1);

$products->values()->all();

/*
[
    ['name' => 'iPhone 6S', 'brand' => 'Apple'],
    ['name' => 'Galaxy S7', 'brand' => 'Samsung'],
]
*/
```

在这个例子里，调用 `flatten` 方法时不传入深度参数会遍历嵌套数组降维成一维数组，生成 `['iPhone 6S', 'Apple', 'Galaxy S7', 'Samsung']`，传入深度参数能让你限制降维嵌套数组的层数。

flip()

将集合中的键和对应的数值进行互换：

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$flipped = $collection->flip();

$flipped->all();
```

```
// ['taylor' => 'name', 'laravel' => 'framework']
```

`forget()`

通过集合的键来移除掉集合中的一个项目：

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$collection->forget('name');

$collection->all();

// ['framework' => 'laravel']
```

{note} 与大多数其它集合的方法不同，`forget` 不会返回修改过后的集合；它会直接修改调用它的集合。

`forPage()`

返回可用来在指定页码上所显示项目的新集合。这个方法第一个参数是页码数，第二个参数是每页显示的个数。

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9]);

$chunk = $collection->forPage(2, 3);

$chunk->all();

// [4, 5, 6]
```

`get()`

返回指定键的项目。如果该键不存在，则返回 `null`：

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('name');

// taylor
```

你可以选择性地传入一个默认值作为第二个参数：

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('foo', 'default-value');

// default-value
```

你甚至可以传入回调函数当默认值。如果指定的键不存在，就会返回回调函数的运行结果：

```
$collection->get('email', function () {
    return 'default-value';
});

// default-value
```

groupBy()

根据指定的「键」为集合内的项目分组：

```
$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'],
    ],
]
*/
```

除了传入字符串的「键」之外，你也可以传入回调函数。该函数应该返回你希望用来分组的键的值。

```
$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()`

检查集合中是否含有指定的「键」：

```
$collection = collect(['account_id' => 1, 'product' => 'Desk']);

$collection->has('product');

// true
```

`implode()`

`implode` 方法合并集合中的项目。它的参数依集合中的项目类型而定。假如集合含有数组或对象，你应该传入你希望连接的属性的「键」，以及你希望放在数值之间的拼接字符串：

```
$collection = collect([
    ['account_id' => 1, 'product' => 'Desk'],
    ['account_id' => 2, 'product' => 'Chair'],
]);

$collection->implode('product', ', ');

// Desk, Chair
```

假如集合只含有简单的字符串或数字，则只需要传入拼接的字符串作为该方法的唯一参数即可：

```
collect([1, 2, 3, 4, 5])->implode('-');

// '1-2-3-4-5'
```

`intersect()`

移除任何指定 数组 或集合内所没有的数值。最终集合保存着原集合的键：

```
$collection = collect(['Desk', 'Sofa', 'Chair']);

$intersect = $collection->intersect(['Desk', 'Chair', 'Bookcase']);

$intersect->all();

// [0 => 'Desk', 2 => 'Chair']
```

`isEmpty()`

如果集合是空的，`isEmpty` 方法会返回 `true`；否则返回 `false`：

```
collect([])->isEmpty();

// true
```

`keyBy()`

以指定键的值作为集合项目的键。如果几个数据项有相同的键，那在新集合中只显示最后一项：

```
$collection = collect([
    ['product_id' => 'prod-100', 'name' => 'desk'],
    ['product_id' => 'prod-200', 'name' => 'chair'],
]);

$keyed = $collection->keyBy('product_id');

$keyed->all();

/*
[
    'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
]
*/
```

你也可以传入自己的回调函数，该函数应该返回集合的键的值：

```
$keyed = $collection->keyBy(function ($item) {
    return strtoupper($item['product_id']);
});
```

```
$keyed->all();

/*
[
    'PROD-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'PROD-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
]
*/
```

`keys()`

返回该集合所有的键：

```
$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()`

返回集合中，最后一个通过指定测试的元素：

```
collect([1, 2, 3, 4])->last(function ($value, $key) {
    return $value < 3;
});

// 2
```

你也可以不传入参数使用 `last` 方法以获取集合中最后一个元素。如果集合是空的，则会返回

`null`：

```
collect([1, 2, 3, 4])->last();

// 4
```

`map()`

遍历整个集合并将每一个数值传入回调函数。回调函数可以任意修改并返回项目，形成修改过的项目组成的新集合：

```
$collection = collect([1, 2, 3, 4, 5]);

$multiplied = $collection->map(function ($item, $key) {
    return $item * 2;
});

$multiplied->all();

// [2, 4, 6, 8, 10]
```

{note} 正如集合大多数其它的方法一样，`map` 返回一个新集合实例；它并没有修改被调用的集合。假如你想改变原始的集合，得使用 `transform` 方法。

mapWithKeys()

遍历整个集合并将每一个数值传入回调函数。回调函数返回包含一个键值对的关联数组：

```
$collection = collect([
    [
        'name' => 'John',
        'department' => 'Sales',
        'email' => 'john@example.com'
    ],
    [
        'name' => 'Jane',
        'department' => 'Marketing',
        'email' => 'jane@example.com'
    ]
]);

$keyed = $collection->mapWithKeys(function ($item) {
    return [$item['email'] => $item['name']];
});

$keyed->all();

/*
[
    'john@example.com' => 'John',
    'jane@example.com' => 'Jane',
]
*/
```

max()

计算指定键的最大值：

```
$max = collect([('foo' => 10], ['foo' => 20])->max('foo');

// 20

$max = collect([1, 2, 3, 4, 5])->max();

// 5
```

merge()

合并数组进集合。数组「键」对应的数值会覆盖集合「键」对应的数值：

```
$collection = collect(['product_id' => 1, 'price' => 100]);

$merged = $collection->merge(['price' => 200, 'discount' => false]);

$merged->all();

// ['product_id' => 1, 'price' => 200, 'discount' => false]
```

如果指定数组的「键」为数字，则「值」将会合并到集合的后面：

```
$collection = collect(['Desk', 'Chair']);

$merged = $collection->merge(['Bookcase', 'Door']);

$merged->all();

// ['Desk', 'Chair', 'Bookcase', 'Door']
```

min()

计算指定「键」的最小值：

```
$min = collect([('foo' => 10], ['foo' => 20])->min('foo');

// 10

$min = collect([1, 2, 3, 4, 5])->min();

// 1
```

nth()

由每隔第 n 个元素组成一个新的集合：

```
$collection = collect(['a', 'b', 'c', 'd', 'e', 'f']);

$collection->nth(4);

// ['a', 'e']
```

你也可以选择传入一个偏移量作为第二个参数

```
$collection->nth(4, 1);

// ['b', 'f']
```

only()

返回集合中指定键的所有项目：

```
$collection = collect(['product_id' => 1, 'name' => 'Desk', 'price' => 100, 'discount' => false]);

$filtered = $collection->only(['product_id', 'name']);

$filtered->all();

// ['product_id' => 1, 'name' => 'Desk']
```

与 `only` 相反的方法请查看 [except](#)。

partition()

结合 PHP 中的 `list` 方法来分开符合指定条件的元素以及那些不符合指定条件的元素：

```
$collection = collect([1, 2, 3, 4, 5, 6]);

list($underThree, $aboveThree) = $collection->partition(function ($i) {
    return $i < 3;
});
```

pipe()

将集合传给回调函数并返回结果：

```
$collection = collect([1, 2, 3]);

$piped = $collection->pipe(function ($collection) {
    return $collection->sum();
});

// 6
```

pluck()

获取集合中指定「键」所有对应的值：

```
$collection = collect([
    ['product_id' => 'prod-100', 'name' => 'Desk'],
    ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$plucked = $collection->pluck('name');

$plucked->all();

// ['Desk', 'Chair']
```

你也可以指定最终集合的键：

```
$plucked = $collection->pluck('name', 'product_id');

$plucked->all();

// ['prod-100' => 'Desk', 'prod-200' => 'Chair']
```

pop()

移除并返回集合最后一个项目：

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->pop();

// 5

$collection->all();

// [1, 2, 3, 4]
```

prepend()

在集合前面增加一项数组的值：

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$collection->prepend(0);  
  
$collection->all();  
  
// [0, 1, 2, 3, 4, 5]
```

你可以传递第二个参数来设置新增加项的键：

```
$collection = collect(['one' => 1, 'two' => 2]);  
  
$collection->prepend(0, 'zero');  
  
$collection->all();  
  
// ['zero' => 0, 'one' => 1, 'two' => 2]
```

pull()

把「键」对应的值从集合中移除并返回：

```
$collection = collect(['product_id' => 'prod-100', 'name' => 'Desk']);  
  
$collection->pull('name');  
  
// 'Desk'  
  
$collection->all();  
  
// ['product_id' => 'prod-100']
```

push()

在集合的后面新添加一个元素：

```
$collection = collect([1, 2, 3, 4]);  
  
$collection->push(5);  
  
$collection->all();
```

```
// [1, 2, 3, 4, 5]
```

put()

在集合内设置一个「键/值」：

```
$collection = collect(['product_id' => 1, 'name' => 'Desk']);

$collection->put('price', 100);

$collection->all();

// ['product_id' => 1, 'name' => 'Desk', 'price' => 100]
```

random()

`random` 方法从集合中随机返回一个项目：

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->random();

// 4 - (retrieved randomly)
```

你可以选择性地传入一个整数到 `random`。如果该整数大于 `1`，则会返回一个集合：

```
$random = $collection->random(3);

$random->all();

// [2, 4, 5] - (retrieved randomly)
```

reduce()

`reduce` 方法将集合缩减到单个数值，该方法会将每次迭代的结果传入到下一次迭代：

```
$collection = collect([1, 2, 3]);

$total = $collection->reduce(function ($carry, $item) {
    return $carry + $item;
});

// 6
```

第一次迭代时 `$carry` 的数值为 `null`；然而你也可以传入第二个参数进 `reduce` 以指定它的初始值：

```
$collection->reduce(function ($carry, $item) {
    return $carry + $item;
}, 4);

// 10
```

reject()

`reject` 方法以指定的回调函数筛选集合。会移除掉那些通过判断测试（即结果返回 `true`）的项目：

```
$collection = collect([1, 2, 3, 4]);

$filtered = $collection->reject(function ($value, $key) {
    return $value > 2;
});

$filtered->all();

// [1, 2]
```

与 `reject` 相反的方法可以查看 [filter](#) 方法。

reverse()

`reverse` 方法倒转集合内项目的顺序：

```
$collection = collect([1, 2, 3, 4, 5]);

$reversed = $collection->reverse();

$reversed->all();

// [5, 4, 3, 2, 1]
```

search()

`search` 方法在集合内搜索指定的数值并返回找到的键。假如找不到项目，则返回 `false`：

```
$collection = collect([2, 4, 6, 8]);
```

```
$collection->search(4);  
// 1
```

搜索是用「宽松」匹配来进行，也就是说如果字符串值是整数那它就跟这个整数是相等的。要使用严格匹配的话，就传入 `true` 为该方法的第二个参数：

```
$collection->search('4', true);  
// false
```

另外，你可以传入你自己的回调函数来搜索第一个通过你判断测试的项目：

```
$collection->search(function ($item, $key) {  
    return $item > 5;  
});  
  
// 2
```

shift()

`shift` 方法移除并返回集合的第一个项目：

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$collection->shift();  
  
// 1  
  
$collection->all();  
  
// [2, 3, 4, 5]
```

shuffle()

`shuffle` 方法随机排序集合的项目：

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$shuffled = $collection->shuffle();  
  
$shuffled->all();  
  
// [3, 2, 5, 1, 4] // (generated randomly)
```

slice()

`slice` 方法返回集合从指定索引开始的一部分切片：

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

$slice = $collection->slice(4);

$slice->all();

// [5, 6, 7, 8, 9, 10]
```

如果你想限制返回切片的大小，就传入想要的大小为方法的第二个参数：

```
$slice = $collection->slice(4, 2);

$slice->all();

// [5, 6]
```

返回的切片将会保留原始键作为索引。假如你不希望保留原始的键，你可以使用 `values` 方法来重新建立索引。

sort()

对集合排序。排序后的集合保留着原始数组的键，所以在这个例子里我们用 `values` 方法来把键设置为连续数字的键。

```
$collection = collect([5, 3, 1, 2, 4]);

$sorted = $collection->sort();

$sorted->values()->all();

// [1, 2, 3, 4, 5]
```

假如你需要更高级的排序，你可以传入回调函数以你自己的算法进行 排序 。参考 PHP 文档的 `usort`，这是集合的 `sort` 方法在背后所调用的函数。

{tip} 要排序嵌套数组或对象的集合，见 `sortBy` 和 `sortByDesc` 方法。

sortBy()

以指定的键排序集合。排序后的集合保留了原始数组键，所以在这个例子中我们用 `values` method 把键设置为连续数字的索引建：

```
$collection = collect([
    ['name' => 'Desk', 'price' => 200],
    ['name' => 'Chair', 'price' => 100],
    ['name' => 'Bookcase', 'price' => 150],
]);

$sorted = $collection->sortBy('price');

$sorted->values()->all();

/*
[
    ['name' => 'Chair', 'price' => 100],
    ['name' => 'Bookcase', 'price' => 150],
    ['name' => 'Desk', 'price' => 200],
]
*/

```

你也可以传入自己的回调函数以决定如何排序集合数值：

```
$collection = collect([
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]);

$sorted = $collection->sortBy(function ($product, $key) {
    return count($product['colors']);
});

$sorted->values()->all();

/*
[
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]
*/

```

`sortByDesc()`

与 `sortBy` 有着一样的形式，但是会以相反的顺序来排序集合：

`splice()`

返回从指定的索引开始的一小切片项目，原本集合也会被切除：

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$chunk = $collection->splice(2);  
  
$chunk->all();  
  
// [3, 4, 5]  
  
$collection->all();  
  
// [1, 2]
```

你可以传入第二个参数以限制大小：

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$chunk = $collection->splice(2, 1);  
  
$chunk->all();  
  
// [3]  
  
$collection->all();  
  
// [1, 2, 4, 5]
```

此外，你可以传入含有新项目的第三个参数以取代集合中被移除的项目：

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$chunk = $collection->splice(2, 1, [10, 11]);  
  
$chunk->all();  
  
// [3]  
  
$collection->all();  
  
// [1, 2, 10, 11, 4, 5]
```

split()

将集合按指定组数分解：

```
$collection = collect([1, 2, 3, 4, 5]);
```

```
$groups = $collection->split(3);

$groups->toArray();

// [[1, 2], [3, 4], [5]]
```

`sum()`

返回集合内所有项目的总和:

```
collect([1, 2, 3, 4, 5])->sum();

// 15
```

如果集合包含嵌套数组或对象, 你应该传入一个「键」来指定要用哪些数值来计算总和:

```
$collection = collect([
    ['name' => 'JavaScript: The Good Parts', 'pages' => 176],
    ['name' => 'JavaScript: The Definitive Guide', 'pages' => 1096],
]);

$collection->sum('pages');

// 1272
```

此外, 你可以传入自己的回调函数来决定要用哪些数值来计算总和:

```
$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()`

返回有着指定数量项目的集合:

```
$collection = collect([0, 1, 2, 3, 4, 5]);
```

```
$chunk = $collection->take(3);

$chunk->all();

// [0, 1, 2]
```

你也可以传入负整数以获取从集合后面来算指定数量的项目：

```
$collection = collect([0, 1, 2, 3, 4, 5]);

$chunk = $collection->take(-2);

$chunk->all();

// [4, 5]
```

toArray()

将集合转换成纯 PHP 数组。假如集合的数值是 [Eloquent](#) 模型，也会被转换成数组：

```
$collection = collect(['name' => 'Desk', 'price' => 200]);

$collection->toArray();

/*
[
    ['name' => 'Desk', 'price' => 200],
]
*/
```

{note} `toArray` 也会转换所有内嵌的对象为数组。假如你希望获取原本的底层数组，改用 `all` 方法。

toJson()

将集合转换成 JSON：

```
$collection = collect(['name' => 'Desk', 'price' => 200]);

$collection->toJson();

// '{"name": "Desk", "price": 200}'
```

transform()

遍历集合并对集合内每一个项目调用指定的回调函数。集合的项目将会被回调函数返回的数值取代掉：

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->transform(function ($item, $key) {
    return $item * 2;
});

$collection->all();

// [2, 4, 6, 8, 10]
```

{note} 与大多数其它集合的方法不同，`transform` 会修改集合本身。如果你希望创建新集合，就改用 `map` 方法。

union()

将给定的数组合并到集合中，如果数组中含有与集合一样的「键」，集合的键值会被保留：

```
$collection = collect([1 => ['a'], 2 => ['b']]);

$union = $collection->union([3 => ['c'], 1 => ['b']]);

$union->all();

// [1 => ['a'], 2 => ['b'], 3 => ['c']]
```

unique()

`unique` 方法返回集合中所有唯一的项目。返回的集合保留着原始键，所以在这个例子中我们用 `values` 方法来把键重置为连续数字的键。

```
$collection = collect([1, 1, 2, 2, 3, 4, 2]);

$unique = $collection->unique();

$unique->values()->all();

// [1, 2, 3, 4]
```

当处理嵌套数组或对象的时候，你可以指定用来决定唯一性的键：

```
$collection = collect([
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'iPhone 5', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
    ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],
]);
$unique = $collection->unique('brand');
$unique->values()->all();
/*
[
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
]
*/

```

你可以传入自己的回调函数来确定项目的唯一性：

```
$unique = $collection->unique(function ($item) {
    return $item['brand'].$item['type'];
});
$unique->values()->all();
/*
[
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
    ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],
]
*/

```

`values()`

返回「键」重新被设为「连续整数」的新集合：

```
$collection = collect([
    10 => ['product' => 'Desk', 'price' => 200],
    11 => ['product' => 'Desk', 'price' => 200]
]);
$values = $collection->values();
```

```
$values->all();

/*
[
    0 => ['product' => 'Desk', 'price' => 200],
    1 => ['product' => 'Desk', 'price' => 200],
]
*/
```

when()

当第一个参数运算结果为 `true` 的时候，会执行第二个参数传入的闭包：

```
$collection = collect([1, 2, 3]);

$collection->when(true, function ($collection) {
    return $collection->push(4);
});

$collection->all();

// [1, 2, 3, 4]
```

where()

以一对指定的「键 / 数值」筛选集合：

```
$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],
]
*/
```

比较数值的时候用了「宽松」匹配方式，查看 `whereStrict` method 来用严格比较的方式过滤。

`whereStrict()`

这个方法与 `where` 方法有着一样的形式；但是会以「严格」匹配来匹配数值：

`whereIn()`

基于参数中的键值数组进行过滤：

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereIn('price', [150, 200]);

$filtered->all();

/*
[
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Desk', 'price' => 200],
]
*/
```

此方法是用宽松的匹配，你可以使用 `whereInStrict` 做比较 `严格` 的匹配。

`whereInStrict()`

此方法的使用于 `whereIn` 方法类似，只是使用了比较 `严格` 的过滤。

`zip()`

`zip` 方法将集合与指定数组相同索引的值合并在一起：

```
$collection = collect(['Chair', 'Desk']);

$zipped = $collection->zip([100, 200]);

$zipped->all();

// [['Chair', 100], ['Desk', 200]]
```

高阶信息传递

集合也提供「高阶信息传递支持」，这是对集合执行常见操作的快捷方式。支持高阶信息传递的集合方法有：`contains`，`each`，`every`，`filter`，`first`，`map`，`partition`，`reject`，`sortBy`，`sortByDesc` 和 `sum`。

每个高阶信息都能作为集合实例的动态属性来访问。例如，我们在集合中使用 `each` 高阶信息传递方法拉对每个对象去调用一个方法：

```
$users = User::where('votes', '>', 500)->get();

$users->each->markAsVip();
```

同样，我们可以使用 `sum` 高阶信息传递的方式来统计出集合中用户总共的「投票数」：

```
$users = User::where('group', 'Development')->get();

return $users->sum->votes;
```

Laravel 的错误和日志记录

- [简介](#)
- [配置](#)
 - [显示错误信息](#)
 - [日志存储](#)
 - [日志等级](#)
 - [自定义 Monolog 设置](#)
- [异常处理](#)
 - [Report 方法](#)
 - [Render 方法](#)
- [HTTP 异常](#)
 - [自定义错误页面](#)
- [记录](#)

简介

当您启动一个新的 Laravel 项目时，错误和异常处理就已为您配置。应用程序触发的所有异常都被 `App\Exceptions\Handler` 类记录下来，然后渲染给用户。我们将在本文档中深入介绍此类。

Laravel 使用功能强大的 [Monolog](#) 库进行日志处理。Laravel 配置了多几种日志处理 handler，方便您在单个日志文件、多个交替日志文件之间进行选择写入或将错误信息写入系统日志。

配置

显示错误信息

`config/app.php` 文件的 `debug` 选项，决定了是否向用户显示错误信息。默认情况下，此选项设置为存储在 `.env` 文件中的 `APP_DEBUG` 环境变量中。

开发环境下，应该将 `APP_DEBUG` 环境变量设置为 `true`。在您的生产环境中，此值应始终为 `false`。如果在生产中将该值设置为 `true`，则可能会将敏感的配置值暴露给应用程序的最终用户。

日志存储

开箱即用，Laravel 支持 `single`、`daily`、`syslog` 和 `errorlog` 日志模式。要配置 Laravel 使用的存储机制，应该修改 `config/app.php` 配置文件中的 `log` 选项。例如，如果您希望使用每日一个日志文件而不是单个文件，则应将 `app` 配置文件中的 `log` 值设置为 `daily`：

```
'log' => 'daily'
```

日志保存天数限制

使用 `daily` 日志模式时，Laravel 将只保留五天默认的日志文件。如果你想调整保留文件的数量，您可以添加一个 `log_max_files` 配置项目到 `APP` 配置文件：

```
'log_max_files' => 30
```

日志等级

使用 Monolog 时，日志消息可能具有不同的日志等级。默认情况下，Laravel 将所有日志级别写入存储。但是，在生产环境中，您可能希望通过将 `log_level` 选项添加到 `app.php` 配置文件中来配置应记录的最低日志等级。

一旦配置了此选项，Laravel 将记录大于或等于指定日志等级的所有级别。例如，默认将 `log_level` 设置为 `error` 那么将会记录 `error`, `critical`, `alert` 和 `emergency` 日志信息：

```
'log_level' => env('APP_LOG_LEVEL', 'error'),
```

{tip} Monolog 识别以下日志等级 - 从低到高为: `debug` , `info` , `notice` , `warning` , `error` , `critical` , `alert` , `emergency` 。

自定义 Monolog 设置

如果你想让你的应用程序完全控制 Monolog，可以使用应用程序的 `configureMonologUsing` 方法。你应该放置一个回调方法到 `bootstrap/app.php` 文件中，在文件返回 `$app` 变量之前，调用这个方法：

```
$app->configureMonologUsing(function ($monolog) {
    $monolog->pushHandler(...);
});

return $app;
```

异常处理

Report 方法

所有异常都由 `App\Exceptions\Handler` 类处理。这个类包含两个方法：`report` 和 `render` 。我们将详细研究这些方法。`report` 方法用于记录异常或将其实发送到外部服务，如 [Bugsnag](#) 或 [Sentry](#)。默认情况下，`report` 方法只是将异常传递给记录异常的基类。然而，你可以自由选择任何方式进行处理。

例如，如果您需要以不同的方式报告不同类型的异常，您可以使用 PHP `instanceof` 比较运算符：

```
/**
 * 报告或记录异常
 *
 * 这是一个很棒的位置向 Sentry , Bugsnag 等发送异常。
 *
 * @param \Exception $exception
 * @return void
 */
public function report(Exception $exception)
{
    if ($exception instanceof CustomException) {
        //
    }

    return parent::report($exception);
}
```

通过类型忽略异常

异常 handler 的 `$dontReport` 属性包含不会记录的异常类型数组。例如，404错误导致的异常以及其他几种类型的错误不会写入您的日志文件。您可以根据需要向此数组添加其他异常类型：

```
/**
 * 不应报告的异常类型列表
 *
 * @var array
 */
protected $dontReport = [
    \Illuminate\Auth\AuthenticationException::class,
    \Illuminate\Auth\Access\AuthorizationException::class,
    \Symfony\Component\HttpKernel\Exception\HttpException::class,
    \Illuminate\Database\Eloquent\ModelNotFoundException::class,
    \Illuminate\Validation\ValidationException::class,
];
```

Render 方法

`render` 方法负责将异常转换成 HTTP 响应发送给浏览器。默认情况下，异常会传递给为您生成响应的基类。但是，您可以自由检查异常类型或返回您自己的自定义响应：

```
/**
 * 渲染异常并添加到 HTTP 响应中。
 *
 * @param \Illuminate\Http\Request $request
 * @param \Exception $exception
 * @return \Illuminate\Http\Response
 */
public function render($request, Exception $exception)
```

```
{
    if ($exception instanceof CustomException) {
        return response()->view('errors.custom', [], 500);
    }

    return parent::render($request, $exception);
}
```

HTTP 异常

一些异常描述了来自服务器的 HTTP 错误代码。例如这可能是「找不到页面」错误（404），「未授权错误」（401）或甚至开发者生成的500错误。你可以使用 `abort` 函数，在应用程序中的任何地方生成这样的响应：

```
abort(404);
```

`abort` 函数将立即创建一个被渲染的异常。此外，您还可以提供响应文本：

```
abort(403, 'Unauthorized action.');
```

自定义错误页面

Laravel 可以轻松地显示各种HTTP状态代码的自定义错误页面。例如，如果您要自定义404 HTTP状态代码的错误页面，请创建一个 `resources/views/errors/404.blade.php`。此文件将会用于渲染所有404错误。此目录中的视图文件命名应与它们对应的HTTP状态代码匹配。由 `abort` 函数引发的 `HttpException` 实例将作为 `$exception` 变量传递给视图。

记录

Laravel 在强大的 [Monolog](#) 库上提供了一个简单的抽象层。默认情况下，Laravel 日志目录为 `storage/logs`。您可以使用 `Log facade` 将信息写入日志：

```
<?php

namespace App\Http\Controllers;

use App\User;
use Illuminate\Support\Facades\Log;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * 显示给定用户的配置文件
     *
```

```

 * @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)]);
}
}

```

该日志记录器提供八种 [RFC 5424](#) 定义的日志级别: emergency , alert , critical, error , warning , notice , info 和 debug 。

```

Log::emergency($message);
Log::alert($message);
Log::critical($message);
Log::error($message);
Log::warning($message);
Log::notice($message);
Log::info($message);
Log::debug($message);

```

上下文信息

将上下文数据以数组格式传递给日志方法。此上下文数据将被格式化并与日志消息一起显示:

```
Log::info('User failed to login.', ['id' => $user->id]);
```

访问底层 Monolog 实例

Monolog 还有多种其他的处理 handler , 你可以用来记录。如果需要, 您可以访问 Laravel 底层的 Monolog 实例:

```
$monolog = Log::getMonolog();
```

Laravel 的事件系统

- [简介](#)
- [注册事件与监听器](#)
 - [生成事件与监听器](#)
 - [手动注册事件](#)
- [定义事件](#)
- [定义监听器](#)
- [队列化事件监听器](#)
 - [手动访问队列](#)
 - [处理失败任务](#)
- [触发事件](#)
- [事件订阅者](#)
 - [编写事件订阅者](#)
 - [注册事件订阅者](#)

简介

Laravel 事件机制实现了一个简单的观察者模式，让我们可以订阅和监听应用中出现的各种事件。事件类 (Event) 类通常保存在 `app/Events` 目录下，而它们的监听类 (Listener) 类被保存在 `app/Listeners` 目录下。如果你在应用中看不到这些文件夹也不要担心，因为当你使用 Artisan 命令来生成事件和监听器时他们会被自动创建。

事件机制是一种很好的应用解耦方式，因为一个事件可以拥有多个互不依赖的监听器。例如，每次把用户的订单发完货后都希望给他发个 Slack 通知。这时候你可以发起一个 `orderShipped` 事件，它会被监听器接收到再传递给 Slack 通知模块，这样你就不用把订单处理的代码跟 Slack 通知的代码耦合在一起了。

注册事件和监听器

Laravel 应用中的 `EventServiceProvider` 提供了一个很方便的地方来注册所有的事件监听器。它的 `listen` 属性是一个数组，包含所有的事件（键）以及事件对应的监听器（值）。你也可以根据应用需求来增加事件到这个数组中。例如，增加一个 `OrderShipped` 事件：

```
/**
 * 应用程序的事件监听器映射。
 *
 * @var array
 */
protected $listen = [
    'App\Events\OrderShipped' => [
        'App\Listeners\SendShipmentNotification',
    ],
]
```

```
];
```

生成事件和监听器

当然，手动创建每个事件和监听器是很麻烦的。简单的方式是，在 `EventServiceProvider` 类中添加好事件和监听器，然后使用 `event:generate` 命令。这个命令会自动生成 `EventServiceProvider` 类中列出的所有事件和监听器。当然已经存在的事件和监听器将保持不变：

```
php artisan event:generate
```

手动注册事件

一般来说，事件必须通过 `EventServiceProvider` 类的 `$listen` 数组进行注册；不过，你也可以在 `EventServiceProvider` 类的 `boot` 方法中注册闭包事件。

```
/**
 * 注册应用程序中的任何其他事件。
 *
 * @return void
 */
public function boot()
{
    parent::boot();

    Event::listen('event.name', function ($foo, $bar) {
        //
    });
}
```

通配符事件监听器

你甚至可以在注册监听器时使用 `*` 通配符参数，它让你在一个监听器中可以监听到多个事件。通配符监听器接受的第一个参数是事件名称，第二个参数是整个的事件数据：

```
Event::listen('event.*', function ($eventName, array $data) {
    //
});
```

定义事件

事件类就是一个包含与事件相关信息数据的容器。例如，假设我们生成的 `orderShipped` 事件接受一个 [Eloquent ORM](#) 对象：

```
<?php
```

```

namespace App\Events;

use App\Order;
use Illuminate\Queue\SerializesModels;

class OrderShipped
{
    use SerializesModels;

    public $order;

    /**
     * 创建一个事件实例。
     *
     * @param Order $order
     * @return void
     */
    public function __construct(Order $order)
    {
        $this->order = $order;
    }
}

```

正如你所见，这个事件类中没有包含其它逻辑。它仅只是一个被构建的 `order` 对象的容器。如果使用 PHP 的 `serialize` 函数对事件进行序列化，使用了 `SerializesModels` trait 的事件将会优雅的序列化任何的 Eloquent 模型。

定义监听器

接下来，让我们看一下例子中事件的监听器。事件监听器在 `handle` 方法中接受了事件实例作为参数。`event:generate` 命令将会在事件的 `handle` 方法中自动加载正确的事件类和类型提示。在 `handle` 方法中，你可以运行任何需要响应该事件的业务逻辑。

```

<?php

namespace App\Listeners;

use App\Events\OrderShipped;

class SendShipmentNotification
{
    /**
     * 创建事件监听器。
     *
     * @return void
     */
    public function __construct()

```

```

{
    //
}

/**
 * 处理事件
 *
 * @param OrderShipped $event
 * @return void
 */
public function handle(OrderShipped $event)
{
    // 使用 $event->order 来访问 order ...
}
}

```

{tip} 你的事件监听器也可以在构造函数中对任何依赖使用类型提示。所有的事件监听器会经由 Laravel 的 [服务容器](#) 做解析，所以所有的依赖都将会被自动注入：

停止事件传播

有时，你可能希望停止一个事件传播到其他的监听器。这时你可以通过在监听器的 `handle` 方法中返回 `false` 来实现。

队列化事件监听器

如果你的监听器中需要实现一些耗时的任务，比如发送邮件或者进行 HTTP 请求，那把它放到队列中处理是非常有用的。在使用队列化监听器之前，一定要在服务器或者本地环境中配置 [队列](#) 并开启一个队列监听器。

要对监听器进行队列化的话，只需增加 `ShouldQueue` 接口到你的监听器类。由 Artisan 命令 `event:generate` 生成的监听器已经将此接口导入到命名空间了，因此你可以直接使用它：

```

<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    //
}

```

就这样！当事件被监听器调用时，事件分发器会使用 Laravel 的 [队列系统](#) 自动将它进行队列化。如果监听器通过队列运行且没有抛出任何异常，则已执行完的任务将会自动从队列中删除。

自定义队列的连接和名称

如果你想要自定义队列的连接和名称，你可以在监听器类中定义 `$connection` 和 `$queue` 属性。

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    /**
     * 队列化任务使用的连接名称。
     *
     * @var string|null
     */
    public $connection = 'sq';

    /**
     * 队列化任务使用的队列名称。
     *
     * @var string|null
     */
    public $queue = 'listeners';
}
```

手动访问队列

如果你需要手动访问底层队列任务的 `delete` 和 `release` 方法，你可以使用 `Illuminate\Queue\InteractsWithQueue` trait 来实现。这个 trait 在生成的监听器中是默认加载的，它提供了这些方法：

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    public function handle(OrderShipped $event)
    {
        if (true) {
```

```
        $this->release(30);  
    }  
}  
}
```

处理失败任务

有时你队列化的事件监听器可能失败了。如果队列监听器任务执行次数超过在工作队列中定义的最大尝试次数，监听器的 `failed` 方法将会被自动调用。`failed` 方法接受事件实例和失败的异常作为参数：

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

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

    public function failed(OrderShipped $event, $exception)
    {
        //
    }
}
```

触发事件

如果要触发事件，你可以传递一个事件实例给 `event` 辅助函数。这个函数将会把事件分发到它所有已经注册的监听器上。因为 `event` 函数是全局可访问的，所以你可以在应用中的任何地方调用它：

```
<?php

namespace App\Http\Controllers;

use App\Order;
use App\Events\OrderShipped;
use App\Http\Controllers\Controller;
```

```

class OrderController extends Controller
{
    /**
     * 将传递过来的订单发货。
     *
     * @param int $orderId
     * @return Response
     */
    public function ship($orderId)
    {
        $order = Order::findOrFail($orderId);

        // 订单的发货逻辑...

        event(new OrderShipped($order));
    }
}

```

{tip} 测试时，不用真的触发监听器就能断言事件类型是很有用的。 Laravel 内置的测试辅助方法能让这件事变得很容易。

事件订阅者

编写事件订阅者

事件订阅者是一个在自身内部可以订阅多个事件的类，允许你在单个类中定义多个事件处理器。订阅者应该定义一个 `subscribe` 方法，这个方法接受一个事件分发器的实例。你可以调用事件分发器的 `listen` 方法来注册事件监听器：

```

<?php

namespace App\Listeners;

class UserEventSubscriber
{
    /**
     * 处理用户登录事件。
     */
    public function onUserLogin($event) {}

    /**
     * 处理用户注销事件。
     */
    public function onUserLogout($event) {}

    /**
     * 为订阅者注册监听器。
     */
}

```

```

    * @param Illuminate\Events\Dispatcher $events
    */
public function subscribe($events)
{
    $events->listen(
        'Illuminate\Auth\Events\Login',
        'App\Listeners\UserEventSubscriber@onUserLogin'
    );

    $events->listen(
        'Illuminate\Auth\Events\Logout',
        'App\Listeners\UserEventSubscriber@onUserLogout'
    );
}

}

```

注册事件订阅者

一旦订阅者被定义，它就可以被注册到事件分发器中。你可以在 `EventServiceProvider` 类的 `$subscribe` 属性注册订阅者。例如，添加 `UserEventSubscriber` 到列表中：

```

<?php

namespace App\Providers;

use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvider
;

class EventServiceProvider extends ServiceProvider
{
    /**
     * 应用中事件监听器的映射。
     *
     * @var array
     */
    protected $listen = [
        //
    ];

    /**
     * 需要注册的订阅者类。
     *
     * @var array
     */
    protected $subscribe = [
        'App\Listeners\UserEventSubscriber',
    ];
}

```


Laravel 的文件系统和云存储功能集成

- [简介](#)
- [配置](#)
 - [公开磁盘](#)
 - [本地驱动](#)
 - [驱动的前置条件](#)
- [获取磁盘实例](#)
- [提取文件](#)
 - [文件 URLs](#)
 - [文件元数据](#)
- [保存文件](#)
 - [文件上传](#)
 - [文件可见性](#)
- [删除文件](#)
- [目录](#)
- [自定义文件系统](#)

简介

Laravel 提供强大文件抽象能力，这得益于 Frank de Jonge 的 [Flysystem](#) 扩展包。Laravel 集成的 `flysystem` 提供了可支持本地文件系统、Amazon S3 及 Rackspace 云存储的简单易用的驱动程序。更棒的是，由于每个系统的API保持不变，所以在这些存储项之间切换是非常轻松的。

配置

文件系统配置文件位于 `config/filesystems.php`。该文件能让你设置所有的「磁盘（disk）」。每个磁盘代表一个特定的存储驱动及存储位置。各种支持驱动的配置示例已包含其中，仅需要简单的根据你的偏好配置及凭证设置进行修改即可。

当然，你可随意配置多组磁盘，即使多个磁盘使用相同的驱动。

公开磁盘

「公开磁盘」就是指你的文件将可被公开访问，默认下，`public` 磁盘使用 `local` 驱动且将文件存放在 `storage/app/public` 目录下。为了能通过网络访问，你需要创建 `public/storage` 到 `storage/app/public` 的符号链接。这个约定能让你的可公开访问文件保持在同一个目录下，这样在不同的部署系统间就可以轻松共享，如 [Envoyer](#) 的“不停服”部署系统。

你可以使用 `storage:link` Artisan 命令创建符号链接：

```
php artisan storage:link
```

当然了，当文件放好且符号链接创建完毕后，你就可以用 `asset` 辅助函数创建 URL 了：

```
echo asset('storage/file.txt');
```

本地驱动

当使用 `local` 驱动时，所有的操作都是相对于你在配置文件中定义的 `root` 目录进行的。该目录默认是 `storage/app`。所以，下面方法会把文件保存在 `storage/app/file.txt`：

```
Storage::disk('local')->put('file.txt', 'Contents');
```

驱动的预先需求

Composer 包

在使用 S3 或 Rackspace 驱动之前，你需要通过 Composer 安装适当扩展包：

- Amazon S3: `league/flysystem-aws-s3-v3 ~1.0`
- Rackspace: `league/flysystem-rackspace ~1.0`

S3 驱动配置

S3 驱动配置信息位于 `config/filesystems.php` 配置文件中。此文件有个关于S3 驱动的配置数组例子。你可根据自己的 S3 配置和凭证修改该数组。

FTP 驱动配置

Laravel 集成的 Flysystem 能很好的支持 FTP，不过 FTP 的配置示例没被包含在框架默认的 `filesystems.php` 文件中，需要的话照着下面的例子配置：

```
'ftp' => [
    'driver'    => 'ftp',
    'host'      => 'ftp.example.com',
    'username'  => 'your-username',
    'password'  => 'your-password',

    // Optional FTP Settings...
    // 'port'      => 21,
    // 'root'      => '',
    // 'passive'   => true,
    // 'ssl'       => true,
    // 'timeout'  => 30,
],
]
```

Rackspace 驱动配置

Laravel 集成的 Flysystem 能很好的支持 Rackspace，不过 Rackspace 的配置示例没被包含在框架默认的 `filesystems.php` 文件中，需要的话照着下面的例子配置：

```
'rackspace' => [
    'driver'      => 'rackspace',
    'username'   => 'your-username',
    'key'         => 'your-key',
    'container'  => 'your-container',
    'endpoint'   => 'https://identity.api.rackspacecloud.com/v2.0/',
    'region'     => 'IAD',
    'url_type'   => 'publicURL',
],
```

获得磁盘实例

`Storage facade` 用于和所有已设置的磁盘交互。例如，你可以调 `facade` 的 `put` 方法将一张头像保存到默认磁盘上。调 `Storage facade` 的方法前若未先调用 `disk` 方法，此方法会被自动传递给默认磁盘。

```
use Illuminate\Support\Facades\Storage;

Storage::put('avatars/1', $fileContents);
```

要是你的应用和多个磁盘交互，你可以通过 `Storage facade` 的 `disk` 方法访问特定的磁盘：

```
Storage::disk('s3')->put('avatars/1', $fileContents);
```

提取文件

`get` 方法被用作提取文件内容，此方法返回该文件的原始字符串内容。切记，所有文件路径都是基于配置文件中 `root` 目录的相对路径。

```
$contents = Storage::get('file.jpg');
```

`exists` 方法可以被用于判断一个文件是否存在与磁盘：

```
$exists = Storage::disk('s3')->exists('file.jpg');
```

文件 URLs

当使用 `local` 或者 `s3` 驱动的时候，你可以使用 `url` 方法来获取给定文件的 URL。如果你使用 `local` 驱动，一般会在传参的路径前面加上 `/storage` 且返回相对路径。如果是 `s3` 的话，返回的是完整的 S3 文件系统的 URL：

```
use Illuminate\Support\Facades\Storage;

$url = Storage::url('file1.jpg');
```

{note} 切记，如果使用 `local` 驱动，所有想被公开访问的文件都应该放在 `storage/app/public` 目录下。此外，你应该在 `public/storage` [创建符号链接] (#the-public-disk) 来指向 `storage/app/public` 文件夹。

定制本地 URL 主机

如果你想给使用 `local` 驱动的存储文件预定义主机的话，你可以在磁盘配置数组中添加 `url` 键：

```
'public' => [
    'driver' => 'local',
    'root' => storage_path('app/public'),
    'url' => env('APP_URL').'/storage',
    'visibility' => 'public',
],
```

文件元数据

除了读写文件，Laravel 还可以提供有关文件本身的信息。例如，`size` 方法可用来获取以字节为单位的文件大小：

```
use Illuminate\Support\Facades\Storage;

$size = Storage::size('file1.jpg');
```

`lastModified` 方法返回的最后一次文件被修改的 UNIX 时间戳：

```
$time = Storage::lastModified('file1.jpg');
```

保存文件

`put` 方法用于保存文件原始内容到一个磁盘上。你也可以传递 PHP 的 `resource` 给 `put` 方法，它将使用 Flysystem 下的 stream 支持。强烈建议使用 streams 处理大型文件：

```
use Illuminate\Support\Facades\Storage;

Storage::put('file.jpg', $contents);

Storage::put('file.jpg', $resource);
```

自动流

如果您想 Laravel 自动管理指定文件流传输到您想要的存储位置，您可以使用 `putFile` 或 `putFileAs` 方法。这个方法可以接受一个 `Illuminate\Http\File` 或 `Illuminate\Http\UploadedFile` 实例，并自动将文件传输到你想要的位置：

```
use Illuminate\Http\File;

// 自动生成唯一文件名...
Storage::putFile('photos', new File('/path/to/photo'));

// 手动指定一个文件名...
Storage::putFileAs('photos', new File('/path/to/photo'), 'photo.jpg');
```

关于 `putFile` 方法有些重要的提醒。我们只指定一个目录名，而非文件名。默认情况下，该 `putFile` 方法将生成以为唯一ID作为文件名。文件的路径将被 `putFile` 方法返回，因此您可以在数据库中存储路径及文件名。

`putFile` 和 `putFileAs` 方法也接受一个参数指定被存储文件的「可见性」。在使用如 S3 的云存储时，若希望该文件可被公开访问，这将非常有用：

```
Storage::putFile('photos', new File('/path/to/photo'), 'public');
```

插入到文件

`prepend` 及 `append` 方法允许你将内容写入到一个文件的开头或结尾：

```
Storage::prepend('file.log', 'Prepended Text');

Storage::append('file.log', 'Appended Text');
```

复制 & 移动文件

`copy` 方法用于复制一个已存在的文件到磁盘的新位置。`move` 方法用于重命名或是移动一个已存在的文件到新位置：

```
Storage::copy('old/file1.jpg', 'new/file1.jpg');

Storage::move('old/file1.jpg', 'new/file1.jpg');
```

文件上传

在 Web 应用中，存储文件最常见的例子之一就是存储用户上传的文件，如个人资料图片，照片和文档。Laravel 通过使用文件上传实例的 `store` 方法，使其可以非常容易的存储上传的文件。只需使用你想存储的路径来调用 `store` 方法即可：

```
<?php
```

```

namespace App\Http\Controllers;

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

class UserAvatarController extends Controller
{
    /**
     * 更新用户头像。
     *
     * @param Request $request
     * @return Response
     */
    public function update(Request $request)
    {
        $path = $request->file('avatar')->store('avatars');

        return $path;
    }
}

```

关于此例有些注意事项。我们只指定一个目录名，而不是文件名。默认情况下，`store` 方法将生成唯一ID来作为文件名。此文件路径将被 `store` 方法返回，因此你可以在数据库中存储路径及文件名。

你也可以调用 `Storage facade` 的 `putFile` 方法来执行和上面例子相同的文件操作：

```
$path = Storage::putFile('avatars', $request->file('avatar'));
```

指定文件名

如果你不喜欢自动生成的文件名，你可以使用 `storeAs` 方法，它接收的路径、文件名、磁盘（可选的）作为它的参数：

```

$path = $request->file('avatar')->storeAs(
    'avatars', $request->user()->id
);

```

当然，你也可以使用 `Storage facade` 的 `putFileAs` 方法，可以和上面例子的文件操作有相同效果：

```

$path = Storage::putFileAs(
    'avatars', $request->file('avatar'), $request->user()->id
);

```

指定磁盘

默认情况下，此方法将使用默认的磁盘。如果你想指定其他磁盘，给 `store` 方法的第二个参数传磁盘名：

```
$path = $request->file('avatar')->store(
    'avatars/' . $request->user()->id, 's3'
);
```

文件可见性

在 Laravel 的 Flysystem 集成里，「可见性」是跨多平台的文件权限抽象。文件可以被设定为 `public` 或 `private`。当一个文件声明为 `public` 时，就意味着文件一般可供他人访问。例如，使用 S3 驱动时，你可检索 `public` 文件的 URL。

你可通过 `put` 方法设定文件可见性：

```
use Illuminate\Support\Facades\Storage;

Storage::put('file.jpg', $contents, 'public');
```

如果文件已经被保存，其可见性可以通过 `getVisibility` 来获取和 `setVisibility` 方法来设置。

```
$visibility = Storage::getVisibility('file.jpg');

Storage::setVisibility('file.jpg', 'public')
```

删除文件

`delete` 方法接受一个文件名称或文件数组，用于从磁盘移除文件：

```
use Illuminate\Support\Facades\Storage;

Storage::delete('file.jpg');

Storage::delete(['file1.jpg', 'file2.jpg']);
```

目录

获取某目录内的所有文件

`files` 方法返回指定目录下的所有文件数组。如果你还想获取指定目录下子目录的文件，可以使用 `allFiles` 方法。

```
use Illuminate\Support\Facades\Storage;
```

```
$files = Storage::files($directory);

$files = Storage::allFiles($directory);
```

获取单个目录内所有目录

`directories` 方法返回指定目录下的目录数组。另外，你也可以使用 `allDirectories` 方法获取指定目录下的子目录以及子目录所包含的目录。

```
$directories = Storage::directories($directory);

// 递归...
$directories = Storage::allDirectories($directory);
```

创建目录

`makeDirectory` 方法将创建指定的目录，包括任何所需的子目录。

```
Storage::makeDirectory($directory);
```

删除目录

最后，`deleteDirectory` 方法删除目录及所包含的全部文件。

```
Storage::deleteDirectory($directory);
```

自定义文件系统

Laravel 的 Flysystem 集成提供一系列开箱即用的驱动支持；然而 Flysystem 不仅限于此，还拥有其它存储系统适配器。如果在你的 Laravel 的应用中想使用额外的存储适配器，你可以创建自定义驱动。

为了建构一个自定义的文件系统，你需要创建一个如 `DropboxServiceProvider` 的 [服务提供者](#)。并在该提供者的 `boot` 方法使用 `Storage facade` 的 `extend` 方法自定义你的驱动。

```
<?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
{
```

```

    /**
     * 运行服务注册后的启动进程。
     *
     * @return void
     */
    public function boot()
    {
        Storage::extend('dropbox', function($app, $config) {
            $client = new DropboxClient(
                $config['accessToken'], $config['clientIdentifier']
            );

            return new Filesystem(new DropboxAdapter($client));
        });
    }

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

```

`extend` 方法的第一个参数是驱动名，第二个参数则是一个接受 `$app` 及 `$config` 变量的闭包。该闭包必须返回 `League\Flysystem\Filesystem` 的实例。`$config` 变量包含了在 `config/filesystems.php` 定义的特定磁盘配置。

一旦通过创建服务提供者注册此扩展后，你就可以在 `config/filesystem.php` 配置文件中使用 `dropbox` 驱动。

Laravel 的辅助函数列表

- [简介](#)
- [可用方法](#)

简介

Laravel 包含有各种各样的 PHP 辅助函数，许多都是在 Laravel 自身框架中使用到。如果你觉得实用，也可以在你自己的应用中使用它们。

可用方法

数组

```
[array_add](#method-array-add) [arrayCollapse](#method-array-collapse) [arrayDivide](#method-array-divide)
[arrayDot](#method-array-dot) [arrayExcept](#method-array-except) [arrayFirst](#method-array-first)
[arrayFlatten](#method-array-flatten) [arrayForget](#method-array-forget)
[arrayGet](#method-array-get) [arrayHas](#method-array-has) [arrayLast](#method-array-last)
[arrayOnly](#method-array-only) [arrayPluck](#method-array-pluck) [arrayPrepend](#method-array-prepend)
[arrayPull](#method-array-pull) [arraySet](#method-array-set) [arraySort](#method-array-sort)
[arraySortRecursive](#method-array-sort-recursive) [arrayWhere](#method-array-where)
[head](#method-head) [last](#method-last)
```

路径

```
[appPath](#method-app-path) [basePath](#method-base-path) [configPath](#method-config-path)
[databasePath](#method-database-path) [mix](#method-mix) [publicPath](#method-public-path)
[resourcePath](#method-resource-path) [storagePath](#method-storage-path)
```

字符串

```
[camelCase](#method-camel-case) [class_basename](#method-class-basename) [e](#method-e)
[endsWith](#method-ends-with) [snakeCase](#method-snake-case) [strLimit](#method-str-limit)
[startsWith](#method-starts-with) [strContains](#method-str-contains) [strFinish](#method-str-finish)
[strIs](#method-str-is) [strPlural](#method-str-plural) [strRandom](#method-str-random)
[strsingular](#method-str-singular) [strSlug](#method-str-slug) [studlyCase](#method-studly-case)
[titleCase](#method-title-case) [trans](#method-trans) [transChoice](#method-trans-choice)
```

URLs

```
[action](#method-action) [asset](#method-asset) [secureAsset](#method-secure-asset) [secureUrl]
(#method-secure-url) [route](#method-route) [url](#method-url)
```

其他

[abort](#method-abort) [abort_if](#method-abort-if) [abort_unless](#method-abort-unless) [auth](#method-auth) [back](#method-back) [bcrypt](#method-bcrypt) [cache](#method-cache) [collect](#method-collect) [config](#method-config) [csrf_field](#method-csrf-field) [csrf_token](#method-csrf-token) [dd](#method-dd) [dispatch](#method-dispatch) [env](#method-env) [event](#method-event) [factory](#method-factory) [info](#method-info) [logger](#method-logger) [method_field](#method-method-field) [old](#method-old) [redirect](#method-redirect) [request](#method-request) [response](#method-response) [retry](#method-retry) [session](#method-session) [value](#method-value) [view](#method-view)

方法列表

数组

`array_add()`

如果给定的键不存在与数组中，`array_add` 就会把给定的键值对添加到数组中：

```
$array = array_add(['name' => 'Desk'], 'price', 100);

// ['name' => 'Desk', 'price' => 100]
```

`array_collapse()`

`array_collapse` 函数把数组里的每一个数组合并成单个数组：

```
$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，一个包含原本数组的键，另一个包含原本数组的值：

```
list($keys, $values) = array_divide(['name' => 'Desk']);

// $keys: ['name']

// $values: ['Desk']
```

array_dot()

`array_dot` 函数把多维数组压制成一维数组，并用「点」式语法表示深度：

```
$array = array_dot(['foo' => ['bar' => 'baz']]);
// ['foo.bar' => 'baz'];
```

array_except()

`array_except` 函数从数组移除指定的键值对：

```
$array = ['name' => 'Desk', 'price' => 100];
$array = array_except($array, ['price']);
// ['name' => 'Desk']
```

array_first()

`array_first` 函数返回数组中第一个通过指定测试的元素：

```
$array = [100, 200, 300];
$value = array_first($array, function ($value, $key) {
    return $value >= 150;
});
// 200
```

可传递第三个参数作为默认值。当没有元素通过测试时，将会返回该默认值：

```
$value = array_first($array, $callback, $default);
```

array_flatten()

`array_flatten` 函数将多维数组压制成一维数组：

```
$array = ['name' => 'Joe', 'languages' => ['PHP', 'Ruby']];
$array = array_flatten($array);
// ['Joe', 'PHP', 'Ruby'];
```

array_forget()

`array_forget` 函数以「点」式语法从深度嵌套的数组中移除指定的键值对：

```
$array = ['products' => ['desk' => ['price' => 100]]];  
  
array_forget($array, 'products.desk');  
  
// ['products' => []]
```

array_get()

`array_get` 函数使用「点」式语法从深度嵌套的数组中获取指定的值：

```
$array = ['products' => ['desk' => ['price' => 100]]];  
  
$value = array_get($array, 'products.desk');  
  
// ['price' => 100]
```

`array_get` 函数同样也接受默认值，如果指定的键找不到时，则返回该默认值：

```
$value = array_get($array, 'names.john', 'default');
```

array_has()

`array_has` 函数使用「点」式语法检查指定的项目是否存在于数组中：

```
$array = ['product' => ['name' => 'desk', 'price' => 100]];  
  
$hasItem = array_has($array, 'product.name');  
  
// true  
  
$hasItems = array_has($array, ['product.price', 'product.discount']);  
  
// false
```

array_last()

`array_last` 函数返回数组中最后一个通过指定测试的元素：

```
$array = [100, 200, 300, 110];
```

```
$value = array_last($array, function ($value, $key) {
    return $value >= 150;
});

// 300
```

array_only()**array_only** 函数从数组返回指定的键值对：

```
$array = ['name' => 'Desk', 'price' => 100, 'orders' => 10];

$array = array_only($array, ['name', 'price']);

// ['name' => 'Desk', 'price' => 100]
```

array_pluck()**array_pluck** 函数从数组拉出一列指定的键值对：

```
$array = [
    ['developer' => ['id' => 1, 'name' => 'Taylor']],
    ['developer' => ['id' => 2, 'name' => 'Abigail']],
];

$array = array_pluck($array, 'developer.name');

// ['Taylor', 'Abigail'];
```

你也可以指定要以什么作为结果列的键名：

```
$array = array_pluck($array, 'developer.name', 'developer.id');

// [1 => 'Taylor', 2 => 'Abigail'];
```

array_prepend()**array_prepend** 函数将元素加到数组的头部：

```
$array = ['one', 'two', 'three', 'four'];

$array = array_prepend($array, 'zero');

// $array: ['zero', 'one', 'two', 'three', 'four']
```

array_pull()

`array_pull` 函数从数组移除指定键值对并返回该键值对：

```
$array = ['name' => 'Desk', 'price' => 100];

$name = array_pull($array, 'name');

// $name: Desk

// $array: ['price' => 100]
```

array_set()

`array_set` 函数使用「点」式语法在深度嵌套的数组中写入值：

```
$array = ['products' => ['desk' => ['price' => 100]]];

array_set($array, 'products.desk.price', 200);

// ['products' => ['desk' => ['price' => 200]]]
```

array_sort()

`array_sort` 函数根据指定闭包的结果排序数组：

```
$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` 函数递归排序数组：

```
$array = [
    [
        'Roman',
        'Taylor',
        'Li',
    ],
    [
        'PHP',
        'Ruby',
        'JavaScript',
    ],
];
$array = array_sort_recursive($array);

/*
[
    [
        'Li',
        'Roman',
        'Taylor',
    ],
    [
        'JavaScript',
        'PHP',
        'Ruby',
    ]
];
*/
```

array_where()

`array_where` 函数使用指定的闭包过滤数组：

```
$array = [100, '200', 300, '400', 500];

$array = array_where($array, function ($value, $key) {
    return is_string($value);
});

// [1 => 200, 3 => 400]
```

head()

`head` 函数返回指定数组的第一个元素：

```
$array = [100, 200, 300];  
  
$first = head($array);  
  
// 100
```

last()

`last` 函数返回指定数组的最后一个元素：

```
$array = [100, 200, 300];  
  
$last = last($array);  
  
// 300
```

路径

app_path()

`app_path` 函数返回 `app` 文件夹的完整路径。你也可以使用 `app_path` 函数生成针对指定文件相对于 `app` 目录的完整路径：

```
$path = app_path();  
  
$path = app_path('Http/Controllers/Controller.php');
```

base_path()

`base_path` 函数返回项目根目录的完整路径。你也可以使用 `base_path` 函数生成针对指定文件相对于项目根目录的完整路径：

```
$path = base_path();  
  
$path = base_path('vendor/bin');
```

config_path()

`config_path` 函数返回 `config` 目录的完整路径:

```
$path = config_path();
```

`database_path()`

`database_path` 函数返回 `database` 目录的完整路径:

```
$path = database_path();
```

`mix()`

`mix` 函数获取带有版本号的 `mix` 文件:

```
mix($file);
```

`public_path()`

`public_path` 函数返回 `public` 目录的完整路径:

```
$path = public_path();
```

`resource_path()`

`resource_path` 函数返回 `resources` 目录的完整路径。你也可以使用 `resource_path` 函数生成针对指定文件相对于 `resources` 目录的完整路径:

```
$path = resource_path();  
  
$path = resource_path('assets/sass/app.scss');
```

`storage_path()`

`storage_path` 函数返回 `storage` 目录的完整路径。你也可以使用 `storage_path` 函数生成针对指定文件相对于 `storage` 目录的完整路径:

```
$path = storage_path();  
  
$path = storage_path('app/file.txt');
```

字符串

`camel_case()`

`camel_case` 函数将指定的字符串转换成 驼峰式命名：

```
$camel = camel_case('foo_bar');  
  
// fooBar
```

`class_basename()`

`class_basename` 函数返回不包含命名空间的类名称：

```
$class = class_basename('Foo\Bar\Baz');  
  
// Baz
```

`e()`

`e` 函数对指定字符串进行 `htmlentities`：

```
echo e('<html>foo</html>');  
  
// &lt;html&gt;foo&lt;/html&gt;
```

`ends_with()`

`ends_with` 函数判断指定字符串结尾是否为指定内容：

```
$value = ends_with('This is my name', 'name');  
  
// true
```

`snake_case()`

`snake_case` 函数将指定的字符串转换成 蛇形命名：

```
$snake = snake_case('fooBar');
```

```
// foo_bar
```

```
str_limit()
```

`str_limit` 函数限制字符串的字符个数，该函数接受一个字符串作为第一个参数，第二个参数为允许的最大字符个数：

```
$value = str_limit('The PHP framework for web artisans.', 7);  
// The PHP...
```

```
starts_with()
```

`starts_with` 函数判断字符串开头是否为指定内容：

```
$value = starts_with('This is my name', 'This');  
// true
```

```
str_contains()
```

`str_contains` 函数判断字符串是否包含有指定内容：

```
$value = str_contains('This is my name', 'my');  
// true
```

你也可以传递数组，来判断字符串是否包含任意指定内容：

```
$value = str_contains('This is my name', ['my', 'foo']);  
// true
```

```
str_finish()
```

`str_finish` 函数添加指定内容到字符串末尾：

```
$string = str_finish('this/string', '/');  
// this/string/
```

str_is()

`str_is` 函数判断指定的字符串是否匹配指定的格式，星号可作为通配符使用：

```
$value = str_is('foo*', 'foobar');

// true

$value = str_is('baz*', 'foobar');

// false
```

str_plural()

`str_plural` 函数把字符串转换成复数形式。该函数目前只支持英文：

```
$plural = str_plural('car');

// cars

$plural = str_plural('child');

// children
```

你可以传入一个整数作为第二个参数，来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` 函数生成指定长度的随机字符串。该函数使用了 PHP 自带的 `random_bytes` 函数：

```
$string = str_random(40);
```

str_singular()

`str_singular` 函数把字符串转换成单数形式。该函数目前只支持英文：

```
$singular = str_singular('cars');

// car
```

str_slug()

`str_slug` 函数根据指定字符串生成 URL 友好的「slug」：

```
$title = str_slug('Laravel 5 Framework', '-');

// laravel-5-framework
```

studly_case()

`studly_case` 函数把指定字符串转换成 首字母大写：

```
$value = studly_case('foo_bar');

// FooBar
```

title_case()

`title_case` 函数把指定字符串转换成 每个单词首字母大写：

```
$title = title_case('a nice title uses the correct case');

// A Nice Title Uses The Correct Case
```

trans()

`trans` 函数根据你的 [本地化文件](#) 翻译指定的语句：

```
echo trans('validation.required');
```

trans_choice()

`trans_choice` 函数根据给定数量来决定翻译指定语句是复数形式还是单数形式：

```
$value = trans_choice('foo.bar', $count);
```

URLs

`action()`

`action` 函数根据指定控制器的方法生成 URL，你不需要传入该控制器的完整命名空间。只需要传入相对于 `App\Http\Controllers` 命名空间的控制器类名：

```
$url = action('HomeController@index');
```

如果该方法接受路由参数，可以作为第二个参数传入：

```
$url = action('UserController@profile', ['id' => 1]);
```

`asset()`

根据当前请求的协议（HTTP 或 HTTPS）生成资源文件的 URL：

```
$url = asset('img/photo.jpg');
```

`secure_asset()`

使用 HTTPS 协议生成资源文件的 URL：

```
echo secure_asset('foo/bar.zip', $title, $attributes = []);
```

`route()`

`route` 函数生成指定路由名称的 URL：

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

如果该路由接受参数，可以作为第二个参数传入：

```
$url = route('routeName', ['id' => 1]);
```

`secure_url()`

`secure_url` 数使用 HTTPS 协议生成指定路径的完整 URL：

```
echo secure_url('user/profile');

echo secure_url('user/profile', [1]);
```

url()

`url` 函数生成指定路径的完整 URL:

```
echo url('user/profile');

echo url('user/profile', [1]);
```

如果没有提供路径参数，将会返回一个 `Illuminate\Routing\UrlGenerator` 实例:

```
echo url()->current();
echo url()->full();
echo url()->previous();
```

其他

abort()

`abort` 函数抛出一个将被异常处理句柄渲染的 HTTP 异常:

```
abort(401);
```

你也可以传入异常的响应消息:

```
abort(401, 'Unauthorized.');
```

abort_if()

`abort_if` 函数如果指定的布尔表达式值为 `true` 则抛出一个 HTTP 异常:

```
abort_if(! Auth::user()->isAdmin(), 403);
```

abort_unless()

`abort_unless` 函数如果指定的布尔表达式值为 `false` 则抛出一个 HTTP 异常:

```
abort_unless(Auth::user()->isAdmin(), 403);
```

`auth()`

`auth` 函数返回一个 `authenticator` 实例，可以使用它来代替 `Auth facade`：

```
$user = auth()->user();
```

`back()`

`back()` 函数生成一个重定向响应让用户返回到之前的位置：

```
return back();
```

`bcrypt()`

`bcrypt` 函数使用 `Bcrypt` 算法哈希指定的数值。你可以使用它代替 `Hash facade`：

```
$password = bcrypt('my-secret-password');
```

`cache()`

`cache` 函数尝试从缓存获取给定 `key` 的值。如果 `key` 不存在则返回默认值：

```
$value = cache('key');

$value = cache('key', 'default');
```

同时，你也可以传递键值对来设置缓存，第二个参数可以指定缓存的过期时间，单位分钟：

```
cache(['key' => 'value'], 5);

cache(['key' => 'value'], Carbon::now()->addSeconds(10));
```

`collect()`

`collect` 函数根据指定的数组生成 [集合](#) 实例：

```
$collection = collect(['taylor', 'abigail']);
```

`config()`

`config` 函数用于获取配置信息的值，配置信息的值可通过「点」式语法访问，其中包含要访问的文件名以及选项名。可传递一个默认值作为第二参数，当配置信息不存在时，则返回该默认值：

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

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

`config` 辅助函数也可以在运行期间，根据指定的键值对设置指定的配置信息：

```
config(['app.debug' => true]);
```

`csrf_field()`

`csrf_field` 函数生成包含 CSRF 令牌内容的 HTML 表单隐藏字段。例如，使用 [Blade 语法](#)：

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

`csrf_token()`

`csrf_token` 函数获取当前 CSRF 令牌的内容：

```
$token = csrf_token();
```

`dd()`

`dd` 函数输出指定变量的值并终止脚本运行：

```
dd($value);

dd($value1, $value2, $value3, ...);
```

如果你不想终止脚本运行，使用 `dump` 函数代替：

```
dump($value);
```

`dispatch()`

`dispatch` 函数把一个新任务推送到 Laravel 的 [任务队列](#) 中：

```
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` 函数派发指定的 [事件](#) 到所属的侦听器：

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

`factory()`

`factory` 函数根据指定类、名称以及数量生成模型工厂构造器（model factory builder）。可用于 [测试](#) 或 [数据填充](#)：

```
$user = factory(App\User::class)->make();
```

`info()`

`info` 函数以 `info` 级别写入日志：

```
info('Some helpful information!');
```

包含上下文数据的数组可以通过第二个参数传递给函数：

```
info('User login attempt failed.', ['id' => $user->id]);
```

`logger()`

`logger` 函数以 `debug` 级别写入日志：

```
logger('Debug message');
```

同时支持传入数组作为参数:

```
logger('User has logged in.', ['id' => $user->id]);
```

如果没有传入参数，则会返回一个 [日志](#) 的实例:

```
logger()->error('You are not allowed here.');
```

method_field()

`method_field` 函数生成模拟各种 HTTP 动作请求的 HTML 表单隐藏字段。例如，使用 [Blade 语法](#):

```
<form method="POST">
    {{ method_field('DELETE') }}
</form>
```

old()

`old` 函数 [获取](#) session 内一次性的历史输入值:

```
$value = old('value');

$value = old('value', 'default');
```

redirect()

`redirect` 函数返回一个 HTTP 重定向响应，如果调用时没有传入参数则返回 `redirector` 实例:

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

return redirect()->route('route.name');
```

request()

`request` 函数返回当前 [请求](#) 实例或获取输入的项目:

```
$request = request();

$value = request('key', $default = null)
```

response()

`response` 函数创建一个 [响应](#) 实例或获取一个 `response` 工厂实例：

```
return response('Hello World', 200, $headers);

return response()->json(['foo' => 'bar'], 200, $headers);
```

retry()

`retry` 函数将会重复调用给定的回调函数，最多调用指定的次数。如果回调函数没有抛出异常并且有值返回，则 `retry` 函数返回该值。如果回调函数抛出异常，`retry` 函数将拦截异常并自动再次调用回调函数，直到调用给定的次数。如果重试次数超出给定次数，拦截的异常将会抛出：

```
return retry(5, function () {
    // Attempt 5 times while resting 100ms in between attempts...
}, 100);
```

session()

`session` 函数可用于获取或设置单个 `session` 项：

```
$value = session('key');
```

你可以通过传递键值对数组给该函数设置 `session` 项：

```
session(['chairs' => 7, 'instruments' => 3]);
```

该函数在没有传递参数时，将返回 `session` 实例：

```
$value = session()->get('key');

session()->put('key', $value);
```

value()

`value` 函数返回指定数值。而当你传递一个 `闭包` 给该函数时，该 `闭包` 将被运行并返回该 `闭包` 的运行结果：

```
$value = value(function() { return 'bar'; });
```

```
view()
```

view 函数获取 视图 实例：

```
return view('auth.login');
```

Laravel 的邮件发送功能

- 简介
 - 驱动前提
- 生成 mailables
- 编写 mailables
 - 配置发送者
 - 配置视图
 - 视图数据
 - 附件
 - 内部附件
- Markdown 格式的邮件
 - 生成 Markdown 格式的邮件
 - 编写 Markdown 格式的邮件
 - 自定义组件
- 发送邮件
 - 队列邮件
- 邮件与本地开发
- 事件

简介

Laravel 基于 [SwiftMailer](#) 函数库提供了一套干净、简洁的邮件 API，Laravel 为 SMTP、Mailgun、SparkPost、Amazon SES、PHP 的 `mail` 函数及 `sendmail` 提供驱动，让你可以快速从本地或云端服务自由地发送邮件。

驱动前提

基于 API 的驱动，例如 Mailgun 和 SparkPost 通常比 SMTP 服务器更简单快速。如果可能，你应该尽可能使用这些驱动。所有的 API 驱动都需要 Guzzle HTTP 函数库，你可以使用 Composer 包管理器安装它：

```
composer require guzzlehttp/guzzle
```

Mailgun 驱动

要使用 Mailgun 驱动，首先必须安装 Guzzle，之后将 `config/mail.php` 配置文件中的 `driver` 选项设置为 `mailgun`。接下来，确认 `config/services.php` 配置文件包含以下选项：

```
'mailgun' => [
    'domain' => 'your-mailgun-domain',
    'secret' => 'your-mailgun-key',
],
```

SparkPost 驱动

要使用 SparkPost 驱动，首先必须安装 Guzzle，之后将 `config/mail.php` 配置文件中的 `driver` 选项设置为 `sparkpost`。接下来，确认 `config/services.php` 配置文件包含以下选项：

```
'sparkpost' => [
    'secret' => 'your-sparkpost-key',
],
```

SES 驱动

要使用 Amazon SES 驱动，必须安装 PHP 的 Amazon AWS SDK。你可以在 `composer.json` 文件的 `require` 段落加入下面这一行并运行 `composer update` 命令：

```
"aws/aws-sdk-php": "~3.0"
```

接下来，将 `config/mail.php` 配置文件中的 `driver` 设置为 `ses`。然后确认 `config/services.php` 配置文件包含下列选项：

```
'ses' => [
    'key' => 'your-ses-key',
    'secret' => 'your-ses-secret',
    'region' => 'ses-region', // e.g. us-east-1
],
```

生成 mailables

在 Laravel 中，每种类型的邮件都代表一个「mailables」对象。这些对象存储在 `app/Mail` 目录中。如果在你的应用中没有看见这个目录，别担心，在首次使用 `make:mail` 命令创建 `mailables` 类时这个目录会被创建，例如：

```
php artisan make:mail OrderShipped
```

编写 mailables

所有的「mailables」类都在其 `build` 方法中完成配置。在这个方法内，你可以调用其他各种方法，如 `from`、`subject`、`view` 和 `attach` 来配置完成邮件的详情。

配置发送者

使用 `from` 方法

首先，演示配置邮件的发送者，也就是邮件的参数「from」，既谁发送了邮件。有两种方法配置发送者。第一种是你可以在 `build` 方法中使用 `from` 方法：

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->from('example@example.com')
        ->view('emails.orders.shipped');
}
```

使用一个全局 `from` 地址

然而，如果应用使用相同的 `from` 地址，你每次发送邮件这种设置方式显得笨拙，替代的方法就是在 `config/mail.php` 配置文件中设置一个全局 `from` 地址，这个配置在没有单独指定「from」时是默认的 `from`：

```
'from' => ['address' => 'example@example.com', 'name' => 'App Name'],
```

配置视图

在 `build` 方法内，你可以使用 `view` 方法指定邮件的模板，以渲染邮件的内容。因为所有邮件都会使用 [Blade 模板](#) 渲染内容，你能很容易的使用 Blade 模板引擎构建邮件的 HTML：

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped');
}
```

{tip} 你可以创建一个 `resources/views/emails` 目录来存放所有的邮件模板；然而，这不是强制要求，你可以在有的将邮件模板放在 `resources/views` 目录的任意位置。

纯文本邮件

如果你想要定义一个纯文本邮件，你可以使用 `text` 方法。如同 `view` 方法，`text` 方法接受一个模板名称，这个名称告诉方法使用哪个模板来渲染邮件，可以自由定义邮件 HTML 和纯文本消息：

```
/**
```

```

 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
        ->text('emails.orders.shipped_plain');
}

```

视图数据

通过公共属性

通常，你会希望传递一些数据来渲染你的邮件的 HTML。那么有两种方法可以让视图获得数据，第一种，`mailable` 类任何公共属性都可以在视图中使用。所以，例如你可以传递数据到 `mailable` 类的构造函数并且在类中定义数据的公共属性：

```

<?php

namespace App\Mail;

use App\Order;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * order 实例。
     *
     * @var Order
     */
    public $order;

    /**
     * 创建一个新消息实例。
     *
     * @return void
     */
    public function __construct(Order $order)
    {
        $this->order = $order;
    }

    /**

```

```

 * 构建消息。
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped');
}
}

```

一旦数据被设置为公共属性，它们将自动在视图中加载，所以你可以像访问其他 Blade 模板数据一样访问它们：

```

<div>
    Price: {{ $order->price }}
</div>

```

通过 `with` 方法：

你可以使用 `with` 方法来传递数据给模板。一般情况下，你仍然是使用 `mailable` 类的构造函数来接受数据传参。然而你需要为这些数据属性设置 `protected` 或 `private` 声明，否则这些数据会被自动加载到模板中。接下来你可以使用 `with` 方法接受键值数组传参来传递数据给模板，就如控制器里为视图传参一样：

```

<?php

namespace App\Mail;

use App\Order;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * order 实例。
     *
     * @var Order
     */
    protected $order;

    /**
     * 创建一个新消息实例。
     *
     * @return void
     */
}

```

```

    */
public function __construct(Order $order)
{
    $this->order = $order;
}

/**
 * 构建消息。
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
        ->with([
            'orderName' => $this->order->name,
            'orderPrice' => $this->order->price,
        ]);
}
}

```

一旦数据已经用 `with` 方法传递，它们将自动在视图中加载，所以你可以访问像访问其他 Blade 模板数据一样访问它们：

```

<div>
    Price: {{ $orderPrice }}
</div>

```

附件

要在邮件中加入附件，在 `build` 方法中使用 `attach` 方法。`attach` 方法接受文件的绝对路径作为它的第一个参数：

```

/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
        ->attach('/path/to/file');
}

```

附加文件到消息时，你也可以传递 `数组` 给 `attach` 方法作为第二个参数，以指定显示名称和 / 或是 MIME 类型：

```

/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
        ->attach('/path/to/file', [
            'as' => 'name.pdf',
            'mime' => 'application/pdf',
        ]);
}

```

原始数据附件

`attachData` 可以使用字节数据作为附件。例如，你可以使用这个方法将内存中生成而没有保存到磁盘中的 PDF 附加到邮件中。`attachData` 方法第一个参数接收原始字节数据，第二个参数为文件名，第三个参数接受一个数组以指定其他参数：

```

/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
        ->attachData($this->pdf, 'name.pdf', [
            'mime' => 'application/pdf',
        ]);
}

```

内部附件

在邮件中嵌入内部图标通常是件麻烦事；然而 Laravel 提供了一个便利的方法，让你在邮件中附件图片并获取适当的 CID。要嵌入内部图片，在邮件模板的 `$message` 变量上调用 `embed` 方法，Laravel 会自动让你所有邮件模板都能够获取到 `$message` 变量，所以不必担心如何手动传递它：

```

<body>
    这是一张图片：

    
</body>

```

嵌入原始数据附件

如果你已经有想嵌入原始数据，希望嵌入邮件模板，你可以调用 `$message` 变量的 `embedData` 方法：

```
<body>
    这是一张原始数据图片：

    
</body>
```

Markdown 格式的 Mailables 类

Markdown 格式的 mailable 消息允许你从预编译的模板和你的 mailables 类中的邮件提醒组件中受益。因为消息是用 Markdown 格式写的， Laravel 能为消息体渲染出漂亮、响应式的 HTML 模板，也能自动生成一个纯文本的副本。

生成 Markdown 格式的 Mailables

要生成一个包含友好的 Markdown 模板的 mailable 类，你在使用 `make:mail` 这个 Artisan 命令时，要加上 `--markdown` 选项：

```
php artisan make:mail OrderShipped --markdown=emails.orders.shipped
```

然后，在使用 `build` 方法配置 mailable 时，用 `markdown` 方法来换掉 `view` 方法，`markdown` 方法接受一个 Markdown 模板的名称和一个将在模板中可用的选项数组：

```
/**
 * 构建消息。
 *
 * @return $this
 */
public function build()
{
    return $this->from('example@example.com')
        ->markdown('emails.orders.shipped');
}
```

编写 Markdown 格式的消息

Markdown mailables 使用 Blade 组件和 Markdown 语法的组合，允许你轻松地构建邮件消息，同时利用 Laravel 的预制组件。

```
@component('mail::message')
# Order Shipped

Your order has been shipped!
```

```

@component('mail::button', ['url' => $url])
View Order
@endcomponent

Thanks, <br>
{{ config('app.name') }}
@endcomponent

```

{tip} 当你在写 Markdown 格式邮件的时候千万不要有多余缩进，不然 Markdown 解析器很容易会把缩进的内容渲染成代码块。

按钮组件

按钮连组件渲染一个居中的连接按钮，组件接受两个参数，一个 `url` 和一个可选的 `color`。支持的颜色有 `blue`、`green`、和 `red`。你可以在邮件消息体中加入随便多个你想要的按钮。

```

@component('mail::button', ['url' => $url, 'color' => 'green'])
View Order
@endcomponent

```

面板组件

面板组件将面板中给定的一块文字的背景渲染成跟其他的消息体背景稍微不同。这样可以让这块文字引起人们的注意。

```

@component('mail::panel')
This is the panel content.
@endcomponent

```

表格组件

表格组件允许你将一个 Markdown 格式的表格转换成一个 HTML 格式的表格。组件接受 Markdown 格式的表格作为它的内容。表格列对齐方式按照默认的 Markdown 对齐风格而定。

```

@component('mail::table')
| Laravel | Table | Example |
| ----- | :-----: | -----: |
| Col 2 is | Centered | $10 |
| Col 3 is | Right-Aligned | $20 |
@endcomponent

```

自定义组件

你或许会为了自定义而导出你应用中所有的 Markdown 邮件组件。要导出这些组件，使用 `vendor:publish` 这个 Artisan 命令来发布资源文件标签。

```
php artisan vendor:publish --tag=laravel-mail
```

这个命令会发布 Markdown 邮件组件到 `resources/views/vendor/mail` 文件夹。而 `mail` 文件夹会包含一个 `html` 文件夹和一个 `markdown` 文件夹，每个文件夹都包含他们的可用组件的描述。你可以按照你的意愿自定义这些组件。

自定义样式表 CSS

在导出组件之后，`resources/views/vendor/mail/html/themes` 文件夹将包含一个默认的 `default.css` 文件。你可以在这个文件中自定义 CSS，你定义的这些样式将会在 Markdown 格式消息体转换成 HTML 格式时自动得到应用。

{tip} 如果你想为 Markdown 组件构建一个全新的主题，只要写一个新的 CSS 文件，放在 `html/themes` 文件夹，然后在你的 `mail` 配置文件中改变 `theme` 选项就可以了。

发送邮件

要发送邮件，使用 `Mail facade` 的 `to` 方法。`to` 方法接受一个邮件地址，一个 `user` 实现或一个 `users` 集合。如果传递一个对象或集合，mailer 将自动使用 `email` 和 `name` 属性来设置邮件收件人，所以确保你的对象里有这些属性。一旦指定收件人，你可以传递一个实现到 `mailable` 类的 `send` 方法：

```
<?php

namespace App\Http\Controllers;

use App\Order;
use App\Mail\OrderShipped;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Mail;
use App\Http\Controllers\Controller;

class OrderController extends Controller
{
    /**
     * Ship the given order.
     *
     * @param Request $request
     * @param int $orderId
     * @return Response
     */
    public function ship(Request $request, $orderId)
    {
        $order = Order::findOrFail($orderId);

        // Ship order...
    }
}
```

```

        Mail::to($request->user())->send(new OrderShipped($order));
    }
}

```

当然，不局限于只使用「to」给收件人发送邮件，你可以通过一个单一的链式调用来自由的设置「to」，「cc」和「bcc」接收者：

```

Mail::to($request->user())
->cc($moreUsers)
->bcc($evenMoreUsers)
->send(new OrderShipped($order));

```

队列 Mail

将邮件消息加入队列

由于发送消息会大幅延迟应用响应时间，许多开发者选择将邮件消息加入队列在后台进行发送。 Laravel 使用内置的 [统一的队列 API](#) 来轻松完成此工作。将邮件消息加入队列，使用 `Mail facade` 的 `queue` 方法：

```

Mail::to($request->user())
->cc($moreUsers)
->bcc($evenMoreUsers)
->queue(new OrderShipped($order));

```

这个方法会自动将工作加入队列，以便后台发送邮件。当然，在使用这个功能前，你需要 [设置你的队列](#)。

延迟邮件消息队列

如果你希望延迟发送已加入队列的邮件消息，你可以使用 `later` 方法。`later` 第一个参数接受一个 `DateTime` 实现以告知这个消息应该何时发送：

```

$when = Carbon\Carbon::now()->addMinutes(10);

Mail::to($request->user())
->cc($moreUsers)
->bcc($evenMoreUsers)
->later($when, new OrderShipped($order));

```

推送到特定队列

因为所有 `mailable` 类是通过 `make:mail` 命令生成并使用 `Illuminate\Bus\Queueable trait`，你可以在任何 `mailable` 类实现中调用 `onQueue` 来指定队列名称，还有 `onConnection` 方法来指定队列链接名称：

```
$message = (new OrderShipped($order))
    ->onConnection('sq')
    ->onQueue('emails');

Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->queue($message);
```

默认队列

如果你的 `mailable` 类想要默认使用队列，你可以在类中实现 `ShouldQueue` 接口契约。现在，即便你调用 `send` 方法来发送邮件，`mailable` 类仍将邮件放入队列中发送。

```
use Illuminate\Contracts\Queue\ShouldQueue;

class OrderShipped extends Mailable implements ShouldQueue
{
    //
}
```

邮件和本地开发

当应用开发发送邮件时，你或许不想真实的发送邮件到真实的邮件地址。Laravel 提供几种方法以在本地开发时真实发送「失去能力」。

日志驱动

代替真实发送，`log` 邮件驱动将所有邮件消息写入日志文件以供检查，需要更多根据环境来设置应用程序的信息，可参考 [配置文件](#)。

通用收件者

另一个由 Laravel 提供的解决方案是设置一个通用邮件接收者，由框架发送邮件。这个方法将所有邮件发送到一个邮件地址，而不是发送给实际收件人。这可以通过 `config/mail.php` 配置文件的 `to` 选项来完成：

```
'to' => [
    'address' => 'example@example.com',
    'name' => 'Example'
],
```

Mailtrap

最后，你可以使用像 [Mailtrap](#) 和 `smtp` 驱动来将你的邮件消息发送到一个「虚假的」邮箱中，你却可以在一个真实的邮件客户端中查看它们。这个方法的好处是让你可以在 Mailtrap 的消息阅读器中查看最终的实际邮件。

事件

Laravel 会在发送邮件消息之前触发一个事件。切记，这个事件只会在邮件发送时触发，在加入队列时不触发。你可以在你的 `EventServiceProvider` 注册一个事件监听器：

```
/**
 * 应用事件监听映射。
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Mail\Events\MessageSending' => [
        'App\Listeners\LogSentMessage',
    ],
];
```

Laravel 的消息通知系统

- 简介
- 创建通知
- 发送通知
 - 使用 Notifiable Trait
 - 使用 Notification Facade
 - 指定发送频道
 - 队列化通知
- 邮件通知
 - 格式化邮件消息
 - 自定义接受者
 - 自定义主题
 - 自定义模板
- Markdown 邮件通知
 - 生成消息
 - 写消息
 - 自定义组件
- 数据库通知
 - 先决条件
 - 格式化数据库通知
 - 访问通知
 - 标为已读
- 广播通知
 - 先决条件
 - 格式化广播通知
 - 监听通知
- 短信通知
 - 先决条件
 - 格式化短信通知
 - 自定义 `From` 号码
 - 路由短信通知
- Slack 通知
 - 先决条件
 - 格式化 Slack 通知
 - Slack Attachments
 - 路由 Slack 通知
- 通知事件
- 自定义发送频道

简介

除了[发送邮件](#)，Laravel 还支持通过多种频道发送通知，包括邮件、短信（通过[Nexmo](#)）以及[Slack](#)。通知还能存到数据库，这样就能在网页界面上显示了。

通常情况下，通知应该是简短、有信息量的消息来通知用户你的应用发生了什么。举例来说，如果你在编写一个在线交易应用，你应该会通过邮件和短信频道来给用户发送一条「账单已付」的通知。

创建通知

Laravel 中一条通知就是一个类（通常存在 `app/Notifications` 文件夹里）。看不到的话不要担心，运行一下 `make:notification` 命令就能创建了：

```
php artisan make:notification InvoicePaid
```

这个命令会在 `app/Notifications` 目录下生成一个新的通知类。这个类包含 `via` 方法和几个消息构建方法（比如 `toMail` 或 `toDatabase`），它们会针对指定的渠道把通知转换为对应的消息。

发送通知

使用 Notifiable Trait

通知可以通过两种方法发送：`Notifiable` trait 的 `notify` 方法或 `Notification facade`。首先，让我们探索使用 trait：

```
<?php

namespace App;

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

class User extends Authenticatable
{
    use Notifiable;
}
```

默认的 `App\User` 模型中使用了这个 trait，它包含着一个可以用来发通知的方法：`notify`。
`notify` 方法需要一个通知实例做参数：

```
use App\Notifications\InvoicePaid;

$user->notify(new InvoicePaid($invoice));
```

{tip} 记住，你可以在任意模型中使用 `Illuminate\Notifications\Notifiable` trait，而不仅仅是在 `User` 模型中。

使用 Notification Facade

另外，你可以通过 `Notification facade` 来发送通知。它主要用在当你给多个可接收通知的实体发送通知的时候，比如给用户集合发通知。要用 `facade` 发送通知的话，要把可接收通知的实体和通知的实例传递给 `send` 方法：

```
Notification::send($users, new InvoicePaid($invoice));
```

指定发送频道

每个通知类都有个 `via` 方法，它决定了通知在哪个频道上发送。开箱即用的通知频道有 `mail`，`database`，`broadcast`，`nexmo`，和 `slack`。

{tip} 如果你想用其他的频道比如 Telegram 或者 Pusher，可以去看下社区驱动的 [Laravel 通知频道网站](#)。

`via` 方法受到一个 `$notifiable` 实例，它是接收通知的类实例。你可以用 `$notifiable` 来决定通知用哪个频道来发送：

```
/**
 * 获取通知发送频道
 *
 * @param mixed $notifiable
 * @return array
 */
public function via($notifiable)
{
    return $notifiable->prefers_sms ? ['nexmo'] : ['mail', 'database'];
}
```

队列化通知

{note} 在队列化通知前你需要配置队列，并[运行队列处理器](#)。

发送通知可能会花很长时间，尤其是发送频道需要调用外部 API 的时候。要加速应用响应的话，可以通过添加 `ShouldQueue` 接口和 `Queueable` trait 把通知加入队列。它们两个在使用 `make:notification` 命令来生成通知文件的时候就已经被导入了，所以你只需要添加到你的通知类就行了：

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Notifications\Notification;
use Illuminate\Contracts\Queue\ShouldQueue;
```

```
class InvoicePaid extends Notification implements ShouldQueue
{
    use Queueable;

    // ...
}
```

一旦加入 `ShouldQueue` 接口，你就能像平常那样发送通知了。Laravel 会检测 `ShouldQueue` 接口并自动将通知的发送放入队列中。

```
$user->notify(new InvoicePaid($invoice));
```

如果你想延迟发送，你可以通过 `delay` 方法来链式操作你的通知实例：

```
$when = Carbon::now()->addMinutes(10);

$user->notify((new InvoicePaid($invoice))->delay($when));
```

邮件通知

格式化邮件消息

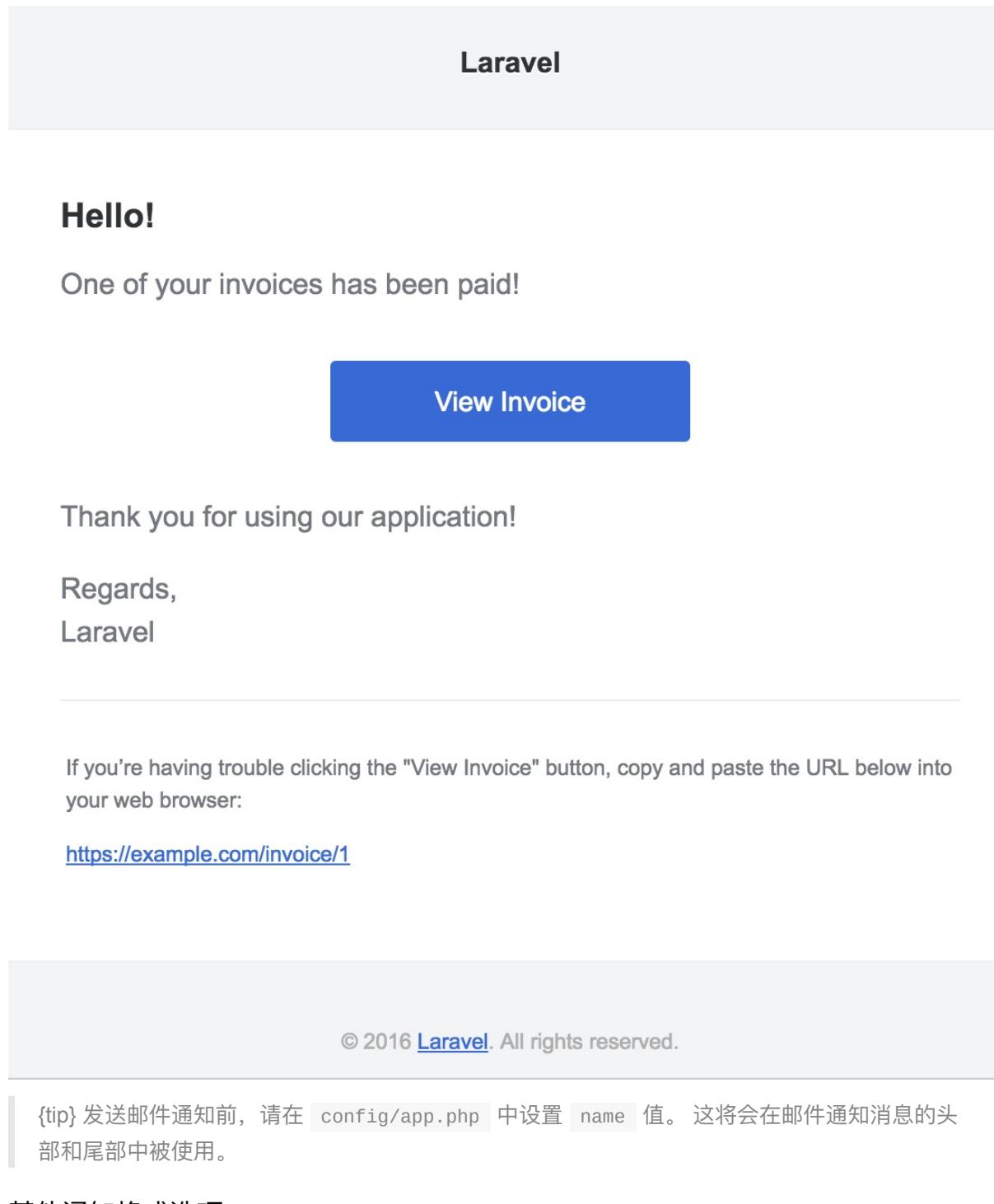
如果一条通知支持以邮件发送，你应该在通知类里定义一个 `toMail` 方法。这个方法将收到一个 `$notifiable` 实体并返回一个 `\Illuminate\Notifications\Messages\MailMessage` 实例。邮件消息可以包含多行文本也可以是引导链接。我们来看一个 `toMail` 方法的例子：

```
/**
 * 获取通知的邮件展示方式
 *
 * @param mixed $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    $url = url('/invoice/'.$this->invoice->id);

    return (new MailMessage)
        ->line('One of your invoices has been paid!')
        ->action('View Invoice', $url)
        ->line('Thank you for using our application!');
}
```

{tip} 注意我们在方法中用了 `$this->invoice->id`，其实你可以传递应用所需要的任何数据来传递给通知的构造器。

在这个例子中，我们注册了一行文本，引导链接，然后又是一行文本。`MailMessage` 提供的这些方法简化了对小的事务性的邮件进行格式化操作。邮件频道将会把这些消息组件转换成漂亮的响应式的 HTML 邮件模板并附上文本。下面是个 `mail` 频道生成的邮件示例：



{tip} 发送邮件通知前，请在 `config/app.php` 中设置 `name` 值。这将会在邮件通知消息的头部和尾部中被使用。

其他通知格式选项

你可以使用 `view` 方法来指定一个应用于渲染通知电子邮件的自定义模板，而不是在通知类中定义文本的「模板」：

```
/**
```

```

 * 获取通知的邮件展示方式
 *
 * @param mixed $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)->view(
        'emails.name', ['invoice' => $this->invoice]
    );
}

```

另外，你可以从 `toMail` 方法中返回一个 `mailable` 对象：

```

use App\Mail\InvoicePaid as Mailable;

/**
 * 获取通知的邮件展示方式
 *
 * @param mixed $notifiable
 * @return Mailable
 */
public function toMail($notifiable)
{
    return (new Mailable($this->invoice))->to($this->user->email);
}

```

错误消息

有些通知是给用户提示错误，比如账单支付失败的提示。你可以通过调用 `error` 方法来指定这条邮件消息被当做一个错误提示。当邮件消息使用了 `error` 方法后，引导链接按钮会变成红色而非蓝色：

```

/**
 * 获取通知的邮件展示方式
 *
 * @param mixed $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)
        ->error()
        ->subject('Notification Subject')
        ->line('...');
}

```

自定义接收者

当通过 `mail` 频道来发送通知的时候，通知系统将会自动寻找你的 `notifiable` 实体中的 `email` 属性。你可以通过在实体中定义 `routeNotificationForMail` 方法来自定义邮件地址。

```
<?php

namespace App;

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

class User extends Authenticatable
{
    use Notifiable;

    /**
     * 邮件频道的路由
     *
     * @return string
     */
    public function routeNotificationForMail()
    {
        return $this->email_address;
    }
}
```

自定义主题

默认情况下，邮件主题是格式化成了标题格式的通知类的类名。所以如果你对通知类名为 `InvoicePaid`，邮件主题将会是 `Invoice Paid`。如果你想显式指定消息的主题，你可以在构建消息时调用 `subject` 方法：

```
/**
 * 获取通知的邮件展示方式
 *
 * @param mixed $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)
        ->subject('Notification Subject')
        ->line('...');

}
```

自定义模板

你可以通过发布通知包的资源来修改 HTML 模板和纯文本模板。运行这个命令后，邮件通知模板就被放在了 `resources/views/vendor/notifications` 文件夹下：

```
php artisan vendor:publish --tag=laravel-notifications
```

Markdown 邮件通知

Markdown 邮件通知可让您利用邮件通知的预先构建的模板，同时给予您更多自由地撰写更长的自定义邮件。由于消息是用 Markdown 编写的， Laravel 能够为消息呈现漂亮，响应的 HTML 模板，同时还自动生成纯文本对应。

生成消息

要使用相应的 Markdown 模板生成通知，您可以使用 `make: notification` Artisan 命令的 `--markdown` 选项：

```
php artisan make:notification InvoicePaid --markdown=mail.invoice.paid
```

与所有其他邮件通知一样，使用 Markdown 模板的通知应在其通知类上定义一个 `toMail` 方法。但是，不使用 `line` 和 `action` 方法来构造通知，而是使用 `markdown` 方法来指定应该使用的 Markdown 模板的名称：

```
/**
 * 获取通知的邮件展示方式
 *
 * @param mixed $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    $url = url('/invoice/'.$this->invoice->id);

    return (new MailMessage)
        ->subject('Invoice Paid')
        ->markdown('mail.invoice.paid', ['url' => $url]);
}
```

写消息

Markdown 邮件通知使用 Blade 组件和 Markdown 语法的组合，允许您轻松构建通知，同时利用 Laravel 的预制通知组件：

```
@component('mail::message')
# Invoice Paid
```

```
Your invoice has been paid!

@Component('mail::button', ['url' => $url])
View Invoice
@endcomponent

Thanks,<br>
{{ config('app.name') }}
@endcomponent
```

按钮组件

按钮组件渲染了一个居中的链接按钮。组件有两个参数，`url` 和一个可选的参数 `color`。支持的颜色有 `blue`，`green`，和 `red`。你可以根据你的需要向通知中添加任意数量的按钮组件：

```
@Component('mail::button', ['url' => $url, 'color' => 'green'])
View Invoice
@endcomponent
```

面板组件

面板部件在面板中呈现给定文本块，该面板具有与通知的其余部分稍微不同的背景颜色。这将会允许你将给定的文本块引起注意：

```
@Component('mail::panel')
This is the panel content.
@endcomponent
```

表格组件

表格组件允许你将一个 Markdown 表格转化成一个 HTML 表格。表格组件接受 Markdown 表格作为内容。支持使用默认的 Markdown 表格对齐语法使表列对齐：

```
@Component('mail::table')
| Laravel      | Table          | Example   |
| -----:|:-----:|-----:|
| Col 2 is    | Centered      | $10       |
| Col 3 is    | Right-Aligned | $20       |
@endcomponent
```

自定义组件

您可以将所有 Markdown 通知组件导出到您自己的应用程序中进行自定义。要导出组件，请使用 `vendor: publish` Artisan 命令来发布 `laravel-mail` 标签：

```
php artisan vendor:publish --tag=laravel-mail
```

此命令将 Markdown 邮件组件发布到 `resources/views/vendor/mail` 目录。`mail` 目录将包含一个 `html` 和一个 `markdown` 目录，每个目录包含它们各自可用组件的展示方式。您可以随意自定义这些组件。

自定义CSS

导出组件后，`resources/views/vendor/mail/html/themes` 目录将包含一个 `default.css` 文件。您可以在此文件中自定义 CSS，并且您的样式将自动内嵌在 Markdown 通知的HTML展示中。

{tip} 如果您想为 Markdown 组件构建一个全新的主题，只需在 `html/themes` 目录中写一个新的 CSS 文件，并更改 `mail` 配置文件中的 `theme` 选项。

数据库通知

先决条件

`database` 通知频道在一张数据表里存储通知信息。这张表包含了比如通知类型、JSON 格式数据等描述通知的信息。

你可以查询这张表的内容在应用界面上展示通知。但是在这之前，你需要先创建一个数据表来保存通知。你可以用 `notifications:table` 命令来生成迁移表：

```
php artisan notifications:table
php artisan migrate
```

格式化数据库通知

如果通知支持被存储到数据表中，你应该在通知类中定义一个 `toDatabase` 或 `toArray` 方法。这个方法接收 `$notifiable` 实体参数并返回一个普通的 PHP 数组。这个返回的数组将被转成 JSON 格式并存储到通知数据表的 `data` 列。我们来看一个 `toArray` 的例子：

```
/**
 * 获取通知的数组展示方式
 *
 * @param mixed $notifiable
 * @return array
 */
public function toArray($notifiable)
{
    return [
        'invoice_id' => $this->invoice->id,
        'amount' => $this->invoice->amount,
    ];
}
```

toDatabase Vs toArray

`toArray` 方法在 `broadcast` 频道也用到了，它用来决定广播给 JavaScript 客户端的数据。如果你想在 `database` 和 `broadcast` 频道中采用两种不同的数组展示方式，你应该定义 `toDatabase` 方法而非 `toArray` 方法。

访问通知

一旦通知被存到数据库中，你需要一种方便的方式来从通知实体中访问它们。

`Illuminate\Notifications\Notifiable` trait 包含一个可以返回这个实体所有通知的 `notifications` Eloquent 关联。要获取这些通知，你可以像用其他 Eloquent 关联一样来使用这个方法。默认情况下，通知将会以 `created_at` 时间戳来排序：

```
$user = App\User::find(1);

foreach ($user->notifications as $notification) {
    echo $notification->type;
}
```

如果你只想检索未读通知，你可以使用 `unreadNotifications` 关联。检索出来的通知也是以 `created_at` 时间戳来排序的：

```
$user = App\User::find(1);

foreach ($user->unreadNotifications as $notification) {
    echo $notification->type;
}
```

{tip} 要从 JavaScript 客户端来访问通知的话，你应该定义一个通知控制器来给可知的实体返回通知，比如给当前用户返回通知。然后你就可以在 JavaScript 客户端来发起对应 URI 的 HTTP 请求了。

标记为已读

通常情况下，当用户查看了通知时，你就希望把通知标为已读。`Illuminate\Notifications\Notifiable` trait 提供了一个 `markAsRead` 方法，它能在对应的数据库记录里更新 `read_at` 列：

```
$user = App\User::find(1);

foreach ($user->unreadNotifications as $notification) {
    $notification->markAsRead();
}
```

你可以使用 `markAsRead` 方法直接操作一个通知集合，而不是一条条遍历每个通知：

```
$user->unreadNotifications->markAsRead();
```

你可以用批量更新的方式来把所有通知标为已读，而不用在数据库里检索：

```
$user = App\User::find(1);

$user->unreadNotifications()->update(['read_at' => Carbon::now()]);
```

当然，你可以通过 `delete` 通知来把它们从数据库删除：

```
$user->notifications()->delete();
```

广播通知

先决条件

在广播通知前，你应该配置并熟悉 Laravel [事件广播](#) 服务。事件广播提供了一种 JavaScript 客户端响应服务端 Laravel 事件的机制。

格式化广播通知

`broadcast` 频道使用 Laravel [事件广播](#) 服务来广播通知，它使得 JavaScript 客户端可以实时捕捉通知。如果一条通知支持广播，你应该在通知类里定义一个 `toBroadcast` 或 `toArray` 方法。这个方法将收到一个 `$notifiable` 实体并返回一个普通的 PHP 数组。返回的数组会被编码成 JSON 格式并广播给你的 JavaScript 客户端。我们来看个 `toArray` 方法的例子：

```
use Illuminate\Notifications\Messages\BroadcastMessage;

/**
 * 获取通知的数组展示方式
 *
 * @param mixed $notifiable
 * @return BroadcastMessage
 */
public function toBroadcast($notifiable)
{
    return new BroadcastMessage([
        'invoice_id' => $this->invoice->id,
        'amount' => $this->invoice->amount,
    ]);
}
```

广播队列配置

所有广播通知都排队等待广播。如果要配置用于广播队列操作的队列连接或队列名称，你可以使用 `BroadcastMessage` 的 `onConnection` 和 `onQueue` 方法：

```
return new BroadcastMessage($data)
    ->onConnection('sq')
    ->onQueue('broadcasts');
```

{tip} 除了你指定的数据外，广播通知也包含一个 `type` 字段，这个字段包含了通知类的类名。

监听通知

通知将会在一个私有频道里进行广播，频道格式为 `{notifiable}.{id}`。所以，如果你给 ID 为 `1` 的 `App\User` 实例发送通知，这个通知就在 `App.User.1` 私有频道里被发送。当你使用 [Laravel Echo](#) 的时候，你可以很简单地使用 `notification` 辅助函数来监听一个频道的通知：

```
Echo.private('App.User.' + userId)
.notification((notification) => {
    console.log(notification.type);
});
```

自定义通知通道

如果您想自定义应通知实体接收其广播通知的渠道，您可以定义一个 `receiveBroadcastNotificationsOn` 方法：

```
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * 用户接收的通知广播
     *
     * @return array
     */
    public function receiveBroadcastNotificationsOn()
    {
        return [
            new PrivateChannel('users.'.$this->id),
        ];
    }
}
```

```

    }
}
```

短信通知

先决条件

在 Laravel 中发送短信通知是基于 [Nexmo](#) 服务的。在通过 Nexmo 发送短信通知前，你需要安装 `nexmo/client` Composer 包并在 `config/services.php` 配置文件中添加几个配置选项。你可以复制下面的配置示例来开始使用：

```
'nexmo' => [
    'key' => env('NEXMO_KEY'),
    'secret' => env('NEXMO_SECRET'),
    'sms_from' => '15556666666',
],
```

`sms_from` 选项是短信消息发送者的手机号码。你可以在 Nexmo 控制面板中生成一个手机号码。

格式化短信通知

如果通知支持以短信方式发送，你应该在通知类里定义一个 `toNexmo` 方法。这个方法将会收到一个 `$notifiable` 实体并返回一个 `Illuminate\Notifications\Messages\NexmoMessage` 实例：

```
/**
 * 获取通知的 Nexmo / 短信展示方式
 *
 * @param mixed $notifiable
 * @return NexmoMessage
 */
public function toNexmo($notifiable)
{
    return (new NexmoMessage)
        ->content('Your SMS message content');
}
```

Unicode 内容

如果您的 SMS 消息包含 unicode 字符，您应该在构建 `NexmoMessage` 实例时调用 `unicode` 方法：

```
/**
 * 获取通知的 Nexmo / 短信展示方式
 *
 * @param mixed $notifiable
 * @return NexmoMessage
 */
```

```
public function toNexmo($notifiable)
{
    return (new NexmoMessage)
        ->content('Your unicode message')
        ->unicode();
}
```

自定义 From 号码

如果你想通过一个手机号来发送某些通知，而这个手机号不同于你的 `config/services.php` 配置文件中指定的话，你可以在 `NexmoMessage` 实例中使用 `from`：

```
/**
 * 获取通知的 Nexmo / 短信展示方式
 *
 * @param mixed $notifiable
 * @return NexmoMessage
 */
public function toNexmo($notifiable)
{
    return (new NexmoMessage)
        ->content('Your SMS message content')
        ->from('15554443333');
}
```

路由短信通知

当通过 `nexmo` 频道来发送通知的时候，通知系统会自动寻找通知实体的 `phone_number` 属性。如果你想自定义通知被发送的手机号码，你要在通知实体里定义一个 `routeNotificationForNexmo` 方法。

```
<?php

namespace App;

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

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Nexmo 频道的路由通知
     *
     * @return string
     */
}
```

```
public function routeNotificationForNexmo()
{
    return $this->phone;
}
```

Slack 通知

先决条件

通过 Slack 发送通知前，你必须通过 Composer 安装 Guzzle HTTP 库：

```
composer require guzzlehttp/guzzle
```

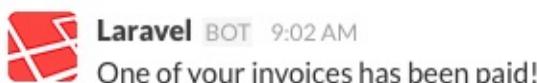
你还需要给 Slack 组配置 「Incoming Webhook」。它将提供你使用 [路由 Slack 通知](#) 时用到的 URL。

格式化 Slack 通知

如果通知支持被当做 Slack 消息发送，你应该在通知类里定义一个 `toSlack` 方法。这个方法将收到一个 `$notifiable` 实体并且返回一个 `Illuminate\Notifications\Messages\SlackMessage` 实例。Slack 消息可以包含文本内容以及一个「attachment」用来格式化额外文本或者字段数组。我们来看个基本的 `toSlack` 例子：

```
/**
 * 获取通知的 Slack 展示方式
 *
 * @param mixed $notifiable
 * @return SlackMessage
 */
public function toSlack($notifiable)
{
    return (new SlackMessage)
        ->content('One of your invoices has been paid!');
}
```

这个例子中，我们只发送了一行文本给 Slack，这会创建类似下面的一条消息：



自定义发件人和收件人

您可以使用 `from` 和 `to` 方法来自定义发件人和收件人。`from` 方法接受用户名和表情符号标识符，而 `to` 方法接受一个频道或用户名：

```
/**
 * 获取通知的 Slack 展示方式
 *
 * @param mixed $notifiable
 * @return SlackMessage
 */
public function toSlack($notifiable)
{
    return (new SlackMessage)
        ->from('Ghost', ':ghost:')
        ->to('#other')
        ->content('This will be sent to #other');
}
```

你也可以不使用表情符号，改为使用图片作为你的 logo：

```
/**
 * 获取通知的 Slack 展示方式
 *
 * @param mixed $notifiable
 * @return SlackMessage
 */
public function toSlack($notifiable)
{
    return (new SlackMessage)
        ->from('Laravel')
        ->image('https://laravel.com/favicon.png')
        ->content('This will display the Laravel logo next to the message')
;
}
```

Slack 附加项 (Attachments)

你可以给 Slack 消息添加 "附加项"。附加项提供了比简单文本消息更丰富的格式化选项。在这个例子中，我们将发送一条有关应用中异常的错误通知，它里面有个可以查看这个异常更多详情的链接：

```
/**
 * 获取通知的 Slack 展示方式。
 *
 * @param mixed $notifiable
 * @return SlackMessage
 */
public function toSlack($notifiable)
{
    $url = url('/exceptions/'.$this->exception->id);

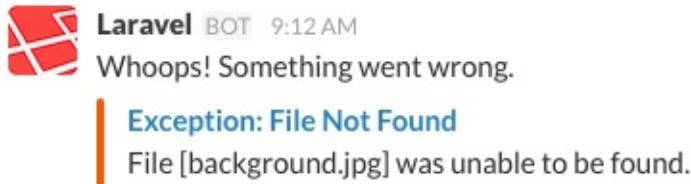
    return (new SlackMessage)
        ->error()
```

```

->content('Whoops! Something went wrong.')
->attachment(function ($attachment) use ($url) {
    $attachment->title('Exception: File Not Found', $url)
        ->content('File [background.jpg] was not found.');
});
}

```

上面这个例子将会生成一条类似下面的 Slack 消息：



附加项也允许你指定一个应该被展示给用户的数组。给定的数据将会以表格样式展示出来，这能方便阅读：

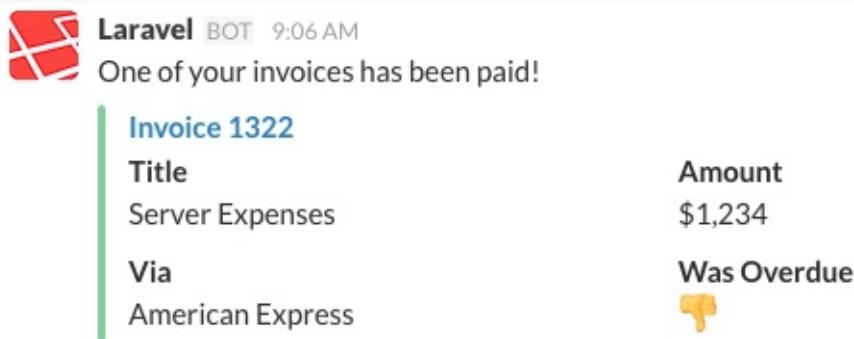
```

/**
 * 获取通知的 Slack 展示方式
 *
 * @param mixed $notifiable
 * @return SlackMessage
 */
public function toslack($notifiable)
{
    $url = url('/invoices/'.$this->invoice->id);

    return (new SlackMessage)
        ->success()
        ->content('One of your invoices has been paid!')
        ->attachment(function ($attachment) use ($url) {
            $attachment->title('Invoice 1322', $url)
                ->fields([
                    'Title' => 'Server Expenses',
                    'Amount' => '$1,234',
                    'Via' => 'American Express',
                    'Was Overdue' => ':--1:',
                ]);
        });
}

```

上面的例子将会生成一条类似下面的 Slack 消息：



Markdown 附件内容

如果一些附件字段包含 Markdown，您可以使用 `markdown` 方法指示 Slack 解析并将给定的附件字段显示为 Markdown 格式的文本：

```
/**
 * 获取通知的 Slack 展示方式
 *
 * @param mixed $notifiable
 * @return SlackMessage
 */
public function toSlack($notifiable)
{
    $url = url('/exceptions/'.$this->exception->id);

    return (new SlackMessage)
        ->error()
        ->content('Whoops! Something went wrong.')
        ->attachment(function ($attachment) use ($url) {
            $attachment->title('Exception: File Not Found', $url)
                ->content('File [background.jpg] was **not found**.')
        })
        ->markdown(['title', 'text']);
}
```

路由 Slack 通知

要把 Slack 通知路由到正确的位置，你要在通知实体中定义 `routeNotificationForSlack` 方法。它返回通知要被发往的 URL 回调地址。URL 可以通过在 Slack 组里面添加 "Incoming Webhook" 服务来生成。

```
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
```

```

use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Slack 频道的通知路由
     *
     * @return string
     */
    public function routeNotificationForSlack()
    {
        return $this->slack_webhook_url;
    }
}

```

通知事件

当通知发送后，通知系统就会触发 `Illuminate\Notifications\Events\NotificationSent` 事件。它包含了「notifiable」实体和通知实例本身。你应该在 `EventServiceProvider` 中注册监听器：

```

/**
 * 应用中的事件监听映射
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Notifications\Events\NotificationSent' => [
        'App\Listeners\LogNotification',
    ],
];

```

{tip} 注册完监听器后，使用 `event:generate` Artisan 命令来快速生成监听器类。

在事件监听器中，你可以访问事件中的 `notifiable`，`notification`，和 `channel` 属性，来了解通知接收者和通知本身：

```

/**
 * 处理事件
 *
 * @param NotificationSent $event
 * @return void
 */
public function handle(NotificationSent $event)
{
    // $event->channel
    // $event->notifiable
}

```

```
// $event->notification
}
```

自定义频道

Laravel 提供了开箱即用的通知频道，但是你可能会想编写自己的驱动来通过其他频道发送通知。 Laravel 很容易实现。首先，定义一个包含 `send` 方法的类。这个方法应该收到两个参数：`$notifiable` 和 `$notification`：

```
<?php

namespace App\Channels;

use Illuminate\Notifications\Notification;

class VoiceChannel
{
    /**
     * 发送给定通知
     *
     * @param mixed $notifiable
     * @param \Illuminate\Notifications\Notification $notification
     * @return void
     */
    public function send($notifiable, Notification $notification)
    {
        $message = $notification->toVoice($notifiable);

        // 将通知发送给 $notifiable 实例
    }
}
```

一旦定义了通知频道类，你应该在所有通知里通过 `via` 方法来简单地返回这个频道的类名。

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use App\Channels\VoiceChannel;
use App\Channels\Messages\VoiceMessage;
use Illuminate\Notifications\Notification;
use Illuminate\Contracts\Queue\ShouldQueue;

class InvoicePaid extends Notification
{
    use Queueable;
```

```
/**
 * 获取通知频道
 *
 * @param mixed $notifiable
 * @return array|string
 */
public function via($notifiable)
{
    return [VoiceChannel::class];
}

/**
 * 获取通知的声音展示方式
 *
 * @param mixed $notifiable
 * @return VoiceMessage
 */
public function toVoice($notifiable)
{
    // ...
}
```

Laravel 的扩展插件开发指南

- 简介
 - Facade 注解
- 服务提供者
- 路由
- 资源文件
 - 配置
 - 数据库迁移
 - 路由
 - 语言包
 - 视图
- 命令
- 公用 Assets
- 发布群组文件

简介

扩展包是添加功能到 Laravel 的主要方式。扩展包可以包含许多好用的功能，像 Carbon 可用于处理时间，或像 Behat 这种完整的 BDD 测试框架。

当然，这有非常多不同类型的扩展包。有些扩展包是独立运作的，意思是指他们并不依赖于任何框架，包括 Laravel。刚刚所提到的 Carbon 及 Behat 就是这种扩展包。要使用这种扩展包只需要在 `composer.json` 文件里引入它们即可。

另一方面，有些扩展包特别指定只能在 Laravel 里面集成。这些扩展包可能包含路由、控制器、视图以及扩展包的相关设置，目的是增强 Laravel 本身的功能。这份指南里将主要以开发 Laravel 专属的扩展包为目标进行说明。

Facade 注解

当开发 Laravel 应用程序时，通常来讲你使用契约(contracts) 还是 facades 并没有什么区别，因为他们都提供了基本相同的水平的可测试性。但是，在进行扩展包开发的时候，最好的方式是使用 contracts，而不是 facades。因为你开发的包并不能访问所有 Laravel 提供的测试辅助函数，模拟 contracts 要比模拟 facade 简单很多。

服务提供者

服务提供者 是你的扩展包与 Laravel 连接的重点。服务提供者负责绑定一些东西至 Laravel 的 服务容器 并告知 Laravel 要从哪加载扩展包的资源，例如视图、配置文件、语言包。

服务提供者继承了 `Illuminate\Support\ServiceProvider` 类并包含了两个方法：`register` 及 `boot`。基底的 `ServiceProvider` 类被放置在 Composer 的 `illuminate/support` 扩展包，你必须将它加入至你自己的扩展包依赖中。

若要了解更多关于服务提供者的结构与用途，请查阅 [它的文档](#)。

路由

要为你的扩展包定义路由，只需在包的服务提供者的 `boot` 方法中传递 routes 文件路径到 `loadRoutesFrom` 方法即可。在你的路由文件中，你可以使用 `Illuminate\Support\Facades\Route` 来 [注册路由](#)，就像在典型的 Laravel 应用程序中一样：

```
/**
 * 在注册后进行服务的启动。
 *
 * @return void
 */
public function boot()
{
    $this->loadRoutesFrom(__DIR__.'/path/to/routes.php');
}
```

资源文件

配置

有时候，你可能想要将扩展包的配置文件发布到应用程序本身的 config 目录上。这能够让扩展包的用户轻松的重写这些默认的设置选项。如果要发布扩展包的配置文件，只需要在服务提供者里的 `boot` 方法内使用 `publishes` 方法：

```
/**
 * 在注册后进行服务的启动。
 *
 * @return void
 */
public function boot()
{
    $this->publishes([
        __DIR__.'/path/to/config/courier.php' => config_path('courier.php'),
    ]);
}
```

现在当扩展包的用户使用 Laravel 的 `vendor:publish` 命令时，扩展包的文件将会被复制到指定的位置上。当然，只要你的配置文件被发布，就可以如其它配置文件一样被访问：

```
$value = config('courier.option');
```

{note} 您不应在配置文件中定义 闭包函数。当用户执行 `config:cache` Artisan命令时，它们不能正确地序列化。

默认的扩展包配置文件

你也可以选择合并你的扩展包配置文件和应用程序里的副本配置文件。这样能够让你的用户在已经发布的副本配置文件中只包含他们想要重写的设置选项。如果想要合并配置文件，可在服务提供者里的 `register` 方法里使用 `mergeConfigFrom` 方法：

```
/**
 * 在容器中注册绑定。
 *
 * @return void
 */
public function register()
{
    $this->mergeConfigFrom(
        __DIR__ . '/path/to/config/courier.php', 'courier'
    );
}
```

{note} 此方法仅合并配置数组的第一级。如果您的用户部分定义了多维配置数组，则不会合并缺失的选项。

路由

如果您的扩展包包中包含路由，您可以使用 `loadRoutesFrom` 方法加载它们。此方法将自动确定应用程序的路由是否已缓存，如果路由已缓存，将不会加载路由文件：

```
/**
 * 执行服务的注册后启动。
 *
 * @return void
 */
public function boot()
{
    $this->loadRoutesFrom(__DIR__ . '/routes.php');
}
```

数据库迁移

如果你的扩展包包含 [数据库迁移](#)，你需要使用 `loadMigrationsFrom` 方法告知 Laravel 如何去加载他们。`loadMigrationsFrom` 方法只需要你的扩展包的迁移文件路径作为唯一参数。

```
/**
 * 在注册后进行服务的启动。
 *
 * @return void
 */
public function boot()
{
    $this->loadMigrationsFrom(__DIR__ . '/path/to/migrations');
}
```

完成扩展包迁移文件注册之后，在运行 `php artisan migrate` 命令时，它们就会自动被执行。你并不需要把他们导出到应用程序的 `database/migrations` 目录。

语言包

如果你的扩展包里面包含了 [本地化](#)，则可以使用 `loadTranslationsFrom` 方法来告知 Laravel 该如何加载它们。举个例子，如果你的扩展包名称为 `courier`，你可以按照以下方式将其添加至服务提供者的 `boot` 方法：

```
/**
 * 在注册后进行服务的启动。
 *
 * @return void
 */
public function boot()
{
    $this->loadTranslationsFrom(__DIR__ . '/path/to/translations', 'courier');
}
```

扩展包翻译参照使用了双分号 `package::file.line` 语法。所以，你可以按照以下方式来加载 `courier` 扩展包中的 `messages` 文件 `welcome` 语句：

```
echo trans('courier::messages.welcome');
```

发布语言包

如果你想将扩展包的语言包发布至应用程序的 `resources/lang/vendor` 目录，则可以使用服务提供者的 `publishes` 方法。`publishes` 方法接受一个包含扩展包路径及对应发布位置的数组。例如，在我们的 `courier` 扩展包中发布语言包：

```
/**
 * 在注册后进行服务的启动。
 *
 * @return void
 */
public function boot()
```

```
{
    $this->loadTranslationsFrom(__DIR__.'/path/to/translations', 'courier');

    $this->publishes([
        __DIR__.'/path/to/translations' => resource_path('lang/vendor/courier'),
    ]);
}
```

现在，当使用你扩展包的用户运行 Laravel 的 `vendor:publish` Artisan 命令时，扩展包的语言包将会被复制到指定的位置上。

视图

若要在 Laravel 中注册扩展包 [视图](#)，则必须告诉 Laravel 你的视图位置。你可以使用服务提供者的 `loadViewsFrom` 方法来实现。`loadViewsFrom` 方法允许两个参数：视图模板路径与扩展包名称。例如，如果你的扩展包名称是 `courier`，你可以按照以下方式将其添加至服务提供者的 `boot` 方法内：

```
/**
 * 在注册后进行服务的启动。
 *
 * @return void
 */
public function boot()
{
    $this->loadViewsFrom(__DIR__.'/path/to/views', 'courier');
}
```

扩展包视图参照使用了双分号 `package::view` 语法。所以，你可以通过如下方式从 `courier` 扩展包中加载 `admin` 视图：

```
Route::get('admin', function () {
    return view('courier::admin');
});
```

重写扩展包视图

当你使用 `loadViewsFrom` 方法时，Laravel 实际上为你的视图注册了两个位置：一个是应用程序的 `resources/views/vendor` 目录，另一个是你所指定的目录。所以，以 `courier` 为例：当用户请求一个扩展包的视图时，Laravel 会在第一时间检查 `resources/views/vendor/courier` 是否有开发者提供的自定义版本视图存在。接着，如果这个路径没有自定义的视图，Laravel 会搜索你在扩展包 `loadViewsFrom` 方法里所指定的视图路径。这个方法可以让用户很方便的自定义或重写扩展包视图。

发布视图

若要发布扩展包的视图至 `resources/views/vendor` 目录，则必须使用服务提供者的 `publishes` 方法。`publishes` 方法允许一个包含扩展包视图路径及对应发布路径的数组。

```
/**
 * 在注册后进行服务的启动。
 *
 * @return void
 */
public function boot()
{
    $this->loadViewsFrom(__DIR__.'/path/to/views', 'courier');

    $this->publishes([
        __DIR__.'/path/to/views' => resource_path('views/vendor/courier'),
    ]);
}
```

现在，当你的扩展包用户运行 Laravel 的 `vendor:publish` Artisan 命令时，扩展包的视图将会被复制到指定的位置上。

命令

给你的扩展包注册 Artisan 命令，您可以使用 `commands` 方法。此方法需要一个命令类的数组。一旦命令被注册，您可以使用 [Artisan 命令行](#) 执行它们：

```
/**
 * 在注册后进行服务的启动。
 *
 * @return void
 */
public function boot()
{
    if ($this->app->runningInConsole()) {
        $this->commands([
            FooCommand::class,
            BarCommand::class,
        ]);
    }
}
```

公用 Assets

你的扩展包内可能会包含许多的资源文件，像 JavaScript、CSS 和图片等文件。如果要发布这些资源文件到应用程序的 `public` 目录上，只需使用服务提供者的 `publishes` 方法。在这个例子中，我们也会增加一个 `public` 的资源分类标签，可用于发布与分类关联的资源文件：

```
/**
 * 在注册后进行服务的启动。
 *
 * @return void
 */
public function boot()
{
    $this->publishes([
        __DIR__.'/path/to/assets' => public_path('vendor/courier'),
    ], 'public');
}
```

现在，当您的扩展包的用户执行 `vendor:publish` 命令时，您的 Assets 将被复制到指定的发布位置。由于每次更新包时通常都需要覆盖资源，因此您可以使用 `--force` 标志：

```
php artisan vendor:publish --tag=public --force
```

发布群组文件

你可能想要分别发布分类的扩展包资源文件或是资源。举例来说，你可能想让用户不用发布扩展包的所有资源文件，只需要单独发布扩展包的配置文件即可。这可以通过在调用 `publishes` 方法时使用「标签」来实现。例如，让我们在扩展包的服务提供者中的 `boot` 方法定义两个发布群组：

```
/**
 * 在注册后进行服务的启动。
 *
 * @return void
 */
public function boot()
{
    $this->publishes([
        __DIR__.'/../config/package.php' => config_path('package.php')
    ], 'config');

    $this->publishes([
        __DIR__.'/../database/migrations/' => database_path('migrations')
    ], 'migrations');
}
```

现在当你的用户使用 `vendor:publish` Artisan 命令时，就可以通过标签名称分别发布不同分类的资源文件：

```
php artisan vendor:publish --tag=config
```


Laravel 的队列系统介绍

- 简介
 - 连接 Vs. 队列
 - 驱动的必要设置
- 创建任务类
 - 生成任务类
 - 任务类结构
- 分发任务
 - 延迟分发
 - 自定义队列 & 连接
 - 指定任务最大尝试次数 / 超时值
 - 错误处理
- 运行队列处理器
 - 队列优先级
 - 队列处理器 & 部署
 - 任务过期 & 超时
- Supervisor 配置
- 处理失败的任务
 - 清除失败任务
 - 任务失败事件
 - 重试失败的任务
- 任务事件

简介

Laravel 队列为不同的后台队列服务提供统一的 API，例如 Beanstalk, Amazon SQS, Redis，甚至其他基于关系型数据库的队列。队列的目的是将耗时的任务延时处理，比如发送邮件，从而大幅度缩短Web请求和相应的时间。

队列配置文件存放在 `config/queue.php`。每一种队列驱动的配置都可以在该文件中找到，包括数据库，`Beanstalkd`, `Amazon SQS`, `Redis`，以及同步（本地使用）驱动。其中还包含了一个 `null` 队列驱动用于那些放弃队列的任务。

连接 Vs. 队列

在开始使用 Laravel 队列前，弄明白「连接」和「队列」的区别是很重要的。在你的 `config/queue.php` 配置文件里，有一个 `connections` 配置选项。这个选项给 Amazon SQS, Beanstalk，或者 Redis 这样的后端服务定义了一个特有的连接。不管是哪一种，一个给定的连接可能会有多个「队列」，而「队列」可以被认为是不同的栈或者大量的队列任务。

要注意的是，`queue` 配置文件中每个连接的配置示例中都包含一个 `queue` 属性。这是默认队列，任务被发给指定连接的时候会被分发到这个队列中。换句话说，如果你分发任务的时候没有显式定义队列，那么它就会被放到连接配置中 `queue` 属性所定义的队列中：

```
// 这个任务将被分发到默认队列...
dispatch(new Job);

// 这个任务将被发送到「emails」队列...
dispatch((new Job)->onQueue('emails'));
```

有些应用可能不需要把任务发到不同的队列，而只发到一个简单的队列中就行了。但是把任务推到不同的队列仍然是非常有用的，因为 Laravel 队列处理器允许你定义队列的优先级，所以你能给不同的队列划分不同的优先级或者区分不同任务的不同处理方式了。比如说，如果你把任务推到 `high` 队列中，你就能让队列处理器优先处理这些任务了：

```
php artisan queue:work --queue=high,default
```

驱动的必要设置

数据库

要使用 `database` 这个队列驱动的话，你需要创建一个数据表来存储任务，你可以用 `queue:table` 这个 Artisan 命令来创建这个数据表的迁移。当迁移创建好以后，就可以用 `migrate` 这条命令来创建数据表：

```
php artisan queue:table

php artisan migrate
```

Redis

为了使用 `redis` 队列驱动，你需要在你的配置文件 `config/database.php` 中配置 Redis 的数据库连接

如果你的 Redis 队列连接使用的是 Redis 集群，你的队列名称必须包含 [key hash tag](#)。这是为了确保所有的 redis 键对于一个给定的队列都置于同一哈希中：

```
'redis' => [
    'driver' => 'redis',
    'connection' => 'default',
    'queue' => '{default}',
    'retry_after' => 90,
],
```

其它队列驱动的依赖扩展包

在使用列表里的队列服务前，必须安装以下依赖扩展包：

- Amazon SQS: `aws/aws-sdk-php ~3.0` - Beanstalkd: `pda/pheanstalk ~3.0` - Redis: `predis/predis ~1.0`

创建任务

生成任务类

在你的应用程序中，队列的任务类都默认放在 `app/Jobs` 目录下，如果这个目录不存在，那当你运行 `make:job artisan` 命令时目录就会被自动创建。你可以用以下的 Artisan 命令来生成一个新的队列任务：

```
php artisan make:job SendReminderEmail
```

生成的类实现了 `Illuminate\Contracts\Queue\ShouldQueue` 接口，这意味着这个任务将会被推送到队列中，而不是同步执行。

任务类结构

任务类的结构很简单，一般来说只会包含一个让队列用来调用此任务的 `handle` 方法。我们来看一个示例的任务类，这个示例里，假设我们管理着一个播客发布服务，在发布之前需要处理上传播客文件：

```
<?php

namespace App\Jobs;

use App\Podcast;
use App\AudioProcessor;
use Illuminate\Bus\Queueable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class ProcessPodcast implements ShouldQueue
{
    use InteractsWithQueue, Queueable, SerializesModels;

    protected $podcast;

    /**
     * 创建一个新的任务实例。
     *
     * @param Podcast $podcast
     * @return void
     */
}
```

```

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

/**
 * 运行任务。
 *
 * @param AudioProcessor $processor
 * @return void
 */
public function handle(AudioProcessor $processor)
{
    // Process uploaded podcast...
}
}

```

注意，在这个例子中，我们在任务类的构造器中直接传递了一个 [Eloquent 模型](#)。因为我们在任务类里引用了 `SerializesModels` 这个，使得 Eloquent 模型在处理任务时可以被优雅地序列化和反序列化。如果你的队列任务类在构造器中接收了一个 Eloquent 模型，那么只有可识别出该模型的属性会被序列化到队列里。当任务被实际运行时，队列系统便会自动从数据库中重新取回完整的模型。这整个过程对你的应用程序来说是完全透明的，这样可以避免在序列化完整的 Eloquent 模式实例时所带来的一个问题。

在队列处理任务时，会调用 `handle` 方法，而这里我们也可以通过 `handle` 方法的参数类型提示，让 Laravel 的 [服务容器](#) 自动注入依赖对象。

{note} 像图片内容这种二进制数据，在放入队列任务之前必须使用 `base64_encode` 方法转换一下。否则，当这项任务放置到队列中时，可能无法正确序列化为 JSON。

分发任务

你写好任务类后，就能通过 `dispatch` 辅助函数来分发它了。唯一需要传递给 `dispatch` 的参数是这个任务类的实例：

```

<?php

namespace App\Http\Controllers;

use App\Jobs\ProcessPodcast;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PodcastController extends Controller
{
    /**
     * 保存播客。
     *

```

```

    * @param Request $request
    * @return Response
    */
public function store(Request $request)
{
    // 创建播客...

    dispatch(new ProcessPodcast($podcast));
}
}

```

{tip} `dispatch` 提供了一种简捷、全局可用的函数，它也非常容易测试。查看下 [Laravel 测试文档](#) 来了解更多。

延迟分发

如果你想延迟执行一个队列中的任务，你可以用任务实例的 `delay` 方法。这个方法是 `Illuminate\Bus\Queueable` trait 提供的，而这个 trait 在所有自动生成的任务类中都是默认加载了的。对于延迟任务我们可以举个例子，比如指定一个被分发10分钟后才执行的任务：

```

<?php

namespace App\Http\Controllers;

use Carbon\Carbon;
use App\Jobs\ProcessPodcast;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PodcastController extends Controller
{
    /**
     * 保存一个新的播客。
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        // 创建播客...

        $job = (new ProcessPodcast($podcast))
            ->delay(Carbon::now()->addMinutes(10));

        dispatch($job);
    }
}

```

{note} Amazon SQS 队列服务最大延迟 15 分钟。

自定义队列 & 连接

分发任务到指定队列

通过推送任务到不同的队列，你可以给队列任务分类，甚至可以控制给不同的队列分配多少任务。记住，这个并不是要推送任务到队列配置文件中不同的「connections」里，而是推送到一个连接中不同的队列里。要指定队列的话，就调用任务实例的 `onQueue` 方法：

```
<?php

namespace App\Http\Controllers;

use App\Jobs\ProcessPodcast;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PodcastController extends Controller
{
    /**
     * 保存一个新的播客。
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        // 创建播客...

        $job = (new ProcessPodcast($podcast))->onQueue('processing');

        dispatch($job);
    }
}
```

分发任务到指定连接

如果你使用了多个队列连接，你可以把任务推到指定连接。要指定连接的话，你可以调用任务实例的 `onConnection` 方法：

```
<?php

namespace App\Http\Controllers;

use App\Jobs\ProcessPodcast;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;
```

```

class PodcastController extends Controller
{
    /**
     * 保存一个新的播客。
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        // 创建播客...

        $job = (new ProcessPodcast($podcast))->onConnection('sq');
        dispatch($job);
    }
}

```

当然，你可以链式调用 `onConnection` 和 `onQueue` 来同时指定任务的连接和队列：

```

$job = (new ProcessPodcast($podcast))
    ->onConnection('sq')
    ->onQueue('processing');

```

指定任务最大尝试次数 / 超时值

最大尝试次数

在一项任务中指定最大的尝试次数可以尝试通过 Artisan 命令行 `--tries` 来设置：

```
php artisan queue:work --tries=3
```

但是，你可以采取更为精致的方法来完成这项工作比如说在任务类中定义最大尝试次数。如果在类和命令行中都定义了最大尝试次数， Laravel 会优先执行任务类中的值：

```

<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
{
    /**
     * 任务最大尝试次数
     *
     * @var int
     */
}

```

```
    public $tries = 5;
}
```

超时

同样的，任务可以运行的最大秒数可以使用 Artisan 命令行上的 `--timeout` 开关指定：

```
php artisan queue:work --timeout=30
```

然而，你也可以在任务类中定义一个变量来设置可运行的最大描述，如果在类和命令行中都定义了最大尝试次数， Laravel 会优先执行任务类中的值：

```
<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
{
    /**
     * 任务运行的超时时间。
     *
     * @var int
     */
    public $timeout = 120;
}
```

错误处理

如果任务运行的时候抛出异常，这个任务就自动被释放回队列，这样它就能被再重新运行了。如果继续抛出异常，这个任务会继续被释放回队列，直到重试次数达到你应用允许的最多次数。这个最多次数是在调用 `queue:work` Artisan 命令的时候通过 `--tries` 参数来定义的。更多队列处理器的信息可以 [在下面看到](#)。

运行队列处理器

Laravel 包含一个队列处理器，当新任务被推到队列中时它能处理这些任务。你可以通过 `queue:work` Artisan 命令来运行处理器。要注意，一旦 `queue:work` 命令开始，它将一直运行，直到你手动停止或者你关闭控制台：

```
php artisan queue:work
```

{tip} 要让 `queue:work` 进程永久在后台运行，你应该使用进程监控工具，比如 `Supervisor` 来保证队列处理器没有停止运行。

一定要记得，队列处理器是长时间运行的进程，并在内存里保存着已经启动的应用状态。这样的结果就是，处理器运行后如果你修改代码那这些改变是不会应用到处理器中的。所以在你重新部署过程中，一定要[重启队列处理器](#)。

指定连接 & 队列

你可以指定队列处理器所使用的连接。你在 `config/queue.php` 配置文件里定义了多个连接，而你传递给 `work` 命令的连接名字要至少跟它们其中一个是一致的：

```
php artisan queue:work redis
```

你可以自定义队列处理器，方式是处理给定连接的特定队列。举例来说，如果你所有的邮件都是在 `redis` 连接中的 `emails` 队列中处理的，你就能通过以下命令启动一个只处理那个特定队列的队列处理器了：

```
php artisan queue:work redis --queue=emails
```

资源注意事项

守护程序队列不会在处理每个作业之前「重新启动」框架。因此，在每个任务完成后，您应该释放任何占用过大的资源。例如，如果你使用GD库进行图像处理，你应该在完成后用 `imagedestroy` 释放内存。

队列优先级

有时候你希望设置处理队列的优先级。比如在 `config/queue.php` 里你可能设置了 `redis` 连接中的默认队列优先级为 `low`，但是你可能偶尔希望把一个任务推到 `high` 优先级的队列中，像这样：

```
dispatch((new Job)->onQueue('high'));
```

要验证 `high` 队列中的任务都是在 `low` 队列中的任务之前处理的，你要启动一个队列处理器，传递给它队列名字的列表并以英文逗号，间隔：

```
php artisan queue:work --queue=high,low
```

队列处理器 & 部署

因为队列处理器都是 long-lived 进程，如果代码改变而队列处理器没有重启，他们是不能应用新代码的。所以最简单的方式就是重新部署过程中要重启队列处理器。你可以很优雅地只输入 `queue:restart` 来重启所有队列处理器。

```
php artisan queue:restart
```

这个命令将会告诉所有队列处理器在执行完当前任务后结束进程，这样才不会有任务丢失。因为队列处理器在执行 `queue:restart` 命令时对结束进程，你应该运行一个进程管理器，比如 [Supervisor](#) 来自动重新启动队列处理器。

任务过期 & 超时

任务过期

`config/queue.php` 配置文件里，每一个队列连接都定义了一个 `retry_after` 选项。这个选项指定了任务最多处理多少秒后就被当做失败重试了。比如说，如果这个选项设置为 `90`，那么当这个任务持续执行了 `90` 秒而没有被删除，那么它将被释放回队列。通常情况下，你应该把 `retry_after` 设置为最长耗时的任务所对应的时间。

{note} 唯一没有 `retry_after` 选项的连接是 Amazon SQS。当用 Amazon SQS 时，你必须通过 [Amazon](#) 命令行来配置这个重试阈值。

队列处理器超时

`queue:work` Artisan 命令对外有一个 `--timeout` 选项。这个选项指定了 Laravel 队列处理器最多执行多长时间后就应该被关闭掉。有时候一个队列的子进程会因为很多原因僵死，比如一个外部的 HTTP 请求没有响应。这个 `--timeout` 选项会移除超出指定事件限制的僵死进程。

```
php artisan queue:work --timeout=60
```

`retry_after` 配置选项和 `--timeout` 命令行选项是不一样的，但是可以同时工作来保证任务不会丢失并且不会重复执行。

{note} `--timeout` 应该永远都要比 `retry_after` 短至少几秒钟的时间。这样就能保证任务进程总能在失败重试前就被杀死了。如果你的 `--timeout` 选项大于 `retry_after` 配置选项，你的任务可能被执行两次。

队列进程睡眠时间

当队列需要处理任务时，进程将继续处理任务，它们之间没有延迟。但是，如果没有新的工作可用，`sleep` 参数决定了工作进程将「睡眠」多长时间：

```
php artisan queue:work --sleep=3
```

Supervisor 配置

安装 Supervisor

Supervisor 是一个 Linux 操作系统上的进程监控软件，它会在 `queue:listen` 或 `queue:work` 命令发生失败后自动重启它们。要在 Ubuntu 安装 Supervisor，可以用以下命令：

```
sudo apt-get install supervisor
```

{tip} 如果自己手动配置 Supervisor 听起来有点难以应付，可以考虑使用 [Laravel Forge](#)，它能给你的 Laravel 项目自动安装与配置 Supervisor。

配置 Supervisor

Supervisor 的配置文件一般是放在 `/etc/supervisor/conf.d` 目录下，在这个目录中你可以创建任意数量的配置文件来要求 Supervisor 怎样监控你的进程。例如我们创建一个 `laravel-worker.conf` 来启动与监控一个 `queue:work` 进程：

```
[program:laravel-worker]
process_name=%(program_name)s_%(process_num)02d
command=php /home/forge/app.com/artisan queue:work sqs --sleep=3 --tries=3
autostart=true
autorestart=true
user=forge
numprocs=8
redirect_stderr=true
stdout_logfile=/home/forge/app.com/worker.log
```

这个例子里的 `numprocs` 命令会要求 Supervisor 运行并监控 8 个 `queue:work` 进程，并且在它们运行失败后重新启动。当然，你必须更改 `command` 命令的 `queue:work sqs`，以显示你所选择的队列驱动。

启动 Supervisor

当这个配置文件被创建后，你需要更新 Supervisor 的配置，并用以下命令来启动该进程：

```
sudo supervisorctl reread
sudo supervisorctl update
sudo supervisorctl start laravel-worker:*
```

更多有关 Supervisor 的设置与使用，请参考 [Supervisor](#) 官方文档。

处理失败的任务

有时候你队列中的任务会失败。不要担心，本来事情就不会一帆风顺。Laravel 内置了一个方便的方式来指定任务重试的最大次数。当任务超出这个重试次数后，它就会被插入到 `failed_jobs` 数据表里面。要创建 `failed_jobs` 表的话，你可以用 `queue:failed-table` 命令：

```
php artisan queue:failed-table
```

```
php artisan migrate
```

然后运行队列处理器，在调用 `queue:work` 命令时你应该通过 `--tries` 参数指定任务的最大重试次数。如果不指定，任务就会永久重试：

```
php artisan queue:work redis --tries=3
```

清除失败任务

你可以在任务类里直接定义 `failed` 方法，它能在任务失败时运行任务的清除逻辑。这个地方用来发一条警告给用户或者重置任务执行的操作等再好不过了。导致任务失败的异常信息会被传递到 `failed` 方法：

```
<?php

namespace App\Jobs;

use Exception;
use App\Podcast;
use App\AudioProcessor;
use Illuminate\Bus\Queueable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class ProcessPodcast implements ShouldQueue
{
    use InteractsWithQueue, Queueable, SerializesModels;

    protected $podcast;

    /**
     * 创建一个新的任务实例。
     *
     * @param Podcast $podcast
     * @return void
     */
    public function __construct(Podcast $podcast)
    {
        $this->podcast = $podcast;
    }

    /**
     * 执行任务。
     *
     * @param AudioProcessor $processor
     * @return void
     */
}
```

```

public function handle(AudioProcessor $processor)
{
    // 处理上传播客...
}

/**
 * 要处理的失败任务。
 *
 * @param Exception $exception
 * @return void
 */
public function failed(Exception $exception)
{
    // 给用户发送失败通知，等等...
}
}

```

任务失败事件

如果你想注册一个当队列任务失败时会被调用的事件，则可以用 `Queue::failing` 方法。这样你就有机会通过这个事件来用 e-mail 或 HipChat 通知你的团队。例如我们可以在 Laravel 内置的 `AppServiceProvider` 中对这个事件附加一个回调函数：

```

<?php

namespace App\Providers;

use Illuminate\Support\Facades\Queue;
use Illuminate\Queue\Events\JobFailed;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * 启动任意应用程序的服务。
     *
     * @return void
     */
    public function boot()
    {
        Queue::failing(function (JobFailed $event) {
            // $event->connectionName
            // $event->job
            // $event->exception
        });
    }

    /**
     * 注册服务提供者。
     *

```

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

重试失败任务

要查看你在 `failed_jobs` 数据表中的所有失败任务，则可以用 `queue:failed` 这个 Artisan 命令：

```
php artisan queue:failed
```

`queue:failed` 命令会列出所有任务的 ID、连接、队列以及失败时间，任务 ID 可以被用在重试失败的任务上。例如要重试一个 ID 为 `5` 的失败任务，其命令如下：

```
php artisan queue:retry 5
```

要重试所有失败的任务，可以使用 `queue:retry` 并使用 `all` 作为 ID：

```
php artisan queue:retry all
```

如果你想删除掉一个失败任务，可以用 `queue:forget` 命令：

```
php artisan queue:forget 5
```

`queue:flush` 命令可以让你删除所有失败的任务：

```
php artisan queue:flush
```

任务事件

使用队列的 `before` 和 `after` 方法，你能指定任务处理前和处理后的回调处理。在这些回调里正是实现额外的日志记录或者增加统计数据的好时机。通常情况下，你应该在 [服务容器](#) 中调用这些方法。例如，我们使用 Laravel 中的 AppServiceProvider：

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Queue;
use Illuminate\Support\ServiceProvider;
use Illuminate\Queue\Events\JobProcessed;
```

```

use Illuminate\Queue\Events\JobProcessing;

class AppServiceProvider extends ServiceProvider
{
    /**
     * 启动任意服务。
     *
     * @return void
     */
    public function boot()
    {
        Queue::before(function (JobProcessing $event) {
            // $event->connectionName
            // $event->job
            // $event->job->payload()
        });

        Queue::after(function (JobProcessed $event) {
            // $event->connectionName
            // $event->job
            // $event->job->payload()
        });
    }

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

```

在 队列 facade 中使用 looping 方法，你可以尝试在队列获取任务之前执行指定的回调方法。举个例子，你可以用闭包来回滚之前已失败任务的事务。

```

Queue::looping(function () {
    while (DB::transactionLevel() > 0) {
        DB::rollBack();
    }
});

```

Laravel 的任务调度（计划任务）功能 Task Scheduling

- [简介](#)
- [定义调度](#)
 - [调度频率设置](#)
 - [避免任务重复](#)
 - [维护模式](#)
- [任务输出](#)
- [任务钩子](#)

简介

在过去，开发者必须为每个需要调度的任务生成单独的 Cron 项目。然而令人头疼的是任务调度不受版本控制，并且需要 SSH 到服务器上来增加 Cron 条目。

Laravel 命令调度器允许你在 Laravel 中对命令调度进行清晰流畅的定义，并且仅需要在服务器上增加一条 Cron 项目即可。你的调度已经定义在 `app\Console\Kernel.php` 文件的 `schedule` 方法中。为了方便你开始，在该方法内包含了一个简单的例子。你可以随意增加调度到 `Schedule` 对象中。

启动调度器

使用调度器时，你只需要把 Cron 添加到你的服务器，如果你不知道如何添加到服务器，你可以使用 [Laravel Forge](#) 服务来管理你的 Cron。

```
* * * * * php /path/to/artisan schedule:run >> /dev/null 2>&1
```

该 Cron 将于每分钟调用一次 Laravel 命令调度器，当 `schedule:run` 命令执行时，Laravel 会评估你的计划任务并运行预定任务。

定义调度

你可以将所有的计划任务定义在 `App\Console\Kernel` 类的 `schedule` 方法中。在开始之前，先让我们来看看一个任务的调度示例。在该例子中，我们计划了一个会在每天午夜被调用的 闭包。该 闭包 将运行数据库查询语句来清除某个数据表：

```
<?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::class,
    ];

    /**
     * 定义应用的命令调度
     *
     * @param \Illuminate\Console\Scheduling\Schedule $schedule
     * @return void
     */
    protected function schedule(Schedule $schedule)
    {
        $schedule->call(function () {
            DB::table('recent_users')->delete();
        })->daily();
    }
}

```

除了计划 闭包 调用，你还能计划 Artisan 命令以及系统命令操作。举个例子，你可以使用 command 方法传参命令名称或者命令类名称来计划一个 Artisan 命令：

```

$schedule->command('emails:send --force')->daily();

$schedule->command(EmailsCommand::class, ['--force'])->daily();

```

exec 命令可发送命令到操作系统上：

```
$schedule->exec('node /home/forge/script.js')->daily();
```

调度频率设置

当然，你可以针对你的任务来分配多种调度计划：

方法	描述
->cron('* * * * *');	自定义调度任务
->everyMinute();	每分钟执行一次任务
->everyFiveMinutes();	每五分钟执行一次任务
->everyTenMinutes();	每十分钟执行一次任务

<code>->everyThirtyMinutes();</code>	每半小时执行一次任务
<code>->hourly();</code>	每小时执行一次任务
<code>->hourlyAt(17);</code>	每一个小时的第 17 分钟运行一次
<code>->daily();</code>	每到午夜执行一次任务
<code>->dailyAt('13:00');</code>	每天的 13:00 执行一次任务
<code>->twiceDaily(1, 13);</code>	每天的 1:00 和 13:00 分别执行一次任务
<code>->weekly();</code>	每周执行一次任务
<code>->monthly();</code>	每月执行一次任务
<code>->monthlyOn(4, '15:00');</code>	在每个月的第四天的 15:00 执行一次任务
<code>->quarterly();</code>	每季度执行一次任务
<code>->yearly();</code>	每年执行一次任务
<code>->timezone('America/New_York');</code>	设置时区

这些方法可以合并其它限制条件以生成更精确的调度。例如在某周的某几天运行调度。举个例子，计划一个每周周一的调度：

```
// 每周一的下午一点钟运行
$schedule->call(function () {
    //
})->weekly()->mondays()->at('13:00');

// 工作日中从早上 8 点钟运行到下午 5 点
$schedule->command('foo')
    ->weekdays()
    ->hourly()
    ->timezone('America/Chicago')
    ->between('8:00', '17:00');
```

下方列出其它额外限制条件：

方法	描述
<code>->weekdays();</code>	限制任务在工作日
<code>->sundays();</code>	限制任务在星期日
<code>->mondays();</code>	限制任务在星期一
<code>->tuesdays();</code>	限制任务在星期二
<code>->wednesdays();</code>	限制任务在星期三
<code>->thursdays();</code>	限制任务在星期四
<code>->fridays();</code>	限制任务在星期五
<code>->saturdays();</code>	限制任务在星期六

<code>->between(\$start, \$end);</code>	限制任务运行在开始到结束时间范围内
<code>->when(Closure);</code>	限制任务基于一个为真的验证

时间范围限制

`between` 方法可以用来限制一天中某个时间范围内：

```
$schedule->command('reminders:send')
    ->hourly()
    ->between('7:00', '22:00');
```

类似的，`unlessBetween` 方法可以用来排除时间段：

```
$schedule->command('reminders:send')
    ->hourly()
    ->unlessBetween('23:00', '4:00');
```

为真验证限制条件

`when` 方法可以用来判断是否要运行任务，主要基于一个指定的为真验证的运行结果。如果指定的闭包 返回 `true`，且没有其它限制条件存在，那么这个任务将会被继续运行。

```
$schedule->command('emails:send')->daily()->when(function () {
    return true;
});
```

`skip` 是 `when` 的颠倒意味。`skip` 方法返回 `true` 的话，计划就不会运行。

```
$schedule->command('emails:send')->daily()->skip(function () {
    return true;
});
```

当链式调用了 `when` 方法时，计划命令只有在所有的 `when` 条件返回 `true` 时才运行。

避免任务重复

默认情况，即便之前相同的任务主体仍未结束，现有计划任务依旧会被运行。为了避免这个问题，你可以使用 `withoutOverlapping` 方法：

```
$schedule->command('emails:send')->withoutOverlapping();
```

在这个例子中，如果没有其它 `emails:send` Artisan 命令 在运行的话，此任务将于每分钟被运行一次。当你有些任务运行时间过长，且无法预测出具体所需时间时，`withoutOverlapping` 方法将会特别有帮助。

维护模式

当应用进入 [维护模式](#) 时，默认情况下 Laravel 的调度功能将会停止运行。这是因为我们考虑到你可能在维护模式下做一些破坏性的维护工作，我们不想让任务调度对这些工作造成干扰。然而，如果你想强制某个任务在维护模式下运行的话，你可以使用 `evenInMaintenanceMode` 方法：

```
$schedule->command('emails:send')->evenInMaintenanceMode();
```

任务输出

Laravel 调度器为任务调度输出提供多种便捷方法。首先，通过 `sendOutputTo` 你可以发送输出到单个文件上以便后续检查：

```
$schedule->command('emails:send')
    ->daily()
    ->sendOutputTo($filePath);
```

如果想将输出附加到指定的文件上，则可以使用 `appendOutputTo` 方法：

```
$schedule->command('emails:send')
    ->daily()
    ->appendOutputTo($filePath);
```

通过 `emailOutputTo` 方法，你可以发送输出到你所指定的电子邮件上。注意，你必须先通过 `sendOutputTo` 方法将其输出到一个文件。同时，在邮件发出之前，你需要先设置 Laravel 的 [电子邮件服务](#)：

```
$schedule->command('foo')
    ->daily()
    ->sendOutputTo($filePath)
    ->emailOutputTo('foo@example.com');
```

{note} `emailOutputTo`，`sendOutputTo` 和 `appendOutputTo` 方法只适用于 `command` 方法，不支持 `call` 方法。

任务钩子

通过 `before` 与 `after` 方法，你能让特定的代码在任务完成之前及之后运行：

```
$schedule->command('emails:send')
    ->daily()
    ->before(function () {
        // Task is about to start...
    })
```

```
->after(function () {  
    // Task is complete...  
});
```

Ping 网址

通过 `pingBefore` 与 `thenPing` 方法，调度器能自动的在一个任务完成之前或之后 ping 一个指定的网址。该方法在你计划的任务进行或完成时，可用来有效的通知一个外部服务，例如 [Laravel Envoyer](#):

```
$schedule->command('emails:send')  
    ->daily()  
    ->pingBefore($url)  
    ->thenPing($url);
```

使用 `pingBefore($url)` 或 `thenPing($url)` 功能需要 Guzzle HTTP 函数库的支持。可在 `composer.json` 文件中加入以下代码来安装 Guzzle:

```
composer require guzzlehttp/guzzle
```

9 数据库

数据库：入门

- [简介](#)
 - [配置信息](#)
 - [数据库读写分离](#)
 - [使用多数据库连接](#)
- [运行原生 SQL 语句](#)
 - [监听查询事件](#)
- [数据库事务](#)

简介

Laravel 对主流数据库系统连接和查询都提供了很好的支持，尤其是流畅的 [查询语句构造器](#)， Laravel 支持四种类型的数据库：

- MySQL
- Postgres
- SQLite
- SQL Server

配置信息

Laravel 应用程序的数据库配置文件放置在 `config/database.php` 文件中。在这个配置文件内你可以定义所有的数据库连接，以及指定默认使用哪个连接。在此文件内提供了所有支持的数据库系统示例。

默认情况下， Laravel 的 [环境配置](#) 示例会使用 [Laravel Homestead](#)，对于 Laravel 开发来说这是一个相当便利的本地虚拟机。当然你也可以根据需求来随时修改本机端的数据库设置。

SQLite 配置

请使用 `touch database/database.sqlite` 命令创建一个 SQLite 文件，您可以通过使用数据库的绝对路径来轻松地配置您的环境变量来指向这个新创建的数据库：

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

SQL Server 配置

Laravel 支持 SQL Server 数据库，你需要在 `config/database.php` 中为连接 SQL Server 数据库做配置：

```
'sqlsrv' => [
    'driver' => 'sqlsrv',
```

```
'host' => env('DB_HOST', 'localhost'),
'database' => env('DB_DATABASE', 'forge'),
'username' => env('DB_USERNAME', 'forge'),
'password' => env('DB_PASSWORD', ''),
'charset' => 'utf8',
'prefix' => '',
],
```

数据库读写分离

有时候你希望把一个数据库作为只读数据库，而另一个数据库则负责写入、更新以及删除。Laravel 会让你轻而易举的实现，并适用于原始查找、查询语句构造器或是 Eloquent ORM。

通过下面这个例子，我们来学习如何配置数据库读写连接的分离：

```
'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，有两个键加入了这个配置文件数组内：`read` 和 `write`。它们两个都是一个数组且只包含了一个键：`host`。而 `read` 和 `write` 连接的其他配置都包含在 `mysql` 数组中。

所以，如果需要在主要的数组内重写值，只需在 `read` 和 `write` 数组内放置设置参数即可。在这个例子中，`192.168.1.1` 将会只提供数据读取数据的功能，而 `196.168.1.2` 提供数据库写入。数据库的凭证、前缀、编码设置，以及所有其它的选项都被存放在 `mysql` 数组内，这两个连接将会共用这些选项。

使用多数据库连接

当使用多个数据连接时，你可以使用 `DB facade` 的 `connection` 方法。在 `config/database.php` 中定义好的数据库连接 `name` 作为 `connection` 的参数进行传递。

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

你也可以在连接的实例中使用 `getPdo` 方法访问原始的底层 PDO 实例：

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

运行原生 SQL 语句

配置好数据库连接以后，你可以使用 `DB facade` 来执行查询。`DB facade` 提供了 `select`、`update`、`insert`、`delete` 和 `statement` 的查询方法。

运行 Select

运行一个基础的查询语句，你可以使用 `DB facade` 的 `select` 方法：

```
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller
{
    /**
     * 在应用用户中查询出已激活的用户列表。
     *
     * @return Response
     */
    public function index()
    {
        $users = DB::select('select * from users where active = ?', [1]);

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

传递到 `select` 方法的第一个参数是一个原生的 SQL 查询，而第二个参数则是传递的所有绑定到查询中的参数值。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，你可以像下面这样访问结果值：

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

使用命名绑定

除了使用 `?` 来表示参数绑定外，你也可以使用命名绑定运行查找：

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

运行 insert

运行 `Insert` 语句，你可以是使用 `DB facade` 的 `insert` 方法。像 `select` 一样，该方法将原生SQL语句作为第一个参数，将参数绑定作为第二个参数：

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

运行 Update

`update` 方法用于更新已经存在于数据库的记录。该方法会返回此语句执行所影响的行数：

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

运行 Delete

`delete` 方法用于删除已经存在于数据库的记录。如同 `update` 一样，删除的行数将会被返回。

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

运行一般声明

有些数据库没有返回值，对于这种类型的操作，可以使用 `DB facade` 的 `statement` 方法。

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

监听查询事件

如果你希望能够监控到程序执行的每一条 SQL 语句，那么你可以使用 `listen` 方法。该方法对查询日志和调试非常有用，你可以在 [服务容器](#) 中注册该方法：

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\DB;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * 启动应用服务。
     *
     * @return void
    
```

```

    */
public function boot()
{
    DB::listen(function ($query) {
        // $query->sql
        // $query->bindings
        // $query->time
    });
}

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

```

数据库事务

想要在一个数据库事务中运行一连串操作，可以使用 `DB facade` 的 `transaction` 方法。如果在事务的 `Closure` 中抛出了异常，那么事务会自动的执行回滚操作。如果 `Closure` 成功的执行，那么事务就会自动的进行提交操作。你不需要在使用 `transaction` 方法时考虑手动执行回滚或者提交操作：

```

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

    DB::table('posts')->delete();
});

```

处理死锁

`transaction` 方法参数列表的第二位接收一个可选的参数，这个参数定义了在发生死锁时，事务会重试的次数。如果重试结束还没有成功执行，将会抛出一个异常：

```

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

    DB::table('posts')->delete();
}, 5);

```

手动操作事务

如果你想要手动开始一个事务的回滚和提交操作，你可以使用 `DB facade` 的 `beginTransaction` 方法。

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

你也可以通过 `rollBack` 方法来回滚事务：

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

最后，可以通过 `commit` 方法来提交这个事务：

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

{tip} 使用 `DB facade` 的事务方法也适用于 [查询语句构造器](#) 和 [Eloquent ORM](#)。

Laravel 数据库之：数据库请求构建器

- [简介](#)
- [获取结果](#)
 - [分块结果](#)
 - [聚合](#)
- [Selects](#)
- [原始表达式](#)
- [Joins](#)
- [Unions](#)
- [Where 子句](#)
 - [参数分组](#)
 - [Where Exists 语法](#)
 - [JSON 查询语句](#)
- [Ordering, Grouping, Limit, & Offset](#)
- [条件语句](#)
- [Inserts](#)
- [Updates](#)
 - [更新 JSON](#)
 - [自增或自减](#)
- [Deletes](#)
- [悲观锁](#)

简介

Laravel 的数据库查询构造器提供了一个方便、流畅的接口，用来创建及运行数据库查询语句。它能用来执行应用程序中的大部分数据库操作，且能在所有被支持的数据库系统中使用。

Laravel 的查询构造器使用 PDO 参数绑定，来保护你的应用程序免受 SQL 注入的攻击。在绑定传入字符串前不需要清理它们。

获取结果

从数据表中获取所有的数据列

你可以使用 `DB facade` 的 `table` 方法开始查询。这个 `table` 方法针对查询表返回一个查询构造器实例，允许你在查询时链式调用更多约束，并使用 `get` 方法获取最终结果：

```
<?php  
  
namespace App\Http\Controllers;  
  
use Illuminate\Support\Facades\DB;
```

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

class UserController extends Controller
{
    /**
     * Show a list of all of the application's users.
     *
     * @return Response
     */
    public function index()
    {
        $users = DB::table('users')->get();

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

`get` 方法会返回一个 `Illuminate\Support\Collection` 结果，其中每个结果都是一个 PHP `StdClass` 对象的实例。您可以通过访问列中对象的属性访问每个列的值：

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

从数据表中获取单个列或行

如果你只需要从数据表中获取一行数据，则可以使用 `first` 方法。这个方法将返回单个 `StdClass` 对象：

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

echo $user->name;
```

如果你不需要一整行数据，则可以使用 `value` 方法来从单条记录中取出单个值。此方法将直接返回字段的值：

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

获取一列的值

如果你想要获取一个包含单个字段值的集合，可以使用 `pluck` 方法。在下面的例子中，我们将取出 `roles` 表中 `title` 字段的集合：

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

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

```
}
```

你也可以在返回的数组中指定自定义的键值字段：

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

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

结果分块

如果你需要操作数千条数据库记录，可以考虑使用 `chunk` 方法。这个方法每次只取出一小块结果，并会将每个块传递给一个 `闭包` 处理。这个方法对于编写数千条记录的 [Artisan 命令](#) 是非常有用的。例如，让我们把 `users` 表进行分块，每次操作 100 条数据：

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

你可以从 `闭包` 中返回 `false`，以停止对后续分块的处理：

```
DB::table('users')->orderBy('id')->chunk(100, function ($users) {
    // Process the records...

    return false;
});
```

聚合

查询构造器也支持各种聚合方法，如 `count`、`max`、`min`、`avg` 和 `sum`。你可以在创建查询后调用其中的任意一个方法：

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

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

当然，你也可以将这些方法结合其它子句来进行查询：

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

Selects

指定一个 Select 子句

当然，你并不会总是想从数据表中选出所有的字段。这时可使用 `select` 方法自定义一个 `select` 子句来查询指定的字段：

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

`distinct` 方法允许你强制让查询返回不重复的结果：

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

如果你已有一个查询构造器实例，并且希望在现有的 `select` 子句中加入一个字段，则可以使用 `addSelect` 方法：

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

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

原始表达式

有时候你可能需要在查询中使用原始表达式。这些表达式将会被当作字符串注入到查询中，所以要小心避免造成 SQL 注入攻击！要创建一个原始表达式，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();
```

Joins

Inner Join 语法

查询构造器也可以编写 `join` 语法。若要执行基本的「inner join」，你可以在查询构造器实例上使用 `join` 方法。传递给 `join` 方法的第一个参数是你要 `join` 数据表的名称，而其它参数则指定用来连接的字段约束。当然，如你所见，你可以在单个查找中连接多个数据表：

```
$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();
```

Left Join 语法

如果你想用「left join」来代替「inner join」，请使用 `leftJoin` 方法。`leftJoin` 方法与 `join` 方法有着相同的用法：

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

Cross Join 语法

使用 `crossJoin` 方法和你想要交叉连接的表名来做「交叉连接」。交叉连接通过第一个表和连接表生成一个笛卡尔积：

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

高级 Join 语法

你还可以指定更高级的 `join` 子句。让我们传递一个 闭包 作为 `join` 方法的第二个参数来作为开始。此 闭包 将会收到一个 `JoinClause` 对象，让你可以在 `join` 子句中指定约束：

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

如果你想要在连接中使用「where」风格的子句，则可以在连接中使用 `where` 和 `orWhere` 方法。这些方法将会比较值和对应的字段，而不是比较两个字段：

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

Unions

查询构造器也提供了一个快捷的方法来「合并」两个查询。例如，你可以先创建一个初始查询，并使用 `union` 方法将它与第二个查询进行合并：

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

```

->whereNull('first_name');

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

```

{tip} 也可使用 `unionAll` 方法，它和 `union` 方法有着相同的用法。

Where 子句

简单的 Where 子句

你可以在查询构造器实例中使用 `where` 方法从而把 `where` 子句加入到这个查询中。基本的 `where` 方法需要3个参数。第一个参数是字段的名称。第二个参数是运算符，它可以是数据库所支持的任何运算符。最后，第三个参数是要对字段进行评估的值。

例如，这是一个要验证「votes」字段的值等于 100 的查询：

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

为方便起见，如果你只是想简单的校验某个字段等于一个指定的值，你可以直接将这个值作为第二个参数传入 `where` 方法：

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

当然，在编写 `where` 子句时，你也可以使用各种数据库所支持其它的运算符：

```

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

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

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

```

你也可以通过一个条件数组做 `where` 的查询：

```

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

```

Or 语法

你可以在查询中加入 `or` 子句和 `where` 链式一起来约束查询。`orWhere` 方法接收和 `where` 方法相同的参数：

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

其它 Where 子句

whereBetween

`whereBetween` 方法用来验证字段的值介于两个值之间：

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

whereNotBetween

`whereNotBetween` 方法验证字段的值 不 在两个值之间：

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

whereIn 与 whereNotIn

`whereIn` 方法验证字段的值包含在指定的数组内：

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

`whereNotIn` 方法验证字段的值 不 包含在指定的数组内：

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

whereNull 与 whereNotNull

`whereNull` 方法验证字段的值为 `NULL`：

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

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

`whereNotNull` 方法验证字段的值不为 `NULL` :

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

`whereDate / whereMonth / whereDay / whereYear`

`whereDate` 方法比较某字段的值与指定的日期是否相等:

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

`whereMonth` 方法比较某字段的值是否与一年的某一个月份相等:

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

`whereDay` 方法比较某列的值是否与一月中的某一天相等:

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

`whereYear` 方法比较某列的值是否与指定的年份相等:

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

`whereColumn`

`whereColumn` 方法用来检测两个列的数据是否一致:

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

此方法还可以使用运算符:

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

```
->whereColumn('updated_at', '>', 'created_at')
->get();
```

`whereColumn` 方法可以接收数组参数。条件语句会使用 `and` 连接起来：

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

参数分组

有时你可能需要创建更高级的 `where` 子句，例如「`where exists`」或者嵌套的参数分组。Laravel 的查询构造器也能够处理这些。让我们先来看一个在括号中将约束分组的示例：

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

如你所见，上面例子会传递一个 `闭包` 到 `orWhere` 方法，告诉查询构造器开始一个约束分组。此 `闭包` 接收一个查询构造器实例，你可用它来设置应包含在括号分组内的约束。这个例子会生成以下 SQL：

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

Where Exists 语法

`whereExists` 方法允许你编写 `where exists` SQL 子句。此方法会接收一个 `闭包` 参数，此闭包接收一个查询语句构造器实例，让你可以定义应放在「`exists`」SQL 子句中的查找：

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

上述查询将生成以下 SQL：

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

JSON 查询语句

Laravel 也支持查询 JSON 类型的字段。目前，本特性仅支持 MySQL 5.7+ 和 Postgres 数据库。可以使用 `->` 运算符来查询 JSON 列数据：

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

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

Ordering, Grouping, Limit 及 Offset

orderBy

`orderBy` 方法允许你根据指定字段对查询结果进行排序。`orderBy` 方法的第一个参数是你想要用来排序的字段，而第二个参数则控制排序的顺序，可以为 `asc` 或 `desc`：

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

latest / oldest

`latest` 和 `oldest` 方法允许你更容易的依据日期对查询结果排序。默认查询结果将依据 `created_at` 列。或者，你可以使用字段名称排序：

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

inRandomOrder

`inRandomOrder` 方法可以将查询结果随机排序。例如，你可以使用这个方法获取一个随机用户：

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

groupBy / having / havingRaw

`groupBy` 和 `having` 方法可用来对查询结果进行分组。 `having` 方法的用法和 `where` 方法类似：

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

`havingRaw` 方法可以将一个原始的表达式设置为 `having` 子句的值。例如，我们能找出所有销售额超过 2,500 元的部门：

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

skip / take

你可以使用 `skip` 和 `take` 方法来限制查询结果数量或略过指定数量的查询：

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

或者，你也可以使用 `limit` 和 `offset` 方法：

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

条件语句

有时候，你希望某个值为 `true` 时才执行查询。例如，如果在传入请求中存在指定的输入值的时候才执行这个 `where` 语句。你可以使用 `when` 方法实现：

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

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

只有当 `when` 方法的第一个参数为 `true` 时，闭包里的 `where` 语句才会执行。如果第一个参数是 `false`，这个闭包将不会被执行。

你可能会把另一个闭包当作第三个参数传递给 `when` 方法。如果第一个参数的值为 `false` 时，这个闭包将执行。为了说明如何使用此功能，我们将使用它配置默认排序的查询：

```
$sortBy = null;

$users = DB::table('users')
    ->when($sortBy, function ($query) use ($sortBy) {
        return $query->orderBy($sortBy);
    }, function ($query) {
        return $query->orderBy('name');
    })
    ->get();
```

Inserts

查询构造器也提供了 `insert` 方法，用来插入记录到数据表中。`insert` 方法接收一个包含字段名和值的数组作为参数：

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

你甚至可以在 `insert` 调用中传入一个嵌套数组向表中插入多条记录。每个数组表示要插入表中的行：

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

自增 ID

若数据表存在自增 id，则可以使用 `insertGetId` 方法来插入记录并获取其 ID：

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

{note} 当使用 PostgreSQL 时，`insertGetId` 方法将预测自动递增字段的名称为 `id`。若你要从不同「顺序」来获取 ID，则可以将顺序名称作为第二个参数传递给 `insertGetId` 方法。

Updates

当然，除了在数据库中插入记录外，你也可以使用 `update` 来更新已存在的记录。`update` 方法和 `insert` 方法一样，接收含有字段及值的数组，其中包括要更新的字段。可以使用 `where` 子句来约束 `update` 查找：

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

Updating JSON Columns

当更新一个JSON 列时，你应该使用 `->` 语法来访问 JSON 对象的键。仅在数据库支持 JSON 列的时候才可使用这个操作：

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

自增或自减

查询构造器也为指定字段提供了便利的自增和自减方法。此方法提供了一个比手动编写 `update` 语法更具表达力且更精练的接口。

这两个方法都必须接收至少一个参数（要修改的字段）。也可选择传入第二个参数，用来控制字段应递增 / 递减的量：

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

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

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

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

您还可以指定要操作中更新其它字段：

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

Deletes

查询构造器也可使用 `delete` 方法从数据表中删除记录。在 `delete` 前，还可使用 `where` 子句来约束 `delete` 语法：

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

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

如果你需要清空表，你可以使用 `truncate` 方法，这将删除所有行，并重置自动递增 ID 为零：

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

悲观锁

查询构造器也包含一些可以帮助你在 `select` 语法上实现「悲观锁定」的函数。若要在查询中使用「共享锁」，可以使用 `sharedLock` 方法。共享锁可防止选中的数据列被篡改，直到事务被提交为止：

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

另外，你也可以使用 `lockForUpdate` 方法。使用「更新」锁可避免行被其它共享锁修改或选取：

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

Laravel 的分页功能

- [简介](#)
- [基本使用](#)
 - [对查询语句构造器进行分页](#)
 - [对 Eloquent 模型进行分页](#)
 - [手动创建分页](#)
- [显示分页结果](#)
 - [将结果转换为 JSON](#)
- [自定义分页视图](#)
- [分页器实例方法](#)

简介

在其他的框架中，分页是令人非常烦恼的。 Laravel 的分页器为 [查询构造器](#) 和 [Eloquent ORM](#) 集成提供了方便，并提供基于数据库结果集的分页开箱即用。分页器生产的 HTML 兼容 [Bootstrap CSS framework](#).

基本使用

对查询语句构造器进行分页

有几种方法可以对项目进行分页。最简单的是在 [查询语句构造器](#) 或 [Eloquent 查询](#) 中使用 `paginate` 方法。`paginate` 方法会自动基于当前用户查看的当前页面来设置适当的限制和偏移。默认情况下，当前页面通过 HTTP 请求所带的 `?page` 参数的值来检测。当然，这个值会被 Laravel 自动检测，并且自动插入到由分页器生产的链接中。

首先，让我们先来看看如何在查询上调用 `paginate` 方法。在这个例子中，传递给 `paginate` 方法的唯一参数是你希望在「每页」显示的数据数。在这种情况下，让我们指定希望每页显示 15 条数据：

```
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller
{
    /**
     * 显示应用中所有的用户
     *
     * @return Response

```

```

    */
public function index()
{
    $users = DB::table('users')->paginate(15);

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

```

{note} 目前，使用 `groupBy` 语句的分页操作无法由 Laravel 有效执行。如果你需要在一个分页结果集中使用 `groupBy`，建议你查询数据库并手动创建分页器。

「简单分页」

如果你只需要在你的分页视图中显示简单的「上一页」和「下一页」的链接，你可以使用 `simplePaginate` 方法来执行更高效的查询。当你在渲染视图时不需要显示页码链接，这对于大数据集非常有用。

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

对 Eloquent 模型进行分页

你也可以对 Eloquent 查询进行分页。在这个例子中，我们将对 `User` 模型进行分页并且每页显示 15 条数据。正如你看到的，语法几乎与基于查询语句构造器的分页相同：

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

当然，你可以在对查询设置了其他限制之后调用 `paginate` 方法，例如 `where` 子句：

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

你也可以在 Eloquent 模型进行分页使用 `simplePaginate` 方法：

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

手动创建分页

有时候你可能希望手动创建一个分页实例，并传递其到项目数组中。你可以依据你的需求创建 `Illuminate\Pagination\Paginator` 或 `Illuminate\Pagination\LengthAwarePaginator` 实例。

`Paginator` 类不需要知道结果集中的数据项总数；然而，由于这个，该类没有用于检索最后一页索引的方法。`LengthAwarePaginator` 接收的参数几乎和 `Paginator` 一样；但是，它需要计算结果集中的数据项总数。

换一种说法，`Paginator` 对应于查询语句构造器和 `Eloquent` 的 `simplePaginate` 方法，而 `LengthAwarePaginator` 对应于 `paginate` 方法。

{note} 当手动创建分页器实例时，你应该手动「切割」传递给分页器的结果集。如果你不确定如何做到这一点，查阅 PHP 的函数 [array_slice](#)。

显示分页结果

当调用 `paginate` 方法的时候，你将会接收到一个 `Illuminate\Pagination\LengthAwarePaginator` 实例。而当你调用 `simplePaginate` 方法时，你将会接收到一个 `Illuminate\Pagination\Paginator` 实例。这些对象提供了一些描述结果集的方法。除了这些辅助方法，分页器是迭代器并且可以作为数组循环。因此，一旦检索到结果集，你可以使用 `Blade` 模板显示结果集并渲染页面链接：

```
<div class="container">
    @foreach ($users as $user)
        {% raw %}{{ $user->name }} {% endraw %}
    @endforeach
</div>

{{ $users->links() }}
```

`links` 方法将会渲染结果集中的其他页链接。这些链接中每一个都已经包含了 `?page` 查询字符串变量。记住，`links` 方法生产的 HTML 兼容 [Bootstrap CSS framework](#)

自定义分页器的 URI

`withPath` 方法允许你在生成分页链接时自定义 URI。例如，如果你想分页器生成的链接如 `http://example.com/custom/url?page=N`，你应该传递 `custom/url` 到 `withPath` 方法：

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

    $users->withPath('custom/url');

    //
});
```

附加参数到分页链接中

你可以使用 `append` 方法附加查询参数到分页链接中。例如，要附加 `sort=votes` 到每个分页链接，你应该这样调用 `append` 方法：

```
{% raw %}{{ $users->appends(['sort' => 'votes'])->links() }} {% endraw %}
```

如果你希望附加「哈希片段」到分页器的链接中，你应该使用 `fragment` 方法。例如，要附加 `#foo` 到每个分页链接的末尾，应该这样调用 `fragment` 方法：

```
{% raw %}{{ $users->fragment('foo')->links() }} {% endraw %}
```

将结果转换为 JSON

Laravel 分页器结果类实现了 `Illuminate\Contracts\Support\Jsonable` 接口契约并且提供 `toJson` 方法，所以它很容易将你的分页结果集转换为 Json。你也可以简单地通过从路由或者控制器动作返回分页实例并将其转换为 JSON：

```
Route::get('users', function () {
    return App\User::paginate();
});
```

从分页器获取的 JSON 将包含元信息，如：`total`，`current_page`，`last_page` 等等。实际的结果对象将通过 JSON 数组中的 `data` 键来获取。以下是一个从路由返回分页器实例创建的 JSON 示例：

```
{
    "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
        }
    ]
}
```

自定义分页视图

在默认情况下，视图渲染显示的分页链接都兼容 Bootstrap CSS 框架。但是，如果你不使用 Bootstrap，你可以自定义你自己的视图去渲染这些链接。当在分页器实例中调用 `links` 方法，传递视图名称作为方法的第一参数：

```
{% raw %}{{ $paginator->links('view.name') }} {% endraw %}
```

```
// 传递数据到视图中...
{{ $paginator->links('view.name', ['foo' => 'bar']) }}
```

然而，自定义分页视图最简单的方法是通过 `vendor:publish` 命令将它们导出到你的 `resources/views/vendor` 目录：

```
php artisan vendor:publish --tag=laravel-pagination
```

这个命令将视图放置在 `resources/views/vendor/pagination` 目录中。这个目录下的 `default.blade.php` 文件对应于默认分页视图。你可以简单地编辑这个文件以修改分页的 HTML。

分页器实例方法

每个分页器实例通过以下方法提供额外的分页信息：

- `$results->count()`
- `$results->currentPage()`
- `$results->firstItem()`
- `$results->hasMorePages()`
- `$results->lastItem()`
- `$results->lastPage()` (当使用 `simplePagination` 时无效)
- `$results->nextPageUrl()`
- `$results->perPage()`
- `$results->previousPageUrl()`
- `$results->total()` (当使用 `simplePagination` 时无效)
- `$results->url($page)`

Laravel 的数据库迁移 Migrations

- 简介
- 生成迁移
- 迁移结构
- 运行迁移
 - 回滚迁移
- 数据表
 - 创建数据表
 - 重命名与删除数据表
- 字段
 - 创建字段
 - 字段修饰
 - 修改字段
 - 移除字段
- 索引
 - 创建索引
 - 删除索引
 - 外键约束

简介

数据库迁移就像是数据库的版本控制，可以让你的团队轻松修改并共享应用程序的数据库结构。迁移通常会搭配上 Laravel 的数据库结构构造器来让你方便地构建数据库结构。如果你曾经出现过让同事手动在数据库结构中添加字段的情况，数据库迁移可以解决你这个问题。

Laravel 的 `Schema facade` 对所有 Laravel 支持的数据库系统提供了创建和操作数据表的相应支持。

生成迁移

使用 `make:migration Artisan 命令` 来创建迁移：

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

新的迁移文件将会被放置在 `database/migrations` 目录中。每个迁移文件的名称都包含了一个时间戳，以便让 Laravel 确认迁移的顺序。

`--table` 和 `--create` 选项可用来指定数据表的名称，或是该迁移被执行时会创建的新数据表。这些选项需在预生成迁移文件时填入指定的数据表：

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

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

如果你想为生成的迁移指定一个自定义输出路径，则可以在运行 `make:migration` 命令时添加 `--path` 选项。提供的路径必须是相对于应用程序的基本路径。

迁移结构

一个迁移类会包含两个方法：`up` 和 `down`。`up` 方法可为数据库添加新的数据表、字段或索引，而 `down` 方法则是 `up` 方法的逆操作。

你可以在这两个方法中使用 Laravel 数据库结构构造器来创建以及修改数据表。若要了解 数据库结构构造器中的所有可用方法，[可查阅它的文档](#)。以下的迁移实例会创建一张 `flights` 数据表：

```
<?php

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

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

    /**
     * 回滚数据库迁移
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('flights');
    }
}
```

运行迁移

使用 `migrate` Artisan 命令，来运行所有未运行过的迁移：

```
php artisan migrate
```

{note} 如果你使用的是 Homestead 虚拟机，你需要在虚拟机中执行以上命令。

在线上环境强制执行迁移

一些迁移的操作是具有破坏性的，它们可能会导致数据丢失。为了保护线上环境的数据库，系统会在这些命令被运行之前显示确认提示。若要忽略此提示并强制运行命令，则可以使用 `--force` 标记：

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

回滚迁移

若要回滚最后一次迁移，则可以使用 `rollback` 命令。此命令是对上一次执行的「批量」迁移回滚，其中可能包括多个迁移文件：

```
php artisan migrate:rollback
```

在 `rollback` 命令后加上 `step` 参数，你可以限制回滚迁移的个数。例如，下面的命令将会回滚最后的 5 个迁移。

```
php artisan migrate:rollback --step=5
```

`migrate:reset` 命令可以回滚应用程序中的所有迁移：

```
php artisan migrate:reset
```

使用单个命令来执行回滚或迁移

`migrate:refresh` 命令不仅会回滚数据库的所有迁移还会接着运行 `migrate` 命令。所以此命令可以有效的重新创建整个数据库：

```
php artisan migrate:refresh

// 刷新数据库结构并执行数据填充
php artisan migrate:refresh --seed
```

使用 `refresh` 命令并加上 `step` 参数，你也可以限制执行回滚和再迁移的个数。比如，下面的命令会回滚并再迁移最后的 5 个迁移：

```
php artisan migrate:refresh --step=5
```

数据表

创建数据表

要创建一张新的数据表，可以使用 `Schema facade` 的 `create` 方法。 `create` 方法接收两个参数：第一个参数为数据表的名称，第二个参数为一个 `闭包`，此闭包会接收一个用于定义新数据表的 `Blueprint` 对象：

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

当然，在创建数据表的时候，你也可以使用任何数据库结构构造器的 [字段方法](#) 来定义数据表的字段。

检查数据表或字段是否存在

你可以方便地使用 `hasTable` 和 `hasColumn` 方法来检查数据表或字段是否存在：

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

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

数据库连接与存储引擎

如果你想要在一个非默认的数据库连接中进行数据库结构操作，可以使用 `connection` 方法：

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

你可以在数据库结构构造器上设置 `engine` 属性来设置数据表的存储引擎：

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

    $table->increments('id');
});
```

重命名与删除数据表

若要重命名一张已存在的数据表，可以使用 `rename` 方法：

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

要删除已存在的数据表，可使用 `drop` 或 `dropIfExists` 方法：

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

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

重命名带外键的数据表

在重命名前，你需要检查外键的约束涉及到的数据表名需要在迁移文件中显式的提供，而不是让 Laravel 按照约定来设置一个名称。因为那样会让外键约束关联到旧的数据表上。

字段

创建字段

使用 `Schema facade` 的 `table` 方法可以更新已有的数据表。如同 `create` 方法，`table` 方法会接收两个参数：一个是数据表的名称，另一个则是接收 `Blueprint` 实例的 `闭包`。我们可以使用它来为数据表新增字段：

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

可用的字段类型

数据库结构构造器包含了许多字段类型，供你构建数据表时使用：

命令	描述
<code>\$table->bigIncrements('id');</code>	递增 ID（主键），相当于「UNSIGNED BIG INTEGER」型态。
<code>\$table->bigInteger('votes');</code>	相当于 BIGINT 型态。
<code>\$table->binary('data');</code>	相当于 BLOB 型态。
<code>\$table->boolean('confirmed');</code>	相当于 BOOLEAN 型态。
<code>\$table->char('name', 4);</code>	相当于 CHAR 型态，并带有长度。
<code>\$table->date('created_at');</code>	相当于 DATE 型态
<code>\$table->dateTime('created_at');</code>	相当于 DATETIME 型态。
<code>\$table->dateTimeTz('created_at');</code>	DATETIME (带时区) 形态
<code>\$table->decimal('amount', 5, 2);</code>	相当于 DECIMAL 型态，并带有精度与基数。

<code>\$table->double('column', 15, 8);</code>	相当于 DOUBLE 型态，总共有 15 位数，在小数点后面有 8 位数。
<code>\$table->enum('choices', ['foo', 'bar']);</code>	相当于 ENUM 型态。
<code>\$table->float('amount', 8, 2);</code>	相当于 FLOAT 型态，总共有 8 位数，在小数点后面有 2 位数。
<code>\$table->increments('id');</code>	递增的 ID (主键)，使用相当于「UNSIGNED INTEGER」的型态。
<code>\$table->integer('votes');</code>	相当于 INTEGER 型态。
<code>\$table->ipAddress('visitor');</code>	相当于 IP 地址形态。
<code>\$table->json('options');</code>	相当于 JSON 型态。
<code>\$table->jsonb('options');</code>	相当于 JSONB 型态。
<code>\$table->longText('description');</code>	相当于 LONGTEXT 型态。
<code>\$table->macAddress('device');</code>	相当于 MAC 地址形态。
<code>\$table->mediumIncrements('id');</code>	递增 ID (主键)，相当于「UNSIGNED MEDIUM INTEGER」型态。
<code>\$table->mediumInteger('numbers');</code>	相当于 MEDIUMINT 型态。
<code>\$table->mediumText('description');</code>	相当于 MEDIUMTEXT 型态。
<code>\$table->morphs('taggable');</code>	加入整数 <code>taggable_id</code> 与字符串 <code>taggable_type</code> 。
<code>\$table->nullableMorphs('taggable');</code>	与 <code>morphs()</code> 字段相同，但允许为 NULL。
<code>\$table->nullableTimestamps();</code>	与 <code>timestamps()</code> 相同，但允许为 NULL。
<code>\$table->rememberToken();</code>	加入 <code>remember_token</code> 并使用 VARCHAR(100) NULL。
<code>\$table->smallIncrements('id');</code>	递增 ID (主键)，相当于「UNSIGNED SMALL INTEGER」型态。
<code>\$table->smallInteger('votes');</code>	相当于 SMALLINT 型态。
<code>\$table->softDeletes();</code>	加入 <code>deleted_at</code> 字段用于软删除操作。
<code>\$table->string('email');</code>	相当于 VARCHAR 型态。
<code>\$table->string('name', 100);</code>	相当于 VARCHAR 型态，并带有长度。
<code>\$table->text('description');</code>	相当于 TEXT 型态。
<code>\$table->time('sunrise');</code>	相当于 TIME 型态。
<code>\$table->timeTz('sunrise');</code>	相当于 TIME (带时区) 形态。
<code>\$table->tinyInteger('numbers');</code>	相当于 TINYINT 型态。
<code>\$table->timestamp('added_on');</code>	相当于 TIMESTAMP 型态。

<code>\$table->timestampTz('added_on');</code>	相当于 TIMESTAMP (带时区) 形态。
<code>\$table->timestamps();</code>	加入 <code>created_at</code> 和 <code>updated_at</code> 字段。
<code>\$table->timestampsTz();</code>	加入 <code>created_at</code> and <code>updated_at</code> (带时区) 字段，并允许为NULL。
<code>\$table->unsignedBigInteger('votes');</code>	相当于 Unsigned BIGINT 型态。
<code>\$table->unsignedInteger('votes');</code>	相当于 Unsigned INT 型态。
<code>\$table->unsignedMediumInteger('votes');</code>	相当于 Unsigned MEDIUMINT 型态。
<code>\$table->unsignedSmallInteger('votes');</code>	相当于 Unsigned SMALLINT 型态。
<code>\$table->unsignedTinyInteger('votes');</code>	相当于 Unsigned TINYINT 型态。
<code>\$table->uuid('id');</code>	相当于 UUID 型态。

字段修饰

除了上述的字段类型列表，还有一些其它的字段「修饰」，你可以将它增加到字段中。例如，若要让字段「`nullable`」，那么你可以使用 `nullable` 方法：

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

以下列表为字段的可用修饰。此列表不包括 [索引修饰](#)：

Modifier	Description
<code>->after('column')</code>	将此字段放置在其它字段「之后」（仅限 MySQL）
<code>->comment('my comment')</code>	增加注释
<code>->default(\$value)</code>	为此字段指定「默认」值
<code>->first()</code>	将此字段放置在数据表的「首位」（仅限 MySQL）
<code>->nullable()</code>	此字段允许写入 NULL 值
<code>->storedAs(\$expression)</code>	创建一个存储的生成字段（仅限 MySQL）
<code>->unsigned()</code>	设置 <code>integer</code> 字段为 <code>UNSIGNED</code>
<code>->virtualAs(\$expression)</code>	创建一个虚拟的生成字段（仅限 MySQL）

[``](#)

修改字段

先决条件

在修改字段之前，请务必在你的 `composer.json` 中增加 `doctrine/dbal` 依赖。Doctrine DBAL 函数库被用于判断当前字段的状态以及创建调整指定字段的 SQL 查询。

```
composer require doctrine/dbal
```

更新字段属性

`change` 方法让你可以修改一些已存在的字段类型，或修改字段属性。比如，你可能想增加字符串字段的长度。想了解 `change` 方法如何使用，让我们来把 `name` 字段的长度从 25 增加到 50：

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

我们也能将字段修改为 `nullable`：

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

{note} 下面的字段类型不能被「修改」：char, double, enum, mediumInteger, timestamp, tinyInteger, ipAddress, json, jsonb, macAddress, mediumIncrements, morphs, nullableMorphs, nullableTimestamps, softDeletes, timeTz, timestampTz, timestamps, timestampsTz, unsignedMediumInteger, unsignedTinyInteger, uuid。

重命名字段

要重命名字段，可使用数据库结构构造器的 `renameColumn` 方法。在重命名字段前，请确保你的 `composer.json` 文件内已经加入 `doctrine/dbal` 依赖：

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

{note} 数据表的 `enum` 字段暂时不支持修改字段属性。

移除字段

要移除字段，可使用数据库结构构造器的 `dropColumn` 方法。在删除 SQLite 数据库的字段前，你需要在 `composer.json` 文件中加入 `doctrine/dbal` 依赖并在终端执行 `composer update` 来安装函数库：

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

```
$table->dropColumn('votes');
});
```

你可以传递多个字段的数组至 `dropColumn` 方法来移除多个字段：

```
Schema::table('users', function (Blueprint $table) {
    $table->dropColumn(['votes', 'avatar', 'location']);
});
```

{note} SQLite 数据库并不支持在单个迁移中移除或修改多个字段。

索引

创建索引

数据库结构构造器支持多种类型的索引。首先，让我们先来看一个示例，其指定了字段的值必须是唯一的。你可以简单的在字段定义之后链式调用 `unique` 方法来创建索引：

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

此外，你也可以在定义完字段之后创建索引。例如：

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

你也可以传递一个字段的数组至索引方法来创建复合索引：

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

Laravel 会自动生成一个合理的索引名称，但你也可以使用第二个参数来自定义索引名称：

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

可用的索引类型

Command	Description
<code>\$table->primary('id');</code>	加入主键。
<code>\$table->primary(['first', 'last']);</code>	加入复合键。
<code>\$table->unique('email');</code>	加入唯一索引。
<code>\$table->unique('state', 'my_index_name');</code>	自定义索引名称。
<code>\$table->unique(['first', 'last']);</code>	加入复合唯一键。
<code>\$table->index('state');</code>	加入基本索引。

索引长度 & MySQL / MariaDB

Laravel 默认使用 `utf8mb4` 字符，包括支持在数据库存储「表情」。如果你正在运行的 MySQL release 版本低于 5.7.7 或 MariaDB release 版本低于 10.2.2，为了 MySQL 为它们创建索引，你可能需要手动配置迁移生成的默认字符串长度，你可以通过调用 `AppServiceProvider` 中的 `Schema::defaultStringLength` 方法来配置它：

```
use Illuminate\Support\Facades\Schema;

/**
 * 引导任何应用程序服务。
 *
 * @return void
 */
public function boot()
{
    Schema::defaultStringLength(191);
}
```

或者你可以为数据库开启 `innodb_large_prefix` 选项，有关如何正确开启此选项的说明请查阅数据库文档。

移除索引

若要移除索引，则必须指定索引的名称。Laravel 默认会自动给索引分配合理的名称。其将数据表名称、索引的字段名称及索引类型简单地连接在了一起。举例如下：

命令	描述
<code>\$table->dropPrimary('users_id_primary');</code>	从「users」数据表移除主键。
<code>\$table->dropUnique('users_email_unique');</code>	从「users」数据表移除唯一索引。
<code>\$table->dropIndex('geo_state_index');</code>	从「geo」数据表移除基本索引。

如果你对 `dropIndex` 传参索引数组，默认的约定是索引名称由数据库表名字和键名拼接而成：

```
Schema::table('geo', function (Blueprint $table) {
    $table->dropIndex(['state']); // 移除索引 'geo_state_index'
});
```

外键约束

Laravel 也为创建外键约束提供了支持，用于在数据库层中的强制引用完整性。例如，让我们定义一个有 `user_id` 字段的 `posts` 数据表，`user_id` 引用了 `users` 数据表的 `id` 字段：

```
Schema::table('posts', function (Blueprint $table) {
    $table->integer('user_id')->unsigned();
```

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

你也可以指定约束的「on delete」及「on update」：

```
$table->foreign('user_id')
    ->references('id')->on('users')
    ->onDelete('cascade');
```

要移除外键，你可以使用 `dropForeign` 方法。外键约束与索引采用相同的命名方式。所以，我们可以将数据表名称和约束字段连接起来，接着在该名称后面加上「_foreign」后缀：

```
$table->dropForeign('posts_user_id_foreign');
```

你也可以传递一个包含字段的数组，在移除的时候字段会按照惯例被自动转换为对应的外键名称：

```
$table->dropForeign(['user_id']);
```

你可以在迁移文件里使用以下方法来开启和关闭外键约束：

```
Schema::enableForeignKeyConstraints();

Schema::disableForeignKeyConstraints();
```

Laravel 数据库之：数据填充

- [简介](#)
- [编写 Seeders](#)
 - [使用模型工厂](#)
 - [调用其他 Seeders](#)
- [运行 Seeders](#)

简介

Laravel 可以用 `seed` 类轻松地为数据库填充测试数据。所有的 `seed` 类都存放在 `database/seeds` 目录下。你可以任意为 `seed` 类命名，但是应该遵守类似 `UsersTableSeeder` 的命名规范。Laravel 默认定义了一个 `DatabaseSeeder` 类。可以在这个类中使用 `call` 方法来运行其它的 `seed` 类来控制数据填充的顺序。

编写 Seeders

可以通过运行 `make:seeder Artisan 命令` 来生成一个 Seeder。所有由框架生成的 seeders 都将被放置在 `database/seeds` 目录下：

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

一个 `seeder` 类只包含一个默认方法：`run`。这个方法在 `db:seed Artisan 命令` 被调用时执行。在 `run` 方法里你可以为数据库添加任何数据。你也可以用 [查询语句构造器](#) 或 [Eloquent 模型工厂](#) 来手动添加数据。

如下所示，我们来修改默认的 `DatabaseSeeder` 类并为 `run` 方法添加一条数据库插入语句：

```
<?php

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，手动为每个模型填充指定属性很麻烦。作为替代方案，你可以使用 [模型工厂](#) 来轻松地生成大量数据库记录。首先，阅读 [模型工厂文档](#) 来学习如何定义工厂。一旦定义了工厂，就可以使用 `factory` 这个辅助函数来向数据库中添加记录。

如下所示，我们来创建 50 个用户并为每个用户创建关联：

```

/**
 * 运行数据库填充
 *
 * @return void
 */
public function run()
{
    factory(App\User::class, 50)->create()->each(function ($u) {
        $u->posts()->save(factory(App\Post::class)->make());
    });
}

```

调用其他 Seeders

在 `DatabaseSeeder` 类中，你可以使用 `call` 方法来运行其他的 `seed` 类。为避免单个 `seeder` 类过大，可使用 `call` 方法将数据填充拆分成多个文件。只需简单传递要运行的 `seeder` 类名称即可：

```

/**
 * 运行数据库填充
 *
 * @return void
 */
public function run()
{
    $this->call(UsersTableSeeder::class);
    $this->call(PostsTableSeeder::class);
    $this->call(CommentsTableSeeder::class);
}

```

运行 Seeders

一旦完成了 seeder 类的编写，就可以使用 `db:seed` 这个 Artisan 命令来填充数据库。在默认情况下，`db:seed` 命令将运行可以用来调用其他填充类的 `DatabaseSeeder` 类。但是可以用 `--class` 选项来单独运行一个特定的 seeder 类：

```
php artisan db:seed  
  
php artisan db:seed --class=UsersTableSeeder
```

也可以使用会先回滚再重新运行所有迁移的 `migrate:refresh` 命令来填充数据库。这个命令在彻底重构数据库时非常有用：

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

Laravel 的 Redis 使用指南

- 简介
 - 配置
 - Predis
 - PhpRedis
- 基本用法
 - 管道化命令
- 发布与订阅

简介

Redis 是一款开源且先进的键值对数据库。由于它的键指向的数据包含了 [字符串](#)、[哈希](#)、[列表](#)、[集合](#) 和 [有序集合](#) 这些数据类型，因此常被用作数据结构服务器。

在使用 Redis 之前，你需要通过 Composer 安装 `predis/predis` 扩展包。

```
composer require predis/predis
```

还有一种选择，你可以通过 PECL 安装 [PhpRedis](#) PHP 扩展。这个扩展安装起来更复杂，但是你可以在你的程序重度使用 redis 时获得一定的性能提升。

配置

应用程序的 Redis 配置都在 `config/database.php` 配置文件中。在这个文件里，你可以看到 `redis` 数组里面包含了应用程序使用的 Redis 服务器：

```
'redis' => [  
    'client' => 'predis',  
  
    'default' => [  
        'host' => env('REDIS_HOST', 'localhost'),  
        'password' => env('REDIS_PASSWORD', null),  
        'port' => env('REDIS_PORT', 6379),  
        'database' => 0,  
    ],  
,
```

默认的服务器配置对于开发来说应该足够了。当然，你也可以根据使用的环境来随意更改数组。只需给每个 Redis 服务器指定名称、host 和 port 即可。

译者注：关于 Redis 多连接的配置，请参阅 - [Laravel 下配置 Redis 让缓存、Session 各自使用不同的 Redis 数据库。](#)

redis 集群配置

如果你的程序使用 redis 服务器集群，你应该在 redis 配置文件中使用 `clusters` 键来定义：

```
'redis' => [
    'client' => 'predis',
    'clusters' => [
        'default' => [
            [
                'host' => env('REDIS_HOST', 'localhost'),
                'password' => env('REDIS_PASSWORD', null),
                'port' => env('REDIS_PORT', 6379),
                'database' => 0,
            ],
        ],
    ],
],
```

默认情况下，集群可以实现跨节点间客户端共享，允许你实现节点池以及创建大量可用内存。然而，注意客户端共享并没有处理失败情况；因此，主要适用于从另一个主要的数据源来建立缓存数据。如果你喜欢使用 redis 原生集群，你需要在配置文件中配置 `options` 键：

```
'redis' => [
    'client' => 'predis',
    'options' => [
        'cluster' => 'redis',
    ],
    'clusters' => [
        // ...
    ],
],
```

Predis

除了默认的 `Host`，`port`，`database` 和 `password` 服务配置项之外，Predis 还可以为每个 redis 定义其他的 [连接参数](#)。要使用这些额外的配置选项，只需将它们添加到你的 `config/database.php` 配置文件的 Redis 服务器配置项中即可：

```
'default' => [
    'host' => env('REDIS_HOST', 'localhost'),
    'password' => env('REDIS_PASSWORD', null),
    'port' => env('REDIS_PORT', 6379),
    'database' => 0,
    'read_write_timeout' => 60,
],
```

PhpRedis

{note} 如果你是通过 PECL 安装 Redis PHP 扩展，则需要重命名 config/app.php 文件里的 Redis 别名。

要使用 Phredis 扩展，你需要将 client 选项配置为 phredis。这个选项可以在 config/database.php 配置文件中找到：

```
'redis' => [
    'client' => 'phredis',
    // Rest of Redis configuration...
],
```

除了默认的 Host , port , database 和 password 服务配置项之外，Phredis 还支持下列额外连接配置： persistent , prefix , read_timeout 和 timeout 。你可以将这些选项加到 config/database.php 配置文件中 redis 服务器配置项下：

```
'default' => [
    'host' => env('REDIS_HOST', 'localhost'),
    'password' => env('REDIS_PASSWORD', null),
    'port' => env('REDIS_PORT', 6379),
    'database' => 0,
    'read_timeout' => 60,
],
```

基本用法

你可以通过调用 Redis facade 的各种方法与 Redis 进行交互。Redis facade 支持动态方法，意思就是指你可以在该 facade 调用任何 Redis 命令，该命令会直接传递给 Redis。在本例中，我们会通过 Redis facade 的 get 方法来调用 Redis 的 GET 命令：

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

```

use Illuminate\Support\Facades\Redis;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     *
     * @param int $id
     * @return Response
     */
    public function showProfile($id)
    {
        $user = Redis::get('user:profile:'.$id);

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

```

如上所述，你可以在 `Redis facade` 调用任何的 Redis 命令。Laravel 使用魔术方法来传递命令至 Redis 服务器，所以可以简单的传递 Redis 命令所需要的参数：

```

Redis::set('name', 'Taylor');

$values = Redis::lrange('names', 5, 10);

```

另外，你也可以通过 `command` 方法传递命令至服务器，它接收命令的名称作为第一个参数，第二个参数则为值的数组：

```
$values = Redis::command('lrange', ['name', 5, 10]);
```

使用多个 Redis 连接

你可以通过 `Redis::connection` 方法来得到 Redis 实例：

```
$redis = Redis::connection();
```

这会返回配置项中的默认的 redis 服务器。你也可以传递连接或者集群的名字给 `connection` 方法，来获取在 Redis 配置文件中配置的特定的服务器或者集群：

```
$redis = Redis::connection('my-connection');
```

管道化命令

当你想要在单次操作中发送多个命令至服务器时则可以使用管道化命令。`pipeline` 方法接收一个参数：带有 Redis 实例的 `闭包`。你可以发送所有的命令至此 Redis 实例，它们都会在单次操作中运行：

```
Redis::pipeline(function ($pipe) {
    for ($i = 0; $i < 1000; $i++) {
        $pipe->set("key:$i", $i);
    }
});
```

发布与订阅

Laravel 也对 Redis 的 `publish` 及 `subscribe` 提供了方便的接口。这些 Redis 命令让你可以监听指定「频道」的消息。你可以从另一个应用程序发布消息至频道，甚至使用另一种编程语言，让应用程序或进程之间容易沟通。

首先，让我们通过 `Redis` 来使用 `subscribe` 方法在一个频道设置侦听器。我们会将方法调用放置于一个 [Artisan 命令](#) 中，因为调用 `subscribe` 方法会启动一个长时间运行的进程：

```
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;
use Illuminate\Support\Facades\Redis;

class RedisSubscribe extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'redis:subscribe';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Subscribe to a Redis channel';

    /**
     * Execute the console command.
     *
     * @return mixed
     */
    public function handle()
```

```
{  
    Redis::subscribe(['test-channel'], function ($message) {  
        echo $message;  
    });  
}  
}
```

现在，我们可以通过 `publish` 方法发布消息至该频道：

```
Route::get('publish', function () {  
    // Route logic...  
  
    Redis::publish('test-channel', json_encode(['foo' => 'bar']));  
});
```

通配符订阅

你可以使用 `psubscribe` 方法订阅一个通配符频道，这在对所有频道获取所有消息时相当有用。`$channel` 名称会被传递至该方法提供的回调 `闭包` 的第二个参数：

```
Redis::psubscribe(['*'], function ($message, $channel) {  
    echo $message;  
});  
  
Redis::psubscribe(['users.*'], function ($message, $channel) {  
    echo $message;  
});
```

10 Eloquent ORM

Eloquent: 入门

- [简介](#)
- [定义模型](#)
 - [Eloquent 模型约定](#)
- [取回多个模型](#)
 - [集合](#)
 - [分块结果](#)
- [取回单个模型或集合](#)
 - [取回集合](#)
- [添加和更新模型](#)
 - [基本添加](#)
 - [基本更新](#)
 - [批量赋值](#)
 - [其他创建方法](#)
- [删除模型](#)
 - [软删除](#)
 - [查询被软删除的模型](#)
- [查询作用域](#)
 - [全局作用域](#)
 - [本地作用域](#)
- [事件](#)
 - [观察器](#)

简介

Laravel 的 Eloquent ORM 提供了漂亮、简洁的 ActiveRecord 实现来和数据库进行交互。每个数据库表都有一个对应的「模型」可用来跟数据表进行交互。你可以通过模型查询数据表内的数据，以及将记录添加到数据表中。

在开始之前，请确认你已在 `config/database.php` 文件中设置好了数据库连接。更多数据库的设置信息请查看 [数据库设置](#) 文档。

定义模型

开始之前，让我们先来创建一个 Eloquent 模型。模型通常放在 `app` 目录中，不过你可以将他们随意放在任何可通过 `composer.json` 自动加载的地方。所有的 Eloquent 模型都继承自 `Illuminate\Database\Eloquent\Model` 类。

创建模型实例的最简单方法是使用 `make:model` Artisan 命令：

```
php artisan make:model User
```

当你生成一个模型时想要顺便生成一个 [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` 模型类的例子，我们将会用它从 `flights` 数据表中取回与保存信息：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    //
}
```

数据表名称

请注意，我们并没有告诉 Eloquent `Flight` 模型该使用哪一个数据表。除非数据表明确地指定了其它名称，否则将使用类的「蛇形名称」、复数形式名称来作为数据表的名称。因此在此例子中，Eloquent 将会假设 `Flight` 模型被存储记录在 `flights` 数据表中。你可以在模型上定义一个 `table` 属性，用来指定自定义的数据表名称：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * 与模型关联的数据表
     *
     * @var string
     */
    protected $table = 'my_flights';
}
```

主键

Eloquent 也会假设每个数据表都有一个叫做 `id` 的主键字段。你也可以定义一个 `$primaryKey` 属性来重写这个约定。

此外，Eloquent 假定主键是一个递增的整数值，这意味着在默认情况下主键将自动的被强制转换为 `int`。如果你想使用非递增或者非数字的主键，你必须在你的模型 public `$incrementing` 属性设置为 `false`。

时间戳

默认情况下，Eloquent 会认为在你的数据库表有 `created_at` 和 `updated_at` 字段。如果你不希望让 Eloquent 来自动维护这两个字段，可在模型内将 `$timestamps` 属性设置为 `false`：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * 该模型是否被自动维护时间戳
     *
     * @var bool
     */
    public $timestamps = false;
}
```

如果你需要自定义自己的时间戳格式，可在模型内设置 `$dateFormat` 属性。这个属性决定了日期应如何在数据库中存储，以及当模型被序列化成数组或 JSON 格式：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * 模型的日期字段保存格式。
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

数据库连接

默认情况下，所有的 Eloquent 模型会使用应用程序中默认的数据库连接设置。如果你想为模型指定不同的连接，可以使用 `$connection` 属性：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * 此模型的连接名称。
     *
     * @var string
     */
    protected $connection = 'connection-name';
}
```

取回多个模型

一旦你创建并 [关联了一个模型到数据表](#) 上，那么你就可以从数据库中获取数据。可把每个 Eloquent 模型想像成强大的 [查询构造器](#)，它让你可以流畅地查询与模型关联的数据表。例如：

```
<?php

use App\Flight;

$flights = App\Flight::all();

foreach ($flights as $flight) {
    echo $flight->name;
}
```

增加额外的限制

Eloquent 的 `all` 方法会返回在模型数据表中的所有结果。由于每个 Eloquent 模型都可以当作一个 [查询构造器](#)，所以你可以在查询中增加规则，然后使用 `get` 方法来获取结果：

```
$flights = App\Flight::where('active', 1)
    ->orderBy('name', 'desc')
    ->take(10)
    ->get();
```

{tip} 由于 Eloquent 模型是查询构造器，因此你应当去阅读所有 [查询构造器](#) 中可用的方法。你可在 Eloquent 查询中使用这其中的任何方法。

集合

类似 `all` 以及 `get` 之类的可以取回多个结果的 Eloquent 方法，将会返回一个 `Illuminate\Database\Eloquent\Collection` 实例。`Collection` 类提供 [多种辅助函数](#) 来处理你的 Eloquent 结果。

```
$flights = $flights->reject(function ($flight) {
    return $flight->cancelled;
});
```

当然，你也可以简单地像数组一样来遍历集合：

```
foreach ($flights as $flight) {
    echo $flight->name;
}
```

分块结果

如果你需要处理数以千计的 Eloquent 查找结果，则可以使用 `chunk` 命令。`chunk` 方法将会获取一个 Eloquent 模型的「分块」，并将它们送到指定的 闭包 (closure) 中进行处理。当你在处理大量结果时，使用 `chunk` 方法可节省内存：

```
Flight::chunk(200, function ($flights) {
    foreach ($flights as $flight) {
        //
    }
});
```

传递到方法的第一个参数表示每次「分块」时你希望接收的数据数量。闭包则作为第二个参数传递，它将会在每次从数据取出分块时被调用。

使用游标

`cursor` 允许你使用游标来遍历数据库数据，一次只执行单个查询。在处理大数据量请求时 `cursor` 方法可以大幅度减少内存的使用：

```
foreach (Flight::where('foo', 'bar')->cursor() as $flight) {
    //
}
```

取回单个模型 / 集合

当然，除了从指定的数据表取回所有记录，你也可以通过 `find` 和 `first` 方法来取回单条记录。但这些方法返回的是单个模型的实例，而不是返回模型的集合：

```
// 通过主键取回一个模型...
$flight = App\Flight::find(1);

// 取回符合查询限制的第一个模型 ...
$flight = App\Flight::where('active', 1)->first();
```

你也可以用主键的集合为参数调用 `find` 方法，它将返回符合条件的集合：

```
$flights = App\Flight::find([1, 2, 3]);
```

「未找到」异常

有时候你可能希望在找不到模型时抛出一个异常，这在路由或是控制器内特别有用。`findOrFail` 以及 `firstOrFail` 方法会取回查询的第一个结果。如果没有找到相应结果，则会抛出一个 `Illuminate\Database\Eloquent\ModelNotFoundException`：

```
$model = App\Flight::findOrFail(1);

$model = App\Flight::where('legs', '>', 100)->firstOrFail();
```

如果该异常没有被捕获，则会自动返回 HTTP `404` 响应给用户，因此当使用这些方法时，你没有必要明确的编写检查来返回 `404` 响应：

```
Route::get('/api/flights/{id}', function ($id) {
    return App\Flight::findOrFail($id);
});
```

取回集合

当然，你也可以使用 `count`、`sum`、`max`，和其它 [查询构造器](#) 提供的 [聚合函数](#)。这些方法会返回适当的标量值，而不是一个完整的模型实例：

```
$count = App\Flight::where('active', 1)->count();

$max = App\Flight::where('active', 1)->max('price');
```

添加和更新模型

基本添加

要在数据库中创建一条新记录，只需创建一个新模型实例，并在模型上设置属性和调用 `save` 方法即可：

```

<?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)
    {
        // 验证请求...

        $flight = new Flight;

        $flight->name = $request->name;

        $flight->save();
    }
}

```

在这个例子中，我们把来自 HTTP 请求中的 `name` 参数简单地指定给 `App\Flight` 模型实例的 `name` 属性。当我们调用 `save` 方法，就会添加一条记录到数据库中。当 `save` 方法被调用时，`created_at` 以及 `updated_at` 时间戳将会被自动设置，因此我们不需要去手动设置它们。

基本更新

`save` 方法也可以用于更新数据库中已经存在的模型。要更新模型，则须先取回模型，再设置任何你希望更新的属性，接着调用 `save` 方法。同样的，`updated_at` 时间戳将会被自动更新，所以我们不需要手动设置它的值：

```

$flight = App\Flight::find(1);

$flight->name = 'New Flight Name';

$flight->save();

```

批量更新

也可以针对符合指定查询的任意数量模型进行更新。在这个例子中，所有 `active` 并且 `destination` 为 `San Diego` 的航班，都将会被标识为延迟：

```
App\Flight::where('active', 1)
    ->where('destination', 'San Diego')
    ->update(['delayed' => 1]);
```

`update` 方法会期望收到一个含有字段与值对应的数组，而这些字段的内容将会被更新。

{note} 当通过“Eloquent”批量更新时，`saved` 和 `updated` 模型事件将不会被更新后的模型代替。这是因为批量更新时，模型从来没有被取回。

批量赋值

你也可以使用 `create` 方法通过一行代码来保存一个新模型。被插入数据库的模型实例将会返回给你。不过，在这样做之前，你需要先在你的模型上定义一个 `fillable` 或 `guarded` 属性，因为所有的 Eloquent 模型都针对批量赋值（Mass-Assignment）做了保护。

当用户通过 HTTP 请求传入了非预期的参数，并借助这些参数更改了数据库中你并不打算要更改的字段，这时就会出现批量赋值（Mass-Assignment）漏洞。例如，恶意用户可能会通过 HTTP 请求发送 `is_admin` 参数，然后对应到你模型的 `create` 方法，此操作能让该用户把自己升级为一个管理者。

所以，在开始之前，你应该定义好哪些模型属性是可以被批量赋值的。你可以在模型上使用 `$fillable` 属性来实现。例如，让我们让 `Flight` 模型的 `name` 属性可以被批量赋值：

```
<?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'];
}
```

一旦我们已经设置好可以被批量赋值的属性，便能通过 `create` 方法来添加一条新记录到数据库。`create` 方法将返回已经被保存的模型实例：

```
$flight = App\Flight::create(['name' => 'Flight 10']);
```

如果你已经有一个 `model` 实例，你可以使用一个数组传递给 `fill` 方法：

```
$flight->fill(['name' => 'Flight 22']);
```

Guarding Attributes

`$fillable` 作为一个可以被批量赋值的属性「白名单」。另外你也可以选择使用 `$guarded`。`$guarded` 属性应该包含一个你不想要被批量赋值的属性数组。所有不在数组里面的其它属性都可以被批量赋值。因此，`$guarded` 的功能更类似一个「黑名单」。使用的时候应该只选择 `$fillable` 或 `$guarded` 中的其中一个。下面这个例子中，除了 `price` 所有的属性都可以被批量赋值：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * 不可被批量赋值的属性。
     *
     * @var array
     */
    protected $guarded = ['price'];
}
```

如果你想让所有的属性都可以被批量赋值，你应该定义 `$guarded` 为空数组。

```
/**
 * 不可被批量赋值的属性。
 *
 * @var array
 */
protected $guarded = [];
```

其它创建的方法

还有两种其它方法，你可以用来通过属性批量赋值创建你的模型：`firstOrCreate` 和 `firstOrNew`。`firstOrCreate` 方法将会使用指定的字段 / 值对，来尝试寻找数据库中的记录。如果在数据库中找不到模型，则会使用指定的属性来添加一条记录。

`firstOrNew` 方法类似 `firstOrCreate` 方法，它会尝试使用指定的属性在数据库中寻找符合的纪录。如果模型未被找到，将会返回一个新的模型实例。请注意 `firstOrnew` 返回的模型还尚未保存到数据库。你需要通过手动调用 `save` 方法来保存它：

```
// 用属性取回航班，当结果不存在时创建它...
$flight = App\Flight::firstOrCreate(['name' => 'Flight 10']);

// 用属性取回航班，当结果不存在时实例化一个新实例...
$flight = App\Flight::firstOrNew(['name' => 'Flight 10']);
```

其次，你可能会碰到模型已经存在则更新，否则创建新模型的情形，Laravel 提供了一个 `updateOrCreate` 方法来一步完成该操作，类似 `firstOrCreate` 方法，`updateOrCreate` 方法会持久化模型，所以无需调用 `save()`：

```
// If there's a flight from Oakland to San Diego, set the price to $99.
// If no matching model exists, create one.
$flight = App\Flight::updateOrCreate(
    ['departure' => 'Oakland', 'destination' => 'San Diego'],
    ['price' => 99]
);
```

删除模型

要删除模型，必须在模型实例上调用 `delete` 方法：

```
$flight = App\Flight::find(1);

$flight->delete();
```

通过键来删除现有的模型

在上面的例子中，我们在调用 `delete` 方法之前会先从数据库中取回模型。不过，如果你已知道了模型中的主键，则可以不用取回模型就能直接删除它。若要直接删除，请调用 `destroy` 方法：

```
App\Flight::destroy(1);

App\Flight::destroy([1, 2, 3]);

App\Flight::destroy(1, 2, 3);
```

通过查询来删除模型

当然，你也可以运行在一组模型删除查询。在这个例子中，我们会删除被标记为不活跃的所有航班。像批量更新那样，批量删除不会为任何已删除的模型触发模型事件：

```
$deletedRows = App\Flight::where('active', 0)->delete();
```

{note} 当使用 Eloquent 批量删除语句时，`deleting` 和 `deleted` 模型事件不会在被删除模型实例上触发。因为删除语句执行时，不会检索回模型实例。

软删除

除了从数据库中移除实际记录，Eloquent 也可以「软删除」模型。当模型被软删除时，它们并不会真的从数据库中被移除。而是会在模型上设置一个 `deleted_at` 属性并将其添加到数据库。如果模型有一个非空值 `deleted_at`，代表模型已经被软删除了。要在模型上启动软删除，则必须在模型上使用 `Illuminate\Database\Eloquent\SoftDeletes` trait 并添加 `deleted_at` 字段到你的 `$dates` 属性上：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Flight extends Model
{
    use SoftDeletes;

    /**
     * 需要被转换成日期的属性。
     *
     * @var array
     */
    protected $dates = ['deleted_at'];
}
```

当然，你也应该添加 `deleted_at` 字段到数据表中。Laravel [结构生成器](#) 包含了一个用来创建此字段的辅助函数：

```
Schema::table('flights', function ($table) {
    $table->softDeletes();
});
```

现在，当你在模型上调用 `delete` 方法时，`deleted_at` 字段将会被设置成目前的日期和时间。而且，当查询有启用软删除的模型时，被软删除的模型将会自动从所有查询结果中排除。

要确认指定的模型实例是否已经被软删除，可以使用 `trashed` 方法：

```
if ($flight->trashed()) {
    //
}
```

查询被软删除的模型

包含被软删除的模型

如上所述，被软删除的模型将会自动从所有的查询结果中排除。不过，你可以通过在查询中调用 `withTrashed` 方法来强制查询已被软删除的模型：

```
$flights = App\Flight::withTrashed()
    ->where('account_id', 1)
    ->get();
```

`withTrashed` 方法也可以被用在 [关联](#) 查询：

```
$flight->history()->withTrashed()->get();
```

只取出软删除数据

`onlyTrashed` 会只取出软删除数据：

```
$flights = App\Flight::onlyTrashed()
    ->where('airline_id', 1)
    ->get();
```

恢复被软删除的模型

有时候你可能希望「取消删除」一个已被软删除的模型。要恢复一个已被软删除的模型到有效状态，则可在模型实例上使用 `restore` 方法：

```
$flight->restore();
```

你也可以在查询上使用 `restore` 方法来快速地恢复多个模型：

```
App\Flight::withTrashed()
    ->where('airline_id', 1)
    ->restore();
```

与 `withTrashed` 方法类似，`restore` 方法也可以被用在 [关联](#) 查询上：

```
$flight->history()->restore();
```

永久地删除模型

有时候你可能需要真正地从数据库移除模型。要永久地从数据库移除一个已被软删除的模型，则可使用 `forceDelete` 方法：

```
// 强制删除单个模型实例...
$flight->forceDelete();

// 强制删除所有相关模型...
$flight->history()->forceDelete();
```

查询作用域

全局作用域

全局作用域允许我们为给定模型的所有查询添加条件约束。Laravel 自带的 [软删除功能](#) 就使用了全局作用域来从数据库中拉出所有没有被删除的模型。编写自定义的全局作用域可以提供一种方便的、简单的方式，来确保给定模型的每个查询都有特定的条件约束。

编写全局作用域

自定义全局作用域很简单，首先定义一个实现 `Illuminate\Database\Eloquent\Scope` 接口的类，该接口要求你实现一个方法：`apply`。需要的话可以在 `apply` 方法中添加 `where` 条件到查询：

```
<?php

namespace App\Scopes;

use Illuminate\Database\Eloquent\Scope;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Builder;

class AgeScope implements Scope
{
    /**
     * 应用作用域
     *
     * @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);
    }
}
```

{tip} Laravel 没有规定你需要把这些类放置于哪个文件夹，你可以自由在 `app` 文件夹下创建 `Scopes` 文件夹来存放。

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
{
    /**
     * 数据模型的启动方法
     *
     * @return void
     */
    protected static function boot()
    {
        parent::boot();

        static::addGlobalScope(new AgeScope());
    }
}
```

添加作用域后，如果使用 `User::all()` 查询则会生成如下SQL语句：

```
select * from `users` where `age` > 200
```

匿名的全局作用域

Eloquent 还允许我们使用闭包定义全局作用域，这在实现简单作用域的时候特别有用，这样的话，我们就没必要定义一个单独的类了：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Builder;

class User extends Model
{
    /**
     * 数据模型的启动方法
     *
     * @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，利用 `age` 标识符来移除全局作用：

```
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
{
    /**
     * 限制查询只包括受欢迎的用户。
     *
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopePopular($query)
    {

```

```

        return $query->where('votes', '>', 100);
    }

    /**
     * 限制查询只包括活跃的用户。
     *
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopeActive($query)
    {
        return $query->where('active', 1);
    }
}

```

利用查询范围

一旦定义了范围，则可以在查询模型时调用范围方法。在进行方法调用时不需要加上 `scope` 前缀。你甚至可以链式调用不同的范围，如：

```
$users = App\User::popular()->active()->orderBy('created_at')->get();
```

动态范围

有时候，你可能希望定义一个可接受参数的范围。这时只需给你的范围加上额外的参数即可。范围参数应该被定义在 `$query` 参数之后：

```

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 限制查询只包括指定类型的用户。
     *
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopeOfType($query, $type)
    {
        return $query->where('type', $type);
    }
}

```

现在，你可以在范围调用时传递参数：

```
$users = App\User::ofType('admin')->get();
```

事件

Eloquent 模型会触发许多事件，让你在模型的生命周期的多个时间点进行监控：`creating` , `created` , `updating` , `updated` , `saving` , `saved` , `deleting` , `deleted` , `restoring` , `restored` .

事件让你每当有特定的模型类在数据库保存或更新时，执行代码。

当一个新模型被初次保存将会触发 `creating` 以及 `created` 事件。如果一个模型已经存在于数据库且调用了 `save` 方法，将会触发 `updating` 和 `updated` 事件。在这两种情况下都会触发 `saving` 和 `saved` 事件。

开始前，在你的 Eloquent 模型上定义一个 `$events` 属性，将 Eloquent 模型的生命周期的多个点映射到你的 [服务提供者](#) 。

```
<?php

namespace App;

use App\Events\UserSaved;
use App\Events\UserDeleted;
use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * 模型的事件映射。
     *
     * @var array
     */
    protected $events = [
        'saved' => UserSaved::class,
        'deleted' => UserDeleted::class,
    ];
}
```

观察者

如果你在一个给定的模型中监听许多事件，您可以使用观察者将所有监听器变成一个类。观察者类里的方法名应该反映 Eloquent 想监听的事件。每种方法接收 `model` 作为其唯一的参数。 Laravel 不包括观察者默认目录，所以你可以创建任何你喜欢你的目录来存放：

```
<?php
```

```

namespace App\Observers;

use App\User;

class UserObserver
{
    /**
     * 监听用户创建的事件。
     *
     * @param User $user
     * @return void
     */
    public function created(User $user)
    {
        //
    }

    /**
     * 监听用户删除事件。
     *
     * @param User $user
     * @return void
     */
    public function deleting(User $user)
    {
        //
    }
}

```

要注册一个观察者，需要用模型中的 `observe` 方法去观察。你可以在你的服务提供商之一的 `boot` 方法中注册观察者。在这个例子中，我们将在 `AppServiceProvider` 注册观察者：

```

<?php

namespace App\Providers;

use App\User;
use App\Observers\UserObserver;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * 运行所有应用。
     *
     * @return void
     */
    public function boot()
    {

```

```
User::observe(UserObserver::class);  
}  
  
/**  
 * 注册服务提供.  
 *  
 * @return void  
 */  
public function register()  
{  
    //  
}  
}
```

Eloquent: 关联

- [简介](#)
- [定义关联](#)
 - [一对多](#)
 - [一对多（反向关联）](#)
 - [多对多](#)
 - [远层一对多](#)
 - [多态关联](#)
 - [多态多对多关联](#)
- [查找关联](#)
 - [关联方法和动态属性](#)
 - [查找关联是否存在](#)
 - [关联数据计数](#)
- [预加载](#)
 - [预加载条件限制](#)
 - [延迟预加载](#)
- [插入 & 更新关联模型](#)
 - [save 方法](#)
 - [create 方法](#)
 - [更新「从属」关联](#)
 - [多对多关联](#)
- [连动父级时间戳](#)

简介

数据表之间经常会互相进行关联。例如，一篇博客文章可能会有多条评论，或是一张订单可能对应一个下单客户。Eloquent 让管理和处理这些关联变得很容易，同时也支持多种类型的关联：

- [一对多](#)
- [多对多](#)
- [远层一对多](#)
- [多态关联](#)
- [多态多对多关联](#)

定义关联

你可在 Eloquent 模型类内中，把 Eloquent 关联定义成方法（methods）。因为，关联就像 Eloquent 模型一样，也可以作为强大的 [查询语句构造器](#)，定义关联为方法，为其提供了强而有力的链式调用及查找功能。例如，我们可以在 `posts` 关联的链式调用中附加一个约束条件：

```
$user->posts()->where('active', 1)->get();
```

不过，在深入了解使用关联之前，先让我们来学习如何定义每个类型：

一对一

「一对一」关联是一个非常基本的关联关系。举个例子，一个 `User` 模型会关联一个 `Phone` 模型。为了定义这种关联关系，我们需要在 `User` 模型中写一个 `phone` 方法。且 `phone` 方法应该调用 `hasOne` 方法并返回其结果：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 获取与用户关联的电话号码
     */
    public function phone()
    {
        return $this->hasOne('App\Phone');
    }
}
```

第一个传到 `hasOne` 方法里的参数是关联模型的类名。一旦定义好两者之间关联，我们就可以通过使用 Eloquent 的动态属性来获取关联记录。动态属性允许你访问关联方法，如同他们是定义在模型中的属性：

```
$phone = User::find(1)->phone;
```

Eloquent 会假设对应关联的外键名称是基于模型名称的。在这个例子里，它会自动假设 `Phone` 模型拥有 `user_id` 外键。如果你想要重写这个约定，则可以传入第二个参数到 `hasOne` 方法里。

```
return $this->hasOne('App\Phone', 'foreign_key');
```

此外，Eloquent 假设外键会和上层模型的 `id` 字段（或者自定义的 `$primaryKey`）的值相匹配。换句话说，Eloquent 会寻找用户的 `id` 字段与 `Phone` 模型的 `user_id` 字段的值相同的纪录。如果你想让关联使用 `id` 以外的值，则可以传递第三个参数至 `hasOne` 方法来指定你自定义的键：

```
return $this->hasOne('App\Phone', 'foreign_key', 'local_key');
```

定义反向关联

所以，我们可以从 `User` 模型访问到 `Phone` 模型。现在，让我们在 `Phone` 模型上定义一个关联，此关联能够让我们访问拥有此电话的 `User` 模型。我们可以定义与 `hasOne` 关联相对应的 `belongsTo` 方法：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Phone extends Model
{
    /**
     * 获取拥有该电话的用户模型。
     */
    public function user()
    {
        return $this->belongsTo('App\User');
    }
}
```

在上述例子中，Eloquent 会尝试匹配 `Phone` 模型的 `user_id` 至 `User` 模型的 `id`。Eloquent 判断的默认外键名称参考自关联模型的方法名称，并会在方法名称后面加上 `_id`。当然，如果 `Phone` 模型的外键不是 `user_id`，则可以传递自定义键名作为 `belongsTo` 方法的第二个参数：

```
/**
 * 获取拥有该电话的用户模型。
 */
public function user()
{
    return $this->belongsTo('App\User', 'foreign_key');
}
```

如果你的父级模型不是使用 `id` 作为主键，或是希望以不同的字段来连接下层模型，则可以传递第三个参数至 `belongsTo` 方法来指定父级数据表的自定义键：

```
/**
 * 获取拥有该电话的用户模型。
 */
public function user()
{
    return $this->belongsTo('App\User', 'foreign_key', 'other_key');
}
```

一对多

一个「一对多」关联用于定义单个模型拥有任意数量的其它关联模型。例如，一篇博客文章可能会有无限多个评论。就像其它的 Eloquent 关联一样，可以通过在 Eloquent 模型中写一个函数来定义一对多关联：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
     * 获取这篇博文下的所有评论。
     */
    public function comments()
    {
        return $this->hasMany('App\Comment');
    }
}
```

切记，Eloquent 会自动判断 `comment` 模型上正确的外键字段。按约定来说，Eloquent 会取用自身模型的名称的「Snake Case」，并在后方加上 `_id`。所以，以此例来说，Eloquent 会假设 `Comment` 模型的外键是 `post_id`。

一旦关联被定义，则可以通过 `comments` 属性来访问评论的集合。切记，因为 Eloquent 提供了「动态属性」，因此我们可以对关联方法进行访问，就像他们是在模型中定义的属性一样：

```
$comments = App\Post::find(1)->comments;

foreach ($comments as $comment) {
    //
}
```

当然，因为所有的关联也都提供了查询语句构造器的功能，因此你可以对获取到的评论进一步增加条件，通过调用 `comments` 方法然后在该方法后面链式调用查询条件：

```
$comments = App\Post::find(1)->comments()->where('title', 'foo')->first();
```

就像 `hasOne` 方法，你也可以通过传递额外的参数至 `hasMany` 方法来重写外键与本地键：

```
return $this->hasMany('App\Comment', 'foreign_key');

return $this->hasMany('App\Comment', 'foreign_key', 'local_key');
```

一对多（反向关联）

现在我们已经能访问到所有文章的评论，让我们来接着定义一个通过评论访问所属文章的关联。若要定义相对于 `hasMany` 的关联，可在子级模型定义一个叫做 `belongsTo` 方法的关联函数：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
     * 获取该评论所属的文章模型。
     */
    public function post()
    {
        return $this->belongsTo('App\Post');
    }
}
```

一旦关联被定义之后，则可以通过 `post` 的「动态属性」来获取 `Comment` 模型相对应的 `Post` 模型：

```
$comment = App\Comment::find(1);

echo $comment->post->title;
```

在上述例子中，Eloquent 会尝试将 `Comment` 模型的 `post_id` 与 `Post` 模型的 `id` 进行匹配。Eloquent 判断的默认外键名称参考自关联模型的方法，并在方法名称后面加上 `_id`。当然，如果 `Comment` 模型的外键不是 `post_id`，则可以传递自定义键名作为 `belongsTo` 方法的第二个参数：

```
/**
 * 获取该评论所属的文章模型。
 */
public function post()
{
    return $this->belongsTo('App\Post', 'foreign_key');
}
```

如果你的父级模型不是使用 `id` 作为主键，或是你希望以不同的字段来连接下层模型，则可以传递第三个参数给 `belongsTo` 方法来指定上层数据表的自定义键：

```
/**
 * 获取该评论所属的文章模型。
 */
public function post()
```

```
{
    return $this->belongsTo('App\Post', 'foreign_key', 'other_key');
}
```

多对多

「多对多」关联要稍微比 `hasOne` 及 `hasMany` 关联复杂。如，一个用户可能拥有多种身份，而一种身份能同时被多个用户拥有。举例来说，很多用户都拥有「管理员」的身份。要定义这种关联，需要使用三个数据表：`users`、`roles` 和 `role_user`。`role_user` 表命名是以相关联的两个模型数据表来依照字母顺序命名，并包含了 `user_id` 和 `role_id` 字段。

多对多关联通过编写一个在自身 Eloquent 类调用的 `belongsToMany` 的方法来定义。举个例子，让我们在 `User` 模型中定义 `roles` 方法：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 属于该用户的身份。
     */
    public function roles()
    {
        return $this->belongsToMany('App\Role');
    }
}
```

一旦关联被定义，则可以使用 `roles` 动态属性来访问用户的身份：

```
$user = App\User::find(1);

foreach ($user->roles as $role) {
    //
}
```

当然，就如所有其它的关联类型一样，你也可以调用 `roles` 方法并在该关联之后链式调用查询条件：

```
$roles = App\User::find(1)->roles()->orderBy('name')->get();
```

如前文提到那样，Eloquent 会合并两个关联模型的名称并依照字母顺序命名。当然你也可以随意重写这个约定。可通过传递第二个参数至 `belongsToMany` 方法来实现：

```
return $this->belongsToMany('App\Role', 'role_user');
```

除了自定义合并数据表的名称，你也可以通过传递额外参数至 `belongsToMany` 方法来自定义数据表里的键的字段名称。第三个参数是你定义在关联中的模型外键名称，而第四个参数则是你要合并的模型外键名称：

```
return $this->belongsToMany('App\Role', 'role_user', 'user_id', 'role_id');
```

定义相对的关联

要定义相对于多对多的关联，只需简单的放置另一个名为 `belongsToMany` 的方法到你关联的模型上。让我们接着以用户身份为例，在 `Role` 模型中定义 `users` 方法：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Role extends Model
{
    /**
     * 属于该身份的用户。
     */
    public function users()
    {
        return $this->belongsToMany('App\User');
    }
}
```

如你所见，此定义除了简单的参考 `App\User` 模型外，与 `User` 的对应完全相同。因为我们重复使用了 `belongsToMany` 方法，当定义相对于多对多的关联时，所有常用的自定义数据表与键的选项都是可用的。

获取中间表字段

正如你所知，要操作多对多关联需要一个中间数据表。Eloquent 提供了一些有用的方法来和这张表进行交互。例如，假设 `User` 对象关联到很多的 `Role` 对象。访问这些关联对象时，我们可以在模型中使用 `pivot` 属性来访问中间数据表的数据：

```
$user = App\User::find(1);

foreach ($user->roles as $role) {
    echo $role->pivot->created_at;
}
```

注意我们取出的每个 `Role` 模型对象，都会被自动赋予 `pivot` 属性。此属性代表中间表的模型，它可以像其它的 Eloquent 模型一样被使用。

默认情况下，`pivot` 对象只提供模型的键。如果你的 pivot 数据表包含了其它的属性，则可以在定义关联方法时指定那些字段：

```
return $this->belongsToMany('App\Role')->withPivot('column1', 'column2');
```

如果你想要中间表自动维护 `created_at` 和 `updated_at` 时间戳，可在定义关联方法时加上 `withTimestamps` 方法：

```
return $this->belongsToMany('App\Role')->withTimestamps();
```

使用中间表来过滤关联数据

你可以使用 `wherePivot` 和 `wherePivotIn` 来增加中间件表过滤条件：

```
return $this->belongsToMany('App\Role')->wherePivot('approved', 1);

return $this->belongsToMany('App\Role')->wherePivotIn('priority', [1, 2]);
```

定义自定义中间表模型

如果你想定义一个自定义模型来表示你中间表的关联，则可以在定义关联时调用 `using` 方法。所有用来表示中间表关联的自定义模型必须扩展自 `Illuminate\Database\Eloquent\Relations\Pivot` 类：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Role extends Model
{
    /**
     * 属于该身份的用户。
     */
    public function users()
    {
        return $this->belongsToMany('App\User')->using('App\UserRole');
    }
}
```

远层一对多

「远层一对多」提供了方便简短的方法来通过中间的关联获取远层的关联。例如，一个 `Country` 模型可能通过中间的 `Users` 模型关联到多个 `Posts` 模型。让我们来看看定义此种关联的数据表：

```

countries
    id - integer
    name - string

users
    id - integer
    country_id - integer
    name - string

posts
    id - integer
    user_id - integer
    title - string

```

虽然 `posts` 本身不包含 `country_id` 字段，但 `hasManyThrough` 关联通过 `$country->posts` 来让我们可以访问一个国家的文章。若运行此查找，则 Eloquent 会检查中间表 `users` 的 `country_id`。在找到匹配的用户 ID 后，就会在 `posts` 数据表中使用它们来进行查找。

现在我们已经检查完了关联的数据表结构，让我们来接着在 `Country` 模型中定义它：

```

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Country extends Model
{
    /**
     * 获取该国家的所有文章。
     */
    public function posts()
    {
        return $this->hasManyThrough('App\Post', 'App\User');
    }
}

```

`hasManyThrough` 方法的第一个参数为我们希望最终访问的模型名称，而第二个参数为中间模型的名称。

当运行关联查找时，通常会使用 Eloquent 的外键约定。如果你想要自定义关联的键，则可以将它们传递至 `hasManyThrough` 方法的第三与第四个参数。第三个参数为中间模型的外键名称，而第四个参数为最终模型的外键名称，第五个参数则为本地键。

```
class Country extends Model
```

```
{
    public function posts()
    {
        return $this->hasManyThrough(
            'App\Post', 'App\User',
            'country_id', 'user_id', 'id'
        );
    }
}
```

多态关联

数据表结构

多态关联允许一个模型在单个关联中从属一个以上其它模型。举个例子，想象一下使用你应用的用户可以「评论」文章和视频。使用多态关联关系，您可以使用一个 `comments` 数据表就可以同时满足两个使用场景。首先，让我们观察一下用来创建这关联的数据表结构：

```
posts
    id - integer
    title - string
    body - text

videos
    id - integer
    title - string
    url - string

comments
    id - integer
    body - text
    commentable_id - integer
    commentable_type - string
```

有两个需要注意的字段是 `comments` 表中的 `commentable_id` 和 `commentable_type`。其中，`commentable_id` 用于存放文章或者视频的 id，而 `commentable_type` 用于存放所属模型的类名。注意的是，`commentable_type` 是当我们访问 `commentable` 关联时，ORM 用于判断所属的模型是哪个「类型」。

模型结构

接着，让我们来查看创建这种关联所需的模型定义：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
```

```

class Comment extends Model
{
    /**
     * 获取所有拥有的 commentable 模型。
     */
    public function commentable()
    {
        return $this->morphTo();
    }
}

class Post extends Model
{
    /**
     * 获取所有文章的评论。
     */
    public function comments()
    {
        return $this->morphMany('App\Comment', 'commentable');
    }
}

class Video extends Model
{
    /**
     * 获取所有视频的评论。
     */
    public function comments()
    {
        return $this->morphMany('App\Comment', 'commentable');
    }
}

```

获取多态关联

一旦你的数据表及模型被定义，则可以通过模型来访问关联。例如，若要访问某篇文章的所有评论，则可以简单的使用 `comments` 动态属性：

```

$post = App\Post::find(1);

foreach ($post->comments as $comment) {
    //
}

```

你也可以从多态模型的多态关联中，通过访问调用 `morphTo` 的方法名称来获取拥有者，也就是此例子中 `Comment` 模型的 `commentable` 方法。所以，我们可以使用动态属性来访问这个方法：

```
$comment = App\Comment::find(1);
```

```
$commentable = $comment->commentable;
```

`Comment` 模型的 `commentable` 关联会返回 `Post` 或 `Video` 实例，这取决于评论所属模型的类型。

自定义多态关联的类型字段

默认情况下，Laravel 会使用「包含命名空间的类名」作为多态表的类型区分，例如，`comment` 属于 `Post` 或者 `Video`，`commentable_type` 的默认值可以分别是 `App\Post` 或者 `App\Video`。然而，你也许会想使用应用中的内部结构来对数据库进行解耦。在这个例子中，你可以定义一个「多态对照表」来指引 Eloquent 对各个模型使用自定义名称而非类名：

```
use Illuminate\Database\Eloquent\Relations\Relation;

Relation::morphMap([
    'posts' => App\Post::class,
    'videos' => App\Video::class,
]);
```

你需要在你自己的 `AppServiceProvider` 中的 `boot` 函数注册这个 `morphMap`，或者创建一个独立且满足你要求的服务提供者。

多态多对多关联

数据表结构

除了一般的多态关联，你也可以定义「多对多」的多态关联。例如，博客的 `Post` 和 `Video` 模型可以共用多态关联至 `Tag` 模型。使用多对多的多态关联能够让你的博客文章及图片共用独立标签的单个列表。首先，让我们先来查看数据表结构：

```
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

接着，我们已经准备好定义模型的关联。`Post` 及 `Video` 模型都会拥有 `tags` 方法，并在该方法内调用自身 Eloquent 类的 `morphToMany` 方法：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
     * 获取该文章的所有标签。
     */
    public function tags()
    {
        return $this->morphToMany('App\Tag', 'taggable');
    }
}
```

定义相对的关联

然后，在 `Tag` 模型上，你必须为每个要关联的模型定义一个方法。因此，在这个例子中，我们需要定义一个 `posts` 方法及一个 `videos` 方法：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Tag extends Model
{
    /**
     * 获取所有被赋予该标签的文章。
     */
    public function posts()
    {
        return $this->morphedByMany('App\Post', 'taggable');
    }

    /**
     * 获取所有被赋予该标签的图片。
     */
    public function videos()
    {
```

```

        return $this->morphedByMany('App\Video', 'taggable');
    }
}

```

获取关联

一旦你的数据表及模型被定义，则可以通过你的模型来访问关联。例如，你可以简单的使用 `tags` 动态属性来访问文章的所有标签：

```

$post = App\Post::find(1);

foreach ($post->tags as $tag) {
    //
}

```

你也可以从多态模型的多态关联中，通过访问运行调用 `morphedByMany` 的方法名称来获取拥有者。在此例子中，就是 `Tag` 模型的 `posts` 或 `videos` 方法。因此，你可以通过访问使用动态属性来访问这个方法：

```

$tag = App\Tag::find(1);

foreach ($tag->videos as $video) {
    //
}

```

查找关联

所有类型的 Eloquent 关联都是通过方法来定义的，你可以通过调用这些方法来获得关联的一个实例，而不需要实际运行关联的查找。此外，所有类型的 Eloquent 关联也提供了 [查询语句构造器](#) 的功能，让你能够在数据库运行该 SQL 前，在关联查找后面链式调用条件。

例如，假设有一个博客系统，其中 `User` 模型拥有很多关联的 `Post` 模型：

```

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 获取该用户的所有文章。
     */
    public function posts()
    {
        return $this->hasMany('App\Post');
    }
}

```

```

    }
}

```

你可以查找 `posts` 关联并增加额外的条件至关联，像这样：

```

$user = App\User::find(1);

$user->posts()->where('active', 1)->get();

```

你可以在关联中使用 [查询语句构造器](#) 中的任意方法，所以，欢迎查阅查询语句构造器的相关文档以便了解那些有助于你实现的方法。

关联方法与动态属性

如果你不需要增加额外的条件至 Eloquent 的关联查找，则可以简单的像访问属性一样来访问关联。例如我们刚刚的 `User` 及 `Post` 模型示例，我们可以像这样来访问所有用户的文章：

```

$user = App\User::find(1);

foreach ($user->posts as $post) {
    //
}

```

动态属性是「延迟加载」的，意味着它们只会在被访问的时候才加载关联数据。正因为如此，开发者通常需要使用 [预加载](#) 来预先加载关联数据，关联数据将会在模型加载后被访问。预加载能有效减少你的 SQL 查询语句。

查找关联是否存在

当访问模型的纪录时，你可能希望根据关联的存在来对结果进行限制。比方说你想获取博客中那些至少拥有一条评论的文章。则可以通过传递名称至关联的 `has` 方法来实现：

```

// 获取那些至少拥有一条评论的文章...
$posts = App\Post::has('comments')->get();

```

你也可以制定运算符及数量来进一步自定义查找：

```

// 获取所有至少有三条评论的文章...
$posts = Post::has('comments', '>=', 3)->get();

```

也可以使用「点」符号来构造嵌套的 `has` 语句。例如，你可能想获取那些至少有一条评论被投票的文章：

```

// 获取所有至少有一条评论被评分的文章...
$posts = Post::has('comments.votes')->get();

```

如果你想要更高级的用法，则可以使用 `whereHas` 和 `orWhereHas` 方法，在 `has` 查找里设置「`where`」条件。此方法可以让你增加自定义条件至关联条件中，例如对评论内容进行检查：

```
// 获取那些至少有一条评论包含 foo 的文章
$posts = Post::whereHas('comments', function ($query) {
    $query->where('content', 'like', 'foo%');
})->get();
```

关联数据计数

如果你想对关联数据进行计数但又不想再发起单独的 SQL 请求，你可以使用 `withCount` 方法，此方法会在你的结果集中增加一个 `{relation}_count` 字段：

```
$posts = App\Post::withCount('comments')->get();

foreach ($posts as $post) {
    echo $post->comments_count;
}
```

你还可以像在查询语句中添加约束一样，获取多重关联的「计数」。

```
$posts = Post::withCount(['votes', 'comments' => function ($query) {
    $query->where('content', 'like', 'foo%');
}])->get();

echo $posts[0]->votes_count;
echo $posts[0]->comments_count;
```

预加载

当通过属性访问 Eloquent 关联时，该关联数据会被「延迟加载」。意味着该关联数据只有在你使用属性访问它时才会被加载。不过，Eloquent 可以在你查找上层模型时「预加载」关联数据。预加载避免了 $N + 1$ 查找的问题。要说明 $N + 1$ 查找的问题，可试想一个关联到 `Author` 的 `Book` 模型，如下所示：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Book extends Model
{
    /**
```

```

    * 获取编写该书的作者。
    */
public function author()
{
    return $this->belongsTo('App\Author');
}
}

```

现在，让我们来获取所有书籍及其作者的数据：

```

$books = App\Book::all();

foreach ($books as $book) {
    echo $book->author->name;
}

```

上方的循环会运行一次查找并取回所有数据表上的书籍，接着每本书会运行一次查找作者的操作。因此，若存在着 25 本书，则循环就会执行 26 次查找：1 次是查找所有书籍，其它 25 次则是在查找每本书的作者。

很幸运地，我们可以使用预加载来将查找的操作减少至 2 次。可在查找时使用 `with` 方法来指定想要预加载的关联数据：

```

$books = App\Book::with('author')->get();

foreach ($books as $book) {
    echo $book->author->name;
}

```

对于该操作则只会执行两条 SQL 语句：

```

select * from books

select * from authors where id in (1, 2, 3, 4, 5, ...)

```

预加载多种关联

有时你可能想要在单次操作中预加载多种不同的关联。要这么做，只需传递额外的参数至 `with` 方法即可：

```
$books = App\Book::with('author', 'publisher')->get();
```

嵌套预加载

若要预加载嵌套关联，则可以使用「点」语法。例如，让我们在一个 Eloquent 语法中，预加载所有书籍的作者，及所有作者的个人联系方式：

```
$books = App\Book::with('author.contacts')->get();
```

预加载条件限制

有时你可能想要预加载关联，并且指定预加载查询的额外条件。如下所示：

```
$users = App\User::with(['posts' => function ($query) {
    $query->where('title', 'like', '%first%');
}])->get();
```

在这个例子里，Eloquent 只会预加载标题包含 `first` 的文章。当然，你也可以调用其它的 [查询语句构造器](#) 来进一步自定义预加载的操作：

```
$users = App\User::with(['posts' => function ($query) {
    $query->orderBy('created_at', 'desc');
}])->get();
```

延迟预加载

有时你可能需要在上层模型被获取后才预加载关联。当你需要来动态决定是否加载关联模型时会很有帮助，如下所示：

```
$books = App\Book::all();

if ($someCondition) {
    $books->load('author', 'publisher');
}
```

如果你想设置预加载查询的额外条件，则可以传递一个键值为你想要的关联的数组至 `load` 方法。这个数组的值应是用于接收查询实例的 闭包 实例：

```
$books->load(['author' => function ($query) {
    $query->orderBy('published_date', 'asc');
}]);
```

写入关联模型

Save 方法

Eloquent 提供了便捷的方法来将新的模型增加至关联中。例如，将新的 `Comment` 写入至 `Post` 模型。除了手动设置 `comment` 的 `post_id` 属性外，你也可以直接使用关联的 `save` 方法来写入 `Comment`：

```
$comment = new App\Comment(['message' => 'A new comment.']);

$post = App\Post::find(1);

$post->comments()->save($comment);
```

注意我们并没有使用动态属性来访问 `comments` 关联。相反地，我们调用了 `comments` 方法来获取关联的实例。`save` 方法会自动在新的 `Comment` 模型中增加正确的 `post_id` 值。

如果你需要保存多个关联模型，则可以使用 `saveMany` 方法：

```
$post = App\Post::find(1);

$post->comments()->saveMany([
    new App\Comment(['message' => 'A new comment.']),
    new App\Comment(['message' => 'Another comment.']),
]);
```

Create 方法

除了 `save` 与 `saveMany` 方法外，你也可以使用 `create` 方法，该方法允许传入属性的数组来建立模型并写入数据库。`save` 与 `create` 的不同之处在于，`save` 允许传入一个完整的 Eloquent 模型实例，但 `create` 只允许传入原始的 PHP 数组：

```
$post = App\Post::find(1);

$comment = $post->comments()->create([
    'message' => 'A new comment.',
]);
```

在使用 `create` 方法前，请确认你已浏览了文档的 [批量赋值](#) 章节。

更新「从属」关联

当更新一个 `belongsTo` 关联时，可以使用 `associate` 方法。此方法会将外键设置到下层模型：

```
$account = App\Account::find(10);

$user->account()->associate($account);

$user->save();
```

当删除一个 `belongsTo` 关联时，你可以使用 `dissociate` 方法。此方法会置该关联的外键为空 (`null`)：

```
$user->account()->dissociate();

$user->save();
```

多对多关联

附加与卸除

当使用多对多关联时， Eloquent 提供了一些额外的辅助函数让操作关联模型更加方便。例如，让我们假设一个用户可以拥有多个身份，且每个身份都可以被多个用户拥有。要附加一个规则至一个用户，并连接模型以及将记录写入至中间表，则可以使用 `attach` 方法：

```
$user = App\User::find(1);

$user->roles()->attach($roleId);
```

当附加一个关联至模型时，你也可以传递一个需被写入至中间表的额外数据数组：

```
$user->roles()->attach($roleId, ['expires' => $expires]);
```

当然，我们有时也需要来移除用户的身份。要移除一个多对多的纪录，可使用 `detach` 方法。`detach` 方法会从中间表中移除正确的纪录；当然，这两个模型依然会存在于数据库中：

```
// 移除用户身上某一身份...
$user->roles()->detach($roleId);

// 移除用户身上所有身份...
$user->roles()->detach();
```

为了方便，`attach` 与 `detach` 都允许传入 ID 数组：

```
$user = App\User::find(1);

$user->roles()->detach([1, 2, 3]);

$user->roles()->attach([1 => ['expires' => $expires], 2, 3]);
```

更新关联

你也可以使用 `sync` 方法去创建一个多对多的关联。`sync` 方法可以用数组形式的 IDs 插入中间的数据表。任何一个不存在于给定数组的 IDs 将会在中间表内被删除。所以，操作完成之后，只有那些在给定数组内的 IDs 会被保留在中间表中。

```
$user->roles()->sync([1, 2, 3]);
```

你也可以通过 IDs 传递其他的附加中间表值：

```
$user->roles()->sync([1 => ['expires' => true], 2, 3]);
```

如果你不想分离现有的 IDs，你可以 `syncWithoutDetaching` 方法：

```
$user->roles()->syncWithoutDetaching([1, 2, 3]);
```

在中间表上保存额外数据

处理多对多关联时，`save` 方法接收额外中间表属性数组作为第二个参数：

```
App\User::find(1)->roles()->save($role, ['expires' => $expires]);
```

更新中间表记录

如果你需要更新中间表已存在的记录，可以使用 `updateExistingPivot` 方法。这个方法接收中间记录的外键和属性数组进行更新：

```
$user = App\User::find(1);

$user->roles()->updateExistingPivot($roleId, $attributes);
```

连动父级时间戳

当一个模型 `belongsTo` 或 `belongsToMany` 另一个模型时，像是一个 `Comment` 属于一个 `Post`。这对于子级模型被更新时，要更新父级的时间戳相当有帮助。举例来说，当一个 `Comment` 模型被更新时，你可能想要「连动」更新 `Post` 所属的 `updated_at` 时间戳。Eloquent 使得此事相当容易。只要在关联的下层模型中增加一个包含名称的 `touches` 属性即可：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
     * 所有的关联将会被连动。
     *
     * @var array
     */
    protected $touches = ['post'];
}
```

```
/**  
 * 获取拥有此评论的文章。  
 */  
public function post()  
{  
    return $this->belongsTo('App\Post');  
}  
}
```

现在，当你更新一个 Comment 时，它所属的 Post 拥有的 `updated_at` 字段也会被同时更新，使其更方便的得知何时让一个 `Post` 模型的缓存失效：

```
$comment = App\Comment::find(1);  
  
$comment->text = 'Edit to this comment!';  
  
$comment->save();
```

Eloquent: 集合

- [简介](#)
- [可用的方法](#)
- [自定义集合](#)

简介

默认情况下 Eloquent 返回的都是一个 `Illuminate\Database\Eloquent\Collection` 对象的实例，包含通过 `get` 方法或是访问一个关联来获取到的结果。Eloquent 集合对象继承了 Laravel [集合基类](#)，因此它自然也继承了许多可用于与 Eloquent 模型交互的方法。

当然，所有集合都可以作为迭代器，来让你像遍历一个 PHP 数组一样来遍历一个集合：

```
$users = App\User::where('active', 1)->get();

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

然而，集合比数组更强大的地方是其使用了各种 `map` / `reduce` 的直观操作。例如，我们移除所有未激活的用户模型和收集其余各个用户的名字：

```
$users = App\User::where('active', 1)->get();

$names = $users->reject(function ($user) {
    return $user->active === false;
})
->map(function ($user) {
    return $user->name;
});
```

{note} 大部分的 Eloquent 集合会返回新的「Eloquent 集合」实例，但是 `pluck`，`keys`，`zip`，`collapse`，`flatten` 和 `flip` 方法会返回 [基础集合](#) 实例。

相应的，如果一个 `map` 操作返回一个不包含任何 Eloquent 模型的集合，那么它将会自动转换成基础集合。

可用的方法

集合对象

所有 Eloquent 集合都继承了基础的 [Laravel 集合](#) 对象。因此，他们也继承了所有集合类提供的强大的方法：

```
[all]//collections#method-all) [avg]//collections#method-avg) [chunk]
(/docs//collections#method-chunk) [collapse]//collections#method-collapse) [combine]
(/docs//collections#method-combine) [contains]//collections#method-contains) [count]
(/docs//collections#method-count) [diff]//collections#method-diff) [diffKeys]
(/docs//collections#method-diffkeys) [each]//collections#method-each) [every]
(/docs//collections#method-every) [except]//collections#method-except) [filter]
(/docs//collections#method-filter) [first]//collections#method-first) [flatMap]
(/docs//collections#method-flatmap) [flatten]//collections#method-flatten) [flip]
(/docs//collections#method-flip) [forget]//collections#method-forget) [forPage]
(/docs//collections#method-forpage) [get]//collections#method-get) [groupBy]
(/docs//collections#method-groupby) [has]//collections#method-has) [implode]
(/docs//collections#method-implode) [intersect]//collections#method-intersect) [isEmpty]
(/docs//collections#method-isempty) [keyBy]//collections#method-keyby) [keys]
(/docs//collections#method-keys) [last]//collections#method-last) [map]
(/docs//collections#method-map) [max]//collections#method-max) [merge]
(/docs//collections#method-merge) [min]//collections#method-min) [only]
(/docs//collections#method-only) [pluck]//collections#method-pluck) [pop]
(/docs//collections#method-pop) [prepend]//collections#method-prepend) [pull]
(/docs//collections#method-pull) [push]//collections#method-push) [put]
(/docs//collections#method-put) [random]//collections#method-random) [reduce]
(/docs//collections#method-reduce) [reject]//collections#method-reject) [reverse]
(/docs//collections#method-reverse) [search]//collections#method-search) [shift]
(/docs//collections#method-shift) [shuffle]//collections#method-shuffle) [slice]
(/docs//collections#method-slice) [sort]//collections#method-sort) [sortBy]
(/docs//collections#method-sortby) [sortByDesc]//collections#method-sortbydesc) [splice]
(/docs//collections#method-splice) [sum]//collections#method-sum) [take]
(/docs//collections#method-take) [toArray]//collections#method-toarray) [toJson]
(/docs//collections#method-tojson) [transform]//collections#method-transform) [union]
(/docs//collections#method-union) [unique]//collections#method-unique) [values]
(/docs//collections#method-values) [where]//collections#method-where) [whereStrict]
(/docs//collections#method-wherestrict) [whereIn]//collections#method-wherein)
[whereInLoose]//collections#method-whereinloose) [zip]//collections#method-zip)
```

自定义集合

如果你需要使用一个自定义的 `Collection` 对象到自己的扩充方法上，则可以在模型中重写

`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 集合实例对象。
     *
     * @param array $models
     * @return \Illuminate\Database\Eloquent\Collection
     */
    public function newCollection(array $models = [])
    {
        return new CustomCollection($models);
    }
}
```

一旦你定义了 `newCollection` 方法，则可在任何 Eloquent 返回该模型的 `Collection` 实例时，接收到一个你的自定义集合的实例。如果你想要在应用程序的每个模型中使用自定义集合，则应该在所有的模型继承的模型基类中重写 `newCollection` 方法。

Eloquent: 修改器

- 简介
- 访问器 & 修改器
 - 定义一个访问器
 - 定义一个修改器
- 日期转换器
- 属性类型转换
 - 数组 & JSON 转换

简介

访问器和修改器可以让你修改 Eloquent 模型中的属性或者设置它们的值，例如，你可能想要使用 [Laravel 加密器](#) 来加密一个被保存在数据库中的值，当你从 Eloquent 模型访问该属性时该值将被自动解密。

除了自定义访问器和修改器之外，Eloquent 也会自动将日期字段类型转换成 [Carbon](#) 实例或将 [文本字符串类型转换成 JSON](#)。

访问器 & 修改器

定义一个访问器

若要定义一个访问器，则须在你的模型上创建一个 `getFooAttribute` 方法。要访问的 `Foo` 字段需使用「驼峰式」来命名。在这个例子中，我们将为 `first_name` 属性定义一个访问器。当 Eloquent 尝试获取 `first_name` 的值时，将会自动调用此访问器：

```
<?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);
    }
}
```

```
}
```

如你所见，字段的原始值被传递到访问器中，让你可以编辑修改并返回结果。如果要访问被修改的值，则可以像这样来访问 `first_name` 属性：

```
$user = App\User::find(1);

$firstName = $user->first_name;
```

定义一个修改器

若要定义一个修改器，则须在模型上定义一个 `setFooAttribute` 方法。要访问的 `Foo` 字段需使用「驼峰式」来命名。让我们再来定义 `first_name` 属性的修改器。当我们尝试在模型上设置 `first_name` 的值时，将会自动调用此修改器：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 设定用户的名字。
     *
     * @param string $value
     * @return void
     */
    public function setFirstNameAttribute($value)
    {
        $this->attributes['first_name'] = strtolower($value);
    }
}
```

修改器会获取属性已经被设置的值，让你可以操作该值并将其设置到 Eloquent 模型内部的 `$attributes` 属性上。举个例子，如果我们尝试将 `first_name` 属性设置成 `Sally`：

```
$user = App\User::find(1);

$user->first_name = 'Sally';
```

在这个例子中，`setFirstNameAttribute` 函数将会使用 `Sally` 作为参数来调用。修改器会对该名字使用 `strtolower` 函数并将其值设置于内部的 `$attributes` 数组。

日期转换器

默认情况下， Eloquent 将会把 `created_at` 和 `updated_at` 字段转换成 [Carbon](#) 实例，它提供了各种各样的方法，并继承了 PHP 原生的 `DateTime` 类。

你可以在模型中自定义哪些字段需要被自动修改，或完全禁止修改，可通过重写模型的 `$dates` 属性来实现：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 应被转换为日期的属性。
     *
     * @var array
     */
    protected $dates = [
        'created_at',
        'updated_at',
        'deleted_at'
    ];
}
```

当某个字段被认为是日期时，你或许想将其数值设置成一个 UNIX 时间戳、日期字符串（`Y-m-d`）或日期时间（`date-time`）字符串，当然还有 `DateTime` 或 `Carbon` 实例，然后日期数值将会被自动保存到数据库中：

```
$user = App\User::find(1);

$user->deleted_at = Carbon::now();

$user->save();
```

如上所述，在 `$dates` 属性中列出的所有属性被获取到时，都将会自动转换成 [Carbon](#) 实例，让你可在属性上使用任何 `Carbon` 方法：

```
$user = App\User::find(1);

return $user->deleted_at->getTimestamp();
```

时间格式

默认情况下，时间戳将会以 `'Y-m-d H:i:s'` 格式化。如果你想要自定义自己的时间戳格式，可在模型中设置 `$dateFormat` 属性。该属性定义了时间属性应如何被保存到数据库，以及模型应被序列化成一个数组或 JSON 格式：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * 模型的数据字段的保存格式。
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

属性类型转换

`$casts` 属性在模型中提供了将属性转换为常见的数据类型的方法。`$casts` 属性应是一个数组，且键是那些需要被转换的属性名称，值则是代表字段要转换的类型。支持的转换的类型有：

- `integer`
- `real`
- `float`
- `double`
- `string`
- `boolean`
- `object`
- `array`
- `collection`
- `date`
- `datetime`
- `timestamp`

例如，`is_admin` 属性以整数（`0` 或 `1`）被保存在我们的数据库中，让我们来把它转换为布尔值：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
```

```
class User extends Model
{
    /**
     * 应该被转换成原生类型的属性。
     *
     * @var array
     */
    protected $casts = [
        'is_admin' => 'boolean',
    ];
}
```

现在当你访问 `is_admin` 属性时，它将会被转换成布尔值，即便保存在数据库里的值是一个整数：

```
$user = App\User::find(1);

if ($user->is_admin) {
    //
}
```

数组 & JSON 转换

若原本字段保存的是被序列化的 JSON，则 `array` 类型转换将会特别有用。例如，在你的数据库中有一个 `JSON` 或 `TEXT` 字段类型，其包含了被序列化的 JSON，且对该属性添加了 `array` 类型转换。当你在 Eloquent 模型上访问该属性时，它将会被自动反序列化成一个 PHP 数组：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 应该被转换成原生类型的属性。
     *
     * @var array
     */
    protected $casts = [
        'options' => 'array',
    ];
}
```

一旦类型转换被定义，则可以访问 `options` 属性，它将会自动把 JSON 反序列化成一个 PHP 数组。当你设置 `options` 属性的值时，指定的数组将会被自动序列化成 JSON 以便进行保存：

```
$user = App\User::find(1);

$options = $user->options;

$options['key'] = 'value';

$user->options = $options;

$user->save();
```

Eloquent: 序列化

- 简介
- 序列化模型 & 集合
 - 序列化成数组
 - 序列化成 JSON
- 隐藏来自 JSON 的属性
- 添加参数到 JSON 中

简介

当你在创建 JSON API 的时候，经常会需要将模型和关联转换成数组或 JSON。Eloquent 提供了一些便捷的方法来让我们可以完成这些转换，以及控制哪些属性需要被包括在序列化中。

序列化模型 & 集合

序列化成数组

如果要将模型还有其加载的 [关联](#) 转换成一个数组，则可以使用 `toArray` 方法。这个方法是递归的，因此，所有属性和关联（包含关联中的关联）都会被转换成数组：

```
$user = App\User::with('roles')->first();

return $user->toArray();
```

你也可以将整个 [集合](#) 转换成数组：

```
$users = App\User::all();

return $users->toArray();
```

序列化成 JSON

如果要将模型转换成 JSON，则可以使用 `toJson` 方法。如同 `toArray` 方法一样，`toJson` 方法也是递归的。因此，所有的属性以及关联都会被转换成 JSON：

```
$user = App\User::find(1);

return $user->toJson();
```

或者，你也可以强制把一个模型或集合转型成一个字符串，它将会自动调用 `toJson` 方法：

```
$user = App\User::find(1);

return (string) $user;
```

当模型或集合被转型成字符串时，模型或集合便会被转换成 `JSON` 格式，因此你可以直接从应用程序的路由或者控制器中返回 `Eloquent` 对象：

```
Route::get('users', function () {
    return App\User::all();
});
```

隐藏来自 JSON 的属性

有时候你可能会想要限制包含在模型数组或 JSON 表示中的属性，比如说密码。则可以通过在模型中增加 `$hidden` 属性定义来实现：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 在数组中想要隐藏的属性。
     *
     * @var array
     */
    protected $hidden = ['password'];
}
```

{note} 当你要对关联进行隐藏时，需使用关联的方法名称，而不是它的动态属性名称。

另外，你也可以使用 `visible` 属性来定义应该包含在你的模型数组和 JSON 表示中的属性白名单。白名单外的其他属性将隐藏，不会出现在转换后的数组或 JSON 中：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
```

```

    * 在数组中可见的属性。
    *
    * @var array
    */
protected $visible = ['first_name', 'last_name'];
}

```

临时修改属性的可见度

你可以在模型实例后使用 `makeVisible` 方法来显示通常隐藏的属性，且为了便于使用，`makeVisible` 方法会返回一个模型实例：

```
return $user->makeVisible('attribute')->toArray();
```

相应的，你可以在模型实例后使用 `makeHidden` 方法来隐藏通常显示的属性：

```
return $user->makeHidden('attribute')->toArray();
```

添加参数到 JSON 中

有时候，在转换模型到 数组 或 JSON 时，你希望添加一个在数据库中没有对应字段的属性。首先你需要为这个值定义一个 访问器：

```

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 为用户获取管理者的标记。
     *
     * @return bool
     */
    public function getIsAdminAttribute()
    {
        return $this->attributes['admin'] == 'yes';
    }
}

```

访问器创建成功后，只需添加该属性到改模型的 `appends` 属性中。注意，属性名称通常遵循「[Snake Case](#)」的命名方式，即是访问器的名称是基于「[Camel Case](#)」的命名方式。

```
<?php
```

```
namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * 访问器被附加到模型数组的形式。
     *
     * @var array
     */
    protected $appends = ['is_admin'];
}
```

一旦属性被添加到 `appends` 清单，便会将模型中的数组和 JSON 这两种形式都包含进去。在 `appends` 数组中的属性也遵循模型中 `visible` 和 `hidden` 设置。

11 测试

Laravel 测试: 入门指南

- [简介](#)
- [测试环境](#)
- [定义并运行测试](#)

简介

Laravel 天生就具有测试的基因。事实上，Laravel 默认就支持用 PHPUnit 来做测试，并为你的应用程序配置好了 `phpunit.xml` 文件。框架还提供了一些便利的辅助函数，让你可以更直观的测试应用程序。

默认在你应用的 `tests` 目录下包含了两个子目录：`Feature` 和 `Unit`。单元测试是针对你代码中相对独立而且非常少的一部分代码来进行测试。实际上，大多数单元测试可能都是针对某一个方法来进行的。功能测试是针对你代码中大部分的代码来进行测试，包括几个对象的相互作用，甚至是一个完整的 HTTP 请求 JSON 实例。

在 `Feature` 和 `Unit` 目录中都有提供一个 `ExampleTest.php` 的示例文件。安装新的 Laravel 应用程序之后，只需在命令行上运行 `phpunit` 就可以进行测试。

测试环境

在运行测试时，Laravel 会根据 `phpunit.xml` 文件中设定好的环境变量自动将环境变量设置为 `testing`，并将 Session 及缓存以 `array` 的形式存储，也就是说在测试时不会持久化任何 Session 或缓存数据。

你可以随意创建其它必要的测试环境配置。`testing` 环境的变量可以在 `phpunit.xml` 文件中被修改，但是在运行测试之前，请确保使用 `config:clear` Artisan 命令来清除配置信息的缓存。

定义并运行测试

可以使用 `make:test` Artisan 命令，创建一个测试用例：

```
// 在 Feature 目录下创建一个测试类...
php artisan make:test UserTest

// 在 Unit 目录下创建一个测试类...
php artisan make:test UserTest --unit
```

测试类生成之后，你就可以像正常使用 PHPUnit 一样来定义测试方法。要运行测试只需要在终端上运行 `phpunit` 命令即可：

```
<?php
```

```
namespace Tests\Unit;

use Tests\TestCase;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    /**
     * 基本的测试用例。
     *
     * @return void
     */
    public function testBasicTest()
    {
        $this->assertTrue(true);
    }
}
```

{note} 如果要在你的测试类自定义自己的 `setup` 方法, 请确保调用了 `parent::setUp()` 方法。

Laravel 测试之：HTTP 测试

- 基础介绍
- Session / 认证
- 测试 JSON APIs
- 测试文件上传
- 可用的断言方法

基础介绍

Laravel 为 HTTP 请求的生成和发送操作、输出的检查都提供了非常流利的 API。例如，你可以看看下面的这个测试用例：

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function testBasicTest()
    {
        $response = $this->get('/');

        $response->assertStatus(200);
    }
}
```

`get` 方法会创建一个 `GET` 请求来请求你的应用，而 `assertStatus` 方法断言返回的响应是给定的 HTTP 状态码。除了这个简单的断言之外，Laravel 也包含检查响应标头、内容、JSON 结构等等的各种断言。

Session / 认证

Laravel 提供了几个可在测试时使用 Session 的辅助函数。首先，你需要传递一个数组给 `withSession` 方法来设置 Session 数据。这让你在应用程序的测试请求发送之前，先给数据加载 Session 变得简单：

```
<?php

class ExampleTest extends TestCase
{
    public function testApplication()
    {
        $response = $this->withSession(['foo' => 'bar'])
            ->get('/');
    }
}
```

当然，一般使用 Session 时都是用于维持用户的状态，如认证用户。`actingAs` 辅助函数提供了简单的方式来让指定的用户认证为当前的用户。例如，我们可以使用 [模型工厂](#) 来生成并认证用户：

```
<?php

use App\User;

class ExampleTest extends TestCase
{
    public function testApplication()
    {
        $user = factory(User::class)->create();

        $response = $this->actingAs($user)
            ->withSession(['foo' => 'bar'])
            ->get('/');
    }
}
```

你也可以通过传递 guard 名称作为 `actingAs` 的第二参数以指定用户通过哪种 guard 来认证：

```
$this->actingAs($user, 'api')
```

测试 JSON APIs

Laravel 也提供了几个辅助函数来测试 JSON APIs 及其响应。举例来说，`json`，`get`，`post`，`put`，`patch` 和 `delete` 方法可以用于发出各种 HTTP 动作的请求。你也可以轻松的传入数据或标头到这些方法上。首先，让我们来编写一个测试，将 POST 请求发送至 `/user`，并断言其会返回预期数据：

```
<?php
```

```

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $response = $this->json('POST', '/user', ['name' => 'Sally']);

        $response
            ->assertStatus(200)
            ->assertJson([
                'created' => true,
            ]);
    }
}

```

{tip} `assertJson` 方法会将响应转换为数组并且利用 `PHPUnit::assertArraySubset` 方法来验证传入的数组是否在应用返回的 JSON 中。也就是说，即使有其它的属性存在于该 JSON 响应中，但是只要指定的片段存在，此测试仍然会通过。

验证完全匹配的 JSON

如果你想验证传入的数组是否与应用程序返回的 JSON 完全 匹配，你可以使用 `assertExactJson` 方法：

```

<?php

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $response = $this->json('POST', '/user', ['name' => 'Sally']);

        $response
            ->assertStatus(200)
            ->assertExactJson([
                'created' => true,
            ]);
    }
}

```

测试文件上传

`Illuminate\Http\UploadedFile` 类提供了一个 `fake` 方法，可用于生成用于测试的模拟文件或图像。这与 `storage facade` 的 `fake` 方法结合使用，极大地简化文件上传的测试。例如，你可以结合这两个功能轻松测试头像上传表单：

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Http\UploadedFile;
use Illuminate\Support\Facades\Storage;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    public function testAvatarUpload()
    {
        Storage::fake('avatars');

        $response = $this->json('POST', '/avatar', [
            'avatar' => UploadedFile::fake()->image('avatar.jpg')
        ]);

        // 断言文件已存储
        Storage::disk('avatars')->assertExists('avatar.jpg');

        // 断言文件不存在
        Storage::disk('avatars')->assertMissing('missing.jpg');
    }
}
```

自定义模拟文件

当使用 `fake` 方法创建文件时，你可以指定图片的宽度、高度和大小，以便更好的测试你的验证规则：

```
UploadedFile::fake()->image('avatar.jpg', $width, $height)->size(100);
```

除了创建图片，你还可以使用 `create` 方法创建任何其他类型的文件：

```
UploadedFile::fake()->create('document.pdf', $sizeInKilobytes);
```

可用的断言方法

Laravel 为你的 [PHPUnit](#) 测试提供了各种各样的自定义断言方法。`json`，`get`，`post`，`put` 和 `delete` 这些测试方法返回的响应都可以使用这些断言方法：

Method	Description
<code>\$response->assertStatus(\$code);</code>	断言该响应具有给定的状态码。
<code>\$response->assertRedirect(\$uri);</code>	断言该响应被重定向至指定的 URI。
<code>\$response->assertHeader(\$headerName, \$value = null);</code>	断言该响应存在指定的标头。
<code>\$response->assertCookie(\$cookieName, \$value = null);</code>	断言该响应包含了指定的 cookie。
<code>\$response->assertPlainCookie(\$cookieName, \$value = null);</code>	断言该响应包含了指定的 cookie (未加密)。
<code>\$response->assertSessionHas(\$key, \$value = null);</code>	断言该 session 包含指定数据。
<code>\$response->assertSessionHasErrors(array \$keys);</code>	断言该 session 包含指定字段的错误信息。
<code>\$response->assertSessionMissing(\$key);</code>	断言该 session 不包含指定键。
<code>\$response->assertJson(array \$data);</code>	断言该响应包含指定 JSON 数据。
<code>\$response->assertJsonFragment(array \$data);</code>	断言该响应包含指定 JSON 片段。
<code>\$response->assertExactJson(array \$data);</code>	断言该响应包含完全匹配的 JSON 数据。
<code>\$response->assertJsonStructure(array \$structure);</code>	断言该响应存在指定 JSON 结构。
<code>\$response->assertViewHas(\$key, \$value = null);</code>	断言该响应视图存在指定数据。

Laravel 测试之 - 浏览器测试 (Laravel Dusk)

- [简介](#)
- [安装](#)
 - [使用其他浏览器](#)
- [开始](#)
 - [创建测试](#)
 - [运行测试](#)
 - [环境处理](#)
 - [创建浏览器](#)
 - [认证](#)
- [与元素交互](#)
 - [点击链接](#)
 - [文本、值和属性](#)
 - [使用表单](#)
 - [附加文件](#)
 - [使用键盘](#)
 - [使用鼠标](#)
 - [元素作用域](#)
 - [等待元素](#)
- [可用的断言](#)
- [页面](#)
 - [创建页面](#)
 - [配置页面](#)
 - [导航至页面](#)
 - [选择器简写](#)
 - [页面方法](#)

简介

Laravel Dusk 提供了富有表现力、简单易用的浏览器自动化以及相应的测试 API。Dusk 只需要使用一个单独的 [ChromeDriver](#)，不再需要在你的机器中安装 JDK 或者 Selenium。不过，依然可以按照你自己的需要安装其他 Selenium 兼容的驱动引擎。

安装

你需要在你的项目中添加 `laravel/dusk` Composer 依赖：

```
composer require laravel/dusk
```

安装了 Dusk 之后，你需要注册 `Laravel\Dusk\DuskServiceProvider` 服务提供者。由于 Dusk 提供了登录其他用户的能力，所以为了限制 Dusk 的使用环境，你应该在你的 `AppServiceProvider` 中使用 `register` 方法来注册：

```
use Laravel\Dusk\DuskServiceProvider;

/**
 * 在这里可以注册任何应用服务。
 *
 * @return void
 */
public function register()
{
    if ($this->app->environment('local', 'testing')) {
        $this->app->register(DuskServiceProvider::class);
    }
}
```

接下来运行 `dusk:install` Artisan 命令：

```
php artisan dusk:install
```

会在你的 `tests` 目录下创建一个 `Browser` 目录，同时包含了一个测试用例模版。然后在你的 `.env` 文件中设置 `APP_URL` 环境变量。这个变量值要与你在浏览器访问你应用的 URL 一致。

使用 `dusk` Artisan 命令来运行你的测试。`dusk` 命令支持接受 `phpunit` 命令的任何参数：

```
php artisan dusk
```

使用其他浏览器

Dusk 默认使用 Google Chrome 和 [ChromeDriver](#) 来运行你的浏览器测试。当然，你也可以运行你的 Selenium 服务器，然后对任何你想要的浏览器运行测试。

打开你的 `tests/DuskTestCase.php` 文件，这个文件是你应用中最基础的 Dusk 测试用例。你可以在这个文件中移除 `startChromeDriver` 方法。这样 Dusk 就不会自动运行 ChromeDriver：

```
/**
 * 为 Dusk 的测试做准备。
 *
 * @beforeClass
 * @return void
 */
public static function prepare()
{
    // static::startChromeDriver();
}
```

然后，你可以通过简单地修改 `driver` 方法来连接到你指定的 URL 和端口。同时，你要修改传递给 WebDriver 的「desired capabilities」：

```
/**
 * 创建 RemoteWebDriver 实例。
 *
 * @return \Facebook\WebDriver\Remote\RemoteWebDriver
 */
protected function driver()
{
    return RemoteWebDriver::create(
        'http://localhost:4444', DesiredCapabilities::phantomjs()
    );
}
```

开始

创建测试

使用 `dusk:make Artisan` 命令来创建 Dusk 测试。创建好的测试类会放在 `tests/Browser` 目录：

```
php artisan dusk:make LoginTest
```

运行测试

使用 `dusk Artisan` 命令来运行你的浏览器测试：

```
php artisan dusk
```

`dusk` 命令可以接受任何 PHPUnit 能接受的参数。例如，让你可以只在指定 [分组](#) 中运行测试：

```
php artisan dusk --group=foo
```

手动运行 ChromeDriver

Dusk 默认会尝试自动运行 ChromeDriver。如果在你特定的系统中不能正常运行，你可以在运行 `dusk` 命令之前通过手动的方式来运行 ChromeDriver。如果你选择手动运行 ChromeDriver，你需要在你的 `tests/DuskTestCase.php` 文件中注释掉下面这行：

```
/**
 * 为 Dusk 的测试做准备。
 *
 * @beforeClass
 */
```

```
* @return void
*/
public static function prepare()
{
    // static::startChromeDriver();
}
```

另外，如果你是在非 9515 端口运行 ChromeDriver，你需要在 `tests/DuskTestCase.php` 修改 `driver` 方法：

```
/**
 * 创建 RemoteWebDriver 实例。
 *
 * @return \Facebook\WebDriver\Remote\RemoteWebDriver
 */
protected function driver()
{
    return RemoteWebDriver::create(
        'http://localhost:9515', DesiredCapabilities::chrome()
    );
}
```

环境处理

在你项目的根目录创建 `.env.dusk.{environment}` 文件来强制 Dusk 使用自己的的环境文件来运行测试。简单来说，如果你想要以 `local` 环境来运行 `dusk` 命令，你需要创建一个 `.env.dusk.local` 文件。

运行测试的时候，Dusk 会备份你的 `.env` 文件，然后重命名你的 Dusk 环境文件为 `.env`。一旦测试结束之后，将会恢复你的 `.env` 文件。

创建浏览器

让我们来写一个测试用例，这个测试用例可以验证我们是否能够登录系统。生成测试类之后，我们修改这个类，让它可以跳转到登录页面，输入某些登录信息，点击「登录」按钮。使用 `browse` 方法来创建一个浏览器实例：

```
<?php

namespace Tests\Browser;

use App\User;
use Tests\DuskTestCase;
use Laravel\Dusk\Chrome;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class ExampleTest extends DuskTestCase
```

```

{
    use DatabaseMigrations;

    /**
     * 一个基本的浏览器测试示例。
     *
     * @return void
     */
    public function testBasicExample()
    {
        $user = factory(User::class)->create([
            'email' => 'taylor@laravel.com',
        ]);

        $this->browse(function ($browser) use ($user) {
            $browser->visit('/login')
                ->type('email', $user->email)
                ->type('password', 'secret')
                ->press('Login')
                ->assertPathIs('/home');
        });
    }
}

```

在上面的示例中，你可以看到 `browse` 方法接受一个回调参数。Dusk 会自动将这个浏览器实例注入到回调当中，而这个浏览器实例可以让你与你的应用之间进行交互和断言。

{tip} 这个测试用例可以测试由 `make:auth` Artisan 命令来生成的登录页面。

创建多个浏览器

有时你可能需要多个浏览器才能正确地进行测试。例如，多个浏览器可能用于测试通过 websockets 通讯的在线聊天页面。要想创建多个浏览器，你只需要简单地在 `browse` 方法的回调中，用名字来区分浏览器实例，然后传给回调来「申请」多个浏览器实例即可：

```

$this->browse(function ($first, $second) {
    $first->loginAs(User::find(1))
        ->visit('/home')
        ->waitForText('Message');

    $second->loginAs(User::find(2))
        ->visit('/home')
        ->waitForText('Message')
        ->type('message', 'Hey Taylor')
        ->press('Send');

    $first->waitForText('Hey Taylor')
        ->assertSee('Jeffrey Way');
});

```

认证

你可能经常会测试一些需要认证的页面。你可以使用 Dusk 的 `loginAs` 方法来避免每个测试都去登录页面登录一次。`loginAs` 方法可以使用 用户 ID 或者用户模型实例：

```
$this->browse(function ($first, $second) {
    $first->loginAs(User::find(1))
        ->visit('/home');
});
```

与元素交互

点击链接

你可以在你的浏览器实例中使用 `clickLink` 方法来模拟点击一个链接。`clickLink` 方法会点击传入的显示文本：

```
$browser->clickLink($linkText);
```

{note} 这方法基于 JQuery 来进行交互。如果页面中没有可用的 jQuery，Dusk 会自动将 jQuery 注入到页面中。所以他可能会增加测试的时间。

文本、值和属性

获取和设置值

Dusk 提供了几种方法让你和当前页面元素中的显示文本、值和属性进行交互。举例来说，想获得某个指定选择器对应元素的「值」，你可以使用 `value` 方法：

```
// 获取值...
$value = $browser->value('selector');

// 设置值...
$browser->value('selector', 'value');
```

获取文本

`text` 方法用来获取匹配指定选择器的元素的显示文本：

```
$text = $browser->text('selector');
```

获取属性

`attribute` 方法用来获取匹配指定选择器的元素的属性：

```
$attribute = $browser->attribute('selector', 'value');
```

使用表单

输入值

Dusk 提供了与表单和 `input` 元素交互的各种方法。首先，让我们来看看一个在 `input` 框中输入文本的示例：

```
$browser->type('email', 'taylor@laravel.com');
```

注意：虽然 `type` 方法可以传递 CSS 选择器作为第一个参数，但这并不是强制要求。如果传入的不是 CSS 选择器，Dusk 会尝试匹配传入值与 `name` 属性相符的 `input` 框，如果没找到，最后 Dusk 会尝试查找匹配传入值与 `name` 属性相符的 `textarea`。

你可以使用 `clear` 方法来「清除」输入值。

```
$browser->clear('email');
```

下拉菜单

你可以使用 `select` 方法来选择下来菜单中的某个选项。类似于 `type` 方法，`select` 方法并不是一定要传入 CSS 选择器。当你使用 `select` 方法的时候应该注意，你传的值应该是低层选项的值，而不是显示的值：

```
$browser->select('size', 'Large');
```

你也可以通过省略第二个参数来随机选择一个选项：

```
$browser->select('size');
```

复选框

你可以使用 `check` 方法来选中某个复选框。像其他 `input` 相关的方法一样，并不是必须传入 CSS 选择器。如果准确的选择器没法找到的时候，Dusk 会搜索与 `name` 属性匹配的复选框：

```
$browser->check('terms');

$browser->uncheck('terms');
```

单选按钮

你可以使用 `radio` 方法来选择某个单选选项。同样的，并不是必须传入 CSS 选择器。如果准确的选择器没法找到的时候，Dusk 会搜索与 `name` 属性或者 `value` 属性匹配的单选按钮：

```
$browser->radio('version', 'php7');
```

附加文件

`attach` 方法可以用来附加一个文件到 `file input` 框中。同样的，并不是必须传入 CSS 选择器。如果准确的选择器没法找到的时候，Dusk 会搜索与 `name` 属性匹配的文件输入框：

```
$browser->attach('photo', __DIR__.'/photos/me.png');
```

使用键盘

`keys` 方法让你可以在指定元素中输入比 `type` 方法更加复杂的输入序列。举个例子，你可以在输入值的同时按下按键。在本例中，在输入 `taylor` 的同时，`shift` 按键也同时被按下，当 `taylor` 输入完之后，`otwell` 则会正常输入，不会按下任何按键：

```
$browser->keys('selector', ['{shift}', 'taylor'], 'otwell');
```

甚至你可以在你应用中选中某个元素之后按下「快捷键」：

```
$browser->keys('.app', ['{command}', 'j']);
```

{tip} 所有包在 `{}` 中的修饰按键，都应该与 `Facebook\WebDriver\WebDriverKeys` 类中定义的常量一致。你可以在 [GitHub](#) 中找到这个类。

使用鼠标

点击元素

`click` 方法用来「点击」与指定选择器匹配的元素：

```
$browser->click('.selector');
```

鼠标悬停

`mouseover` 方法用来将鼠标悬停在与指定选择器匹配的元素：

```
$browser->mouseover('.selector');
```

拖拽

`drag` 方法用来拖拽与指定选择器匹配的元素到另外一个元素那里：

```
$browser->drag('.from-selector', '.to-selector');
```

元素作用域

有时候你可能希望 只 在某个与选择器匹配的元素中执行一系列的操作。例如，你可能希望在某个 `table` 中断言某些文本，然后在同一个 `table` 中点击按钮。你可以使用 `with` 方法来达到这个目的。`with` 方法的回调参数中，所有的操作都作用在同一个原始元素上：

```
$browser->with('.table', function ($table) {
    $table->assertSee('Hello World')
        ->clickLink('Delete');
});
```

等待元素

在测试应用的时候，由于经常会用到 JavaScript。所以经常需要在开始之前「等待」某些元素或者数据，以确保在测试中是有效可用的。Dusk 让这变得简单。使用一系列的方法，让你可以等待页面元素完全显示，甚至是给定的 JavaScript 表达式返回 `true` 的时候才继续执行测试：

等待

如果你需要暂停指定毫秒数，你可以使用 `pause` 方法：

```
$browser->pause(1000);
```

等待选择器元素

`waitFor` 方法用来暂停测试的执行，直到与 CSS 选择器匹配的元素显示在页面中。在抛出异常之前，默认最多暂停 5 秒。如果需要，你也可以自定义超时时间作为第二个参数传给这个方法：

```
// 最多等待这个元素 5 秒...
$browser->waitFor('.selector');

// 最多等待这个元素 1 秒...
$browser->waitFor('.selector', 1);
```

你也可以等待指定元素直到超时都还在页面中找不到：

```
$browser->waitUntilMissing('.selector');

$browser->waitUntilMissing('.selector', 1);
```

可用元素的作用域

有时候，你可能想要等待与给定选择器匹配的元素，然后与这元素进行交互。举个例子，你可能等待某个模态窗口可用，然后在模态窗口中点击「OK」按钮。在这种情况下，可以使用 `whenAvailable` 方法。所有闭包中的操作都针对这个原始的元素：

```
$browser->whenAvailable('.modal', function ($modal) {
    $modal->assertSee('Hello World')
        ->press('OK');
});
```

等待文本

`waitForText` 方法用于等待指定文本，直到显示在页面中为止：

```
// 最多等待这个文本显示 5 秒...
$browser->waitForText('Hello World');

// 最多等待这个文本显示 1 秒...
$browser->waitForText('Hello World', 1);
```

等待超链接

`waitForLink` 方法用来等待指定链接文本，直到链接文本显示在页面中为止：

```
// 最多等待这个链接 5 秒...
$browser->waitForLink('Create');

// 最多等待这个链接 1 秒...
$browser->waitForLink('Create', 1);
```

等待 JavaScript 表达式

有时候你可能想要暂停测试用例的执行，直到指定的 JavaScript 表达式计算结果为 `true`。使用 `waitUntil` 方法可以让你很容易做到这一点。传递表达式给方法的时候，你不需要包括 `return` 关键词或者结束分号：

```
// 最多花 5 秒等待表达式成立...
$browser->waitUntil('App.dataLoaded');

$browser->waitUntil('App.data.servers.length > 0');

// 最多花 1 秒等待表达式成立...
$browser->waitUntil('App.data.servers.length > 0', 1);
```

可用的断言

Dusk 为你的应用提供了一系列的断言方法。所有的断言方法都记录在下面的表格中：

Assertion	Description
<code>\$browser->assertTitle(\$title)</code>	断言页面标题符合指定文本。
<code>\$browser->assertTitleContains(\$title)</code>	断言页面标题包含指定文本。
<code>\$browser->assertPathIs('/home')</code>	断言当前路径符合指定路径
<code>\$browser->assertHasCookie(\$name)</code>	断言存在指定 Cookie。
<code>\$browser->assertCookieValue(\$name, \$value)</code>	断言指定 Cookie 为指定值。
<code>\$browser->assertPlainCookieValue(\$name, \$value)</code>	断言一个未加密的 Cookie 为指定值。
<code>\$browser->assertSee(\$text)</code>	断言页面中存在指定文本。
<code>\$browser->assertDontSee(\$text)</code>	断言页面中不存在指定文本。
<code>\$browser->assertSeeIn(\$selector, \$text)</code>	断言选择器中存在指定文本。
<code>\$browser->assertDontSeeIn(\$selector, \$text)</code>	断言选择器中不存在指定文本。
<code>\$browser->assertSeeLink(\$linkText)</code>	断言页面中存在指定链接。
<code>\$browser->assertDontSeeLink(\$linkText)</code>	断言页面中不存在指定链接。
<code>\$browser->assertInputValue(\$field, \$value)</code>	断言指定输入框为指定值。
<code>\$browser->assertInputValueIsNot(\$field, \$value)</code>	断言指定输入框不为指定值。
<code>\$browser->assertChecked(\$field)</code>	断言指定复选框被选中。
<code>\$browser->assertNotChecked(\$field)</code>	断言指定复选框没有被选中。
<code>\$browser->assertRadioSelected(\$field, \$value)</code>	断言指定单选按钮被选中。
<code>\$browser->assertRadioNotSelected(\$field, \$value)</code>	断言指定单选按钮没有被选中。
<code>\$browser->assertSelected(\$field, \$value)</code>	断言指定下拉菜单选中了指定选项。
<code>\$browser->assertNotSelected(\$field, \$value)</code>	断言指定下拉菜单没有选中了指定选项。
<code>\$browser->assertValue(\$selector, \$value)</code>	断言匹配指定选择器的元素为指定值。
<code>\$browser->assertVisible(\$selector)</code>	断言匹配指定选择器的元素是可见的。
<code>\$browser->assertMissing(\$selector)</code>	断言匹配指定选择器的元素是不可见的。

页面

有时候，测试有一些复杂的动作需要顺序执行。这很容易让你的测试代码变得难读，并且难以理解。页面允许你定义语义化的动作行为，然后你可以在给定页面中使用单个方法。页面也允许你为你的应用或者单个页面定义简写的公共选择器。

创建页面

使用 `dusk:page` Artisan 命令来创建页面对象。所有的页面对象会存放在 `tests/Browser/Pages` 目录中：

```
php artisan dusk:page Login
```

配置页面

页面默认拥有 3 个方法：`url`，`assert` 和 `elements`。在这里我们先详述 `url` 和 `assert` 方法。`elements` 方法将会 [在下面详细描述](#)。

`url` 方法

`url` 方法应该返回表示页面 URL 的路径。Dusk 将会在浏览器中使用这个 URL 来导航到具体页面：

```
/**
 * 获得当前页面 URL
 *
 * @return string
 */
public function url()
{
    return '/login';
}
```

`assert` 方法

`assert` 方法可以作出任何断言来验证浏览器是否在给定页面上。这个方法并不是必须的。你可以根据你自己的需求来做出这些断言。这些断言会在你浏览到这个页面的时候自动执行：

```
/**
 * 断言浏览器是否正在给定页面。
 *
 * @return void
 */
public function assert(Browser $browser)
{
    $browser->assertPathIs($this->url());
}
```

导航至页面

一旦页面配置好之后，你可以使用 `visit` 方法导航至页面：

```
use Tests\Browser\Pages\Login;
```

```
$browser->visit(new Login);
```

有时候，你可能已经在指定页面了，你需要的只是「加载」当前页面的选择器和方法到当前测试中来。常见的例子有：当你按下一个按钮的时候，你会被重定向至指定页面，而不是直接导航至指定页面。在这种情况下，你需要使用 `on` 方法来加载页面：

```
use Tests\Browser\Pages\CreatePlaylist;

$browser->visit('/dashboard')
    ->clickLink('Create Playlist')
    ->on(new CreatePlaylist)
    ->assertSee('@create');
```

选择器简写

`elements` 方法允许你为页面中的任何 CSS 选择器定义简单易记的简写。举个例子，让我们为应用登录页中的 `email` 输入框定义一个简写：

```
/**
 * 获取页面的元素简写。
 *
 * @return array
 */
public function elements()
{
    return [
        '@email' => 'input[name=email]',
    ];
}
```

现在你可以用这个简写来代替之前页面中使用的完整 CSS 选择器：

```
$browser->type('@email', 'taylor@laravel.com');
```

全局的选择器简写

安装 Dusk 之后，`Page` 基类存放在你的 `tests/Browser/Pages` 目录。这个类中包含一个 `siteElements` 方法，这个方法可以用来定义全局的选择器简写，这样在你应用中每个页面都可以使用这些全局选择器简写了：

```
/**
 * 获取站点全局的选择器简写。.
 *
 * @return array
 */
```

```
public static function siteElements()
{
    return [
        '@element' => '#selector',
    ];
}
```

页面方法

处理页面中已经定义的默认方法之外，你还可以定义在整个测试过程中会使用到的其他方法。举个例子，让我们假设一下我们正在开发一个音乐管理应用。在应用中都可能用到一个公共的方法来创建列表。而不是在每一页，每一个测试类中都重写一遍创建播放列表的逻辑，这时候你可以在你的页面类中定义一个 `createPlaylist` 方法：

```
<?php

namespace Tests\Browser\Pages;

use Laravel\Dusk\Browser;

class Dashboard extends Page
{
    // 其他页面方法...

    /**
     * 创建一个新的播放列表。
     *
     * @param \Laravel\Dusk\Browser $browser
     * @param string $name
     * @return void
     */
    public function createPlaylist(Browser $browser, $name)
    {
        $browser->type('name', $name)
            ->check('share')
            ->press('Create Playlist');
    }
}
```

一旦方法被定义之后，你可以在任何使用到该页的测试中使用这个方法了。浏览器实例会自动传递给页面方法：

```
use Tests\Browser\Pages\Dashboard;

$browser->visit(new Dashboard)
    ->createPlaylist('My Playlist')
    ->assertSee('My Playlist');
```


Laravel 测试之 - 数据库测试

- 简介
- 每次测试后重置数据库
 - 使用迁移
 - 使用事务
- 创建模型工厂
 - 多种模型工厂状态
- 在测试中使用模型工厂
 - 创建模型
 - 持久化模型
 - 模型关联

简介

Laravel 提供了多种有用的工具来让你更容易的测试使用数据库的应用程序。首先，你可以使用

`assertDatabaseHas` 辅助函数，来断言数据库中是否存在与指定条件互相匹配的数据。举例来说，如果我们想验证 `users` 数据表中是否存在 `email` 值为 `sally@example.com` 的数据，我们可以按照以下的方式来做测试：

```
public function testDatabase()
{
    // 创建调用至应用程序...

    $this->assertDatabaseHas('users', [
        'email' => 'sally@example.com'
    ]);
}
```

当然，使用 `assertDatabaseHas` 方法及其它的辅助函数只是为了方便。你也可以随意使用 PHPUnit 内置的所有断言方法来扩充测试。

每次测试后重置数据库

在每次测试结束后都需要对数据进行重置，这样前面的测试数据就不会干扰到后面的测试。

使用迁移

其中有一种方式就是在每次测试后都还原数据库，并在下次测试前运行迁移。Laravel 提供了简洁的 `DatabaseMigrations` trait，它会自动帮你处理好这些操作。你只需在测试类中使用此 trait 即可：

```
<?php
```

```

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    use DatabaseMigrations;

    /**
     * 基本的功能测试示例。
     *
     * @return void
     */
    public function testBasicExample()
    {
        $response = $this->get('/');

        // ...
    }
}

```

使用事务

另一个方式，就是将每个测试案例都包含在数据库事务中。Laravel 提供了一个简洁的 `DatabaseTransactions` trait 来自动帮你处理好这些操作。

```

<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    use DatabaseTransactions;

    /**
     * 基本的功能测试示例。
     *
     * @return void
     */
    public function testBasicExample()
    {

```

```

    $response = $this->get('/');
    // ...
}
}

```

{note} 此 trait 的事务只包含默认的数据库连接。如果你的应用程序使用多个数据库连接，你需要在你的测试类中定义一个 `$connectionsToTransact` 属性，然后你就可以把你测试中需要用到的数据库连接名称以数组的形式放到这个属性中。

创建模型工厂

测试时，常常需要在运行测试之前写入一些数据到数据库中。创建测试数据时，除了手动的来设置每个字段的值，还可以使用 [Eloquent 模型](#) 的「工厂」来设置每个属性的默认值。在开始之前，你可以先查看下应用程序的 `database/factories/ModelFactory.php` 文件。此文件包含一个现成的模型工厂定义：

```

$factory->define(App\User::class, function (Faker\Generator $faker) {
    static $password;

    return [
        'name' => $faker->name,
        'email' => $faker->unique()->safeEmail,
        'password' => $password ?: $password = bcrypt('secret'),
        'remember_token' => str_random(10),
    ];
});

```

闭包内为模型工厂的定义，你可以返回模型中所有属性的默认测试值。在该闭包内会接收到 [Faker](#) PHP 函数库的实例，它可以让你很方便的生成各种随机数据以进行测试。

当然，你也可以随意将自己其他的模型工厂增加至 `ModelFactory.php` 文件。你也可以在 `database/factories` 里为每一个数据模型创建对应的模型工厂类，如 `UserFactory.php` 和 `CommentFactory.php`。在 `factories` 目录中的文件都会被 Laravel 自动加载。

多种模型工厂状态

模型工厂状态可以让你任意组合你的模型工厂，仅需要做出适当差异化的修改，就可以达到让模型拥有多种不同的状态。例如，你的 `User` 模型中可以修改某个默认属性值来达到标识一种 `拖欠债务` 的状态。你可以使用 `state` 方法来进行这种状态转换：

```

$factory->state(App\User::class, 'delinquent', function ($faker) {
    return [
        'account_status' => 'delinquent',
    ];
});

```

在测试中使用模型工厂

创建模型

在模型工厂定义后，就可以在测试或是数据库的填充文件中，通过全局的 `factory` 函数来生成模型实例。接着让我们先来看看几个创建模型的例子。首先我们会使用 `make` 方法创建模型，但不将它们保存至数据库：

```
public function testDatabase()
{
    $user = factory(App\User::class)->make();

    // 在测试中使用模型...
}
```

你也可以创建一个含有多个模型的集合，或创建一个指定类型的模型：

```
// 创建一个 App\User 实例
$users = factory(App\User::class, 3)->make();
```

应用模型工厂状态

你可能需要在你的模型中应用不同的 [模型工厂状态](#)。如果你想模型加上多种不同的状态，你只须指定每个你想添加的状态名称即可：

```
$users = factory(App\User::class, 5)->states('delinquent')->make();

$users = factory(App\User::class, 5)->states('premium', 'delinquent')->make();
```

重写模型属性

如果你想重写模型中的某些默认值，则可以传递一个包含数值的数组至 `make` 方法。只有指定的数值会被替换，其它剩余的数值则会按照模型工厂指定的默认值来设置：

```
$user = factory(App\User::class)->make([
    'name' => 'Abigail',
]);
```

持久化模型

`create` 方法不仅会创建模型实例，同时会使用 Eloquent 的 `save` 方法来将它们保存至数据库：

```
public function testDatabase()
{
```

```
// 创建一个 App\User 实例
$user = factory(App\User::class)->create();

// 创建 3 个 App\User 实例
$users = factory(App\User::class, 3)->create();

// 在测试中使用模型...
}
```

同样的，你可以在数组传递至 `create` 方法时重写模型的属性

```
$user = factory(App\User::class)->create([
    'name' => 'Abigail',
]);
```

模型关联

在本例中，我们还会增加关联至我们所创建的模型。当使用 `create` 方法创建多个模型时，它会返回一个 Eloquent 集合实例，让你能够使用集合所提供的便利函数，像是 `each`：

```
$users = factory(App\User::class, 3)
->create()
->each(function ($u) {
    $u->posts()->save(factory(App\Post::class)->make());
});
```

关联和属性闭包

你可以使用闭包参数来创建模型关联。例如如果你想在创建一个 `Post` 的顺便创建一个 `User` 实例：

```
$factory->define(App\Post::class, function ($faker) {
    return [
        'title' => $faker->title,
        'content' => $faker->paragraph,
        'user_id' => function () {
            return factory(App\User::class)->create()->id;
        }
    ];
});
```

这些闭包也可以获取到生成的模型工厂包含的属性数组：

```
$factory->define(App\Post::class, function ($faker) {
    return [
        'title' => $faker->title,
```

```
'content' => $faker->paragraph,  
'user_id' => function () {  
    return factory(App\User::class)->create()->id;  
},  
'user_type' => function (array $post) {  
    return App\User::find($post['user_id'])->type;  
}  
];  
});
```

Laravel 测试之：测试模拟器

- [介绍](#)
- [任务模拟](#)
- [事件模拟](#)
- [邮件模拟](#)
- [通知模拟](#)
- [队列模拟](#)
- [Storage 模拟](#)
- [Facades 模拟](#)

介绍

测试 Laravel 应用时，有时候你可能想要「模拟」实现应用的部分功能的行为，从而避免该部分在测试过程中真正执行。例如，控制器执行过程中会触发一个事件（Events），你想要模拟这个事件的监听器，从而避免该事件在测试这个控制器时真正执行。如上可以让你仅测试控制器的 HTTP 响应情况，而不用去担心触发事件。当然，你可以在单独的测试中测试该事件的逻辑。

Laravel 针对事件、任务和 facades 的模拟提供了开箱即用的辅助函数。这些辅助函数基于 Mockery 封装而成，使用非常简单，无需你手动调用复杂的 Mockery 函数。当然，你也可以使用 [Mockery](#) 或者 [PHPUnit](#) 创建自己的模拟器。

任务模拟

你可以使用 `Bus facade` 的 `fake` 方法来模拟任务执行，测试的时候任务不会被真实执行。使用 `fakes` 的时候，断言一般出现在测试代码的后面：

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use App\Jobs\ShipOrder;
use Illuminate\Support\Facades\Bus;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
    {
        Bus::fake();

        // 处理订单发货...
    }
}
```

```

        Bus::assertDispatched(ShipOrder::class, function ($job) use ($order) {
            return $job->order->id === $order->id;
        });

        // 断言任务并没有被执行...
        Bus::assertNotDispatched(AnotherJob::class);
    }
}

```

事件模拟

你可以使用 `Event facade` 的 `fake` 方法来模拟事件监听，测试的时候不会触发事件监听器运行。然后你就可以断言事件运行了，甚至可以检查它们收到的数据。使用 `fakes` 的时候，断言一般出现在测试代码的后面：

```

<?php

namespace Tests\Feature;

use Tests\TestCase;
use App\Events\OrderShipped;
use App\Events\OrderFailedToShip;
use Illuminate\Support\Facades\Event;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    /**
     * 测试订单发货.
     */
    public function testOrderShipping()
    {
        Event::fake();

        // 处理订单发货...

        Event::assertDispatched(OrderShipped::class, function ($e) use ($order) {
            return $e->order->id === $order->id;
        });

        Event::assertNotDispatched(OrderFailedToShip::class);
    }
}

```

邮件模拟

你可以使用 `Mail facade` 的 `fake` 方法来模拟邮件发送，测试时不会真的发送邮件。然后你可以断言 `mailables` 发送给了用户，甚至可以检查他们收到的数据。使用 `fakes` 时，断言一般在测试代码的后面：

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use App\Mail\OrderShipped;
use Illuminate\Support\Facades\Mail;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
    {
        Mail::fake();

        // 处理订单发货...

        Mail::assertSent(OrderShipped::class, function ($mail) use ($order) {
            return $mail->order->id === $order->id;
        });

        // 断言一封邮件已经发送给了指定用户...
        Mail::assertSent(OrderShipped::class, function ($mail) use ($user) {
            return $mail->hasTo($user->email) &&
                $mail->hasCc('...') &&
                $mail->hasBcc('...');
        });

        // 断言 mailable 没有发送...
        Mail::assertNotSent(AnotherMailable::class);
    }
}
```

通知模拟

你可以使用 `Notification facade` 的 `fake` 方法来模拟通知发送，测试的时候并不会真的发送通知。然后你可以断言 `通知` 已经发送给你的用户，甚至可以检查他们收到的数据。使用 `fakes` 时，断言一般出现在测试代码的后面。

```
<?php
```

```

namespace Tests\Feature;

use Tests\TestCase;
use App\Notifications\OrderShipped;
use Illuminate\Support\Facades\Notification;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
    {
        Notification::fake();

        // 处理订单发货...

        Notification::assertSentTo(
            $user,
            OrderShipped::class,
            function ($notification, $channels) use ($order) {
                return $notification->order->id === $order->id;
            }
        );

        // 断言通知已经发送给了指定用户...
        Notification::assertSentTo(
            [$user], OrderShipped::class
        );

        // 断言通知没有发送...
        Notification::assertNotSentTo(
            [$user], AnotherNotification::class
        );
    }
}

```

队列模拟

你可以使用 `Queue facade` 的 `fake` 方法来模拟任务队列，测试的时候并不会真的把任务放入队列。然后你可以断言任务被放进了队列，甚至可以检查它们收到的数据。使用 `fakes` 的时候，断言一般出现在测试代码的后面。

```

<?php

namespace Tests\Feature;

use Tests\TestCase;

```

```

use App\Jobs\ShipOrder;
use Illuminate\Support\Facades\Queue;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
    {
        Queue::fake();

        // 处理订单发货...

        Queue::assertPushed(ShipOrder::class, function ($job) use ($order) {
            return $job->order->id === $order->id;
        });

        // 断言任务进入了指定队列
        Queue::assertPushedOn('queue-name', ShipOrder::class);

        // 断言任务没有进入队列
        Queue::assertNotPushed(AnotherJob::class);
    }
}

```

Storage 模拟

利用 `Storage facade` 的 `fake` 方法，你可以轻松地生成一个模拟的磁盘，结合 `UploadedFile` 类的文件生成工具，极大地简化了文件上传测试。例如：

```

<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Http\UploadedFile;
use Illuminate\Support\Facades\Storage;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    public function testAvatarUpload()
    {
        Storage::fake('avatars');

        $response = $this->json('POST', '/avatar', [

```

```

    'avatar' => UploadedFile::fake()->image('avatar.jpg')
]);

// 断言文件已存储
Storage::disk('avatars')->assertExists('avatar.jpg');

// 断言文件不存在
Storage::disk('avatars')->assertMissing('missing.jpg');
}
}
}

```

Facades 模拟

不同于传统的静态函数的调用，`facades` 也是可以被模拟的，相对静态函数来说这是个巨大的优势，即使你在使用依赖注入，测试时依然会非常方便。在很多测试中，你可能经常想在控制器中模拟对 Laravel facade 的调用。比如下面控制器中的行为：

```

<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

class UserController extends Controller
{
    /**
     * 显示网站的所有用户
     *
     * @return Response
     */
    public function index()
    {
        $value = Cache::get('key');

        //
    }
}

```

我们可以通过 `shouldReceive` 方法来模拟 `Cache facade`，此函数会返回一个 `Mockery` 实例，由于对 `facade` 的调用实际上都是由 Laravel 的 [服务容器](#) 管理的，所以 `facade` 能比传统的静态类表现出更好的测试便利性。接下来，让我们来模拟一下 `Cache facade` 的 `get` 方法的调用：

```

<?php

namespace Tests\Feature;

use Tests\TestCase;

```

```
use Illuminate\Support\Facades\Cache;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class UserControllerTest extends TestCase
{
    public function testGetIndex()
    {
        Cache::shouldReceive('get')
            ->once()
            ->with('key')
            ->andReturn('value');

        $response = $this->get('/users');

        // ...
    }
}
```

{note} 不可以模拟 `Request` facade，测试时，如果需要传递指定的数据请使用 HTTP 辅助函数，例如 `get` 和 `post`。类似的，请在你的测试中通过调用 `Config::set` 来模拟 `Config` facade。

12 官方扩展包

Laravel 的收费系统 Cashier

- 简介
- 配置
 - Stripe
 - Braintree
 - 货币配置
- 订阅列表
 - 创建订阅
 - 查询订阅状态
 - 更改订阅计划
 - 按量计费订阅
 - 订阅税额设置
 - 取消订阅
 - 恢复订阅
 - 更新信用卡信息
- 试用订阅
 - 必须提供信用卡信息
 - 非必须提供信用卡
- 处理 Stripe Webhooks
 - 定义 Webhook 事件处理器
 - 订阅失败
- 处理 Braintree Webhooks
 - 定义 Webhook 事件处理器
 - 订阅失败
- 一次性收费
- 发票
 - 生成发票的 PDFs

简介

Laravel Cashier 提供了直观、流畅的接口来接入 Stripe's 和 Braintree's 订阅付费服务。它可以处理几乎所有你写起来非常头疼付费订阅代码。除了提供基本的订阅管理之外，Cashier 还可以帮你处理优惠券，更改订阅计划，按量计费订阅，已取消订阅宽限期管理，甚至生成发票的 PDF 文件。

{note} 如果你只是需要提供一次性的收费服务，建议直接使用 Stripe 和 Braintree 的 SDK，而无需使用 Cashier。

配置

Stripe

Composer

首先，将 Cashier Stripe 扩展包添加到 `composer.json` 文件，然后运行 `composer update` 命令：

```
"laravel/cashier": "~7.0"
```

Service 服务提供者

接下来，需要注册 `Laravel\Cashier\CashierServiceProvider` 服务提供者 到你的 `config/app.php` 配置文件。

数据库迁移

在使用 Cashier 之前，我们需要 [准备一下数据库](#)。需要在 `users` 表中新增几列，以及创建一个新的 `subscriptions` 表来存储客户的订阅信息：

```
Schema::table('users', function ($table) {
    $table->string('stripe_id')->nullable();
    $table->string('card_brand')->nullable();
    $table->string('card_last_four')->nullable();
    $table->timestamp('trial_ends_at')->nullable();
});

Schema::create('subscriptions', function ($table) {
    $table->increments('id');
    $table->integer('user_id');
    $table->string('name');
    $table->string('stripe_id');
    $table->string('stripe_plan');
    $table->integer('quantity');
    $table->timestamp('trial_ends_at')->nullable();
    $table->timestamp('ends_at')->nullable();
    $table->timestamps();
});
```

创建好数据库迁移文件之后，需要运行一下 `migrate` Artisan 命令。

Billable 模型

现在，需要添加 `Billable trait` 到你的模型定义。`Billable trait` 提供了丰富的方法去允许你完成常见的支付任务，比如创建订阅，使用优惠券以及更新信用卡信息。

```
use Laravel\Cashier\Billable;

class User extends Authenticatable
{
    use Billable;
}
```

API Keys

最后，你需要在 `services.php` 配置文件中设置你的 Stripe key。你可以在 Stripe 的控制面板获取到相关的 API keys。

```
'stripe' => [
    'model'  => App\User::class,
    'key'    => env('STRIPE_KEY'),
    'secret' => env('STRIPE_SECRET'),
],
```

Braintree

Braintree 注意事项

对于大多数操作，Stripe 和 Braintree 在 Cashier 的实现方法是一致的，都提供对信用卡订阅计费的支持。但是 Braintree 是通过 Paypal 进行支付的，所以相对 Stripe 来说会缺少一些功能的支持。在选择该使用 Stripe 还是 Braintree 的时候需要留意这些区别。

- Braintree 支持 PayPal，但是 Stripe 不支持。
- Braintree 并不支持对 subscriptions 使用 `increment` 和 `decrement` 方法。这是 Braintree 本身的限制，并不是 Cahier 的限制。
- Braintree 并不支持基于百分比的折扣。这也是 Braintree 本身的限制，并不是 Cahier 的限制。

Composer

首先，将 Cashier Braintree 扩展包添加到 `composer.json` 文件，然后运行 `composer update` 命令：

```
"laravel/cashier-braintree": "~2.0"
```

服务提供者

接下来，需要注册 `Laravel\Cashier\CashierServiceProvider` 服务提供者 到你的 `config/app.php` 配置文件。

信用卡优惠计划

在使用 Cashier 之前，你需要首先在 Braintree 控制面板定义一个 `plan-credit` 折扣。这个折扣会根据用户选择的支付选项匹配合适的折扣比例，比如选择年付还是月付。

在 Braintree 控制面板中配置的折扣总额可以随意填，Cashier 会在每次使用优惠券的时候根据我们自己的定制覆盖该默认值。我们需要这个优惠券计划的原因是 Braintree 并不原生支持按照订阅频率来匹配折扣比例。

数据库迁移

在使用 Cashier 之前，我们需要 [准备一下数据库](#)。需要在 `users` 表中新增几列，以及创建一个新的 `subscriptions` 表来存储客户的订阅信息：

```
Schema::table('users', function ($table) {
    $table->string('braintree_id')->nullable();
    $table->string('paypal_email')->nullable();
    $table->string('card_brand')->nullable();
    $table->string('card_last_four')->nullable();
    $table->timestamp('trial_ends_at')->nullable();
});

Schema::create('subscriptions', function ($table) {
    $table->increments('id');
    $table->integer('user_id');
    $table->string('name');
    $table->string('braintree_id');
    $table->string('braintree_plan');
    $table->integer('quantity');
    $table->timestamp('trial_ends_at')->nullable();
    $table->timestamp('ends_at')->nullable();
    $table->timestamps();
});
```

创建好数据库迁移文件之后，需要运行一下 `migrate` Artisan 命令。

Billable 模型

接下来，添加 `Billable` trait 到你的模型定义。

```
use Laravel\Cashier\Billable;

class User extends Authenticatable
{
    use Billable;
}
```

API Keys

现在，你需要参考下面的示例在 `services.php` 文件中配置相关选项：

```
'braintree' => [
    'model'  => App\User::class,
    'environment' => env('BRAINTREE_ENV'),
    'merchant_id' => env('BRAINTREE_MERCHANT_ID'),
    'public_key' => env('BRAINTREE_PUBLIC_KEY'),
    'private_key' => env('BRAINTREE_PRIVATE_KEY'),
],
```

然后你需要将 Braintree SDK 添加到你的 `AppServiceProvider` 的 `boot` 方法中:

```
\Braintree_Configuration::environment(config('services.braintree.environment'));
\Braintree_Configuration::merchantId(config('services.braintree.merchant_id'));
\Braintree_Configuration::publicKey(config('services.braintree.public_key'));
\Braintree_Configuration::privateKey(config('services.braintree.private_key'));
```

货币配置

Cashier 使用美元 (USD) 作为默认货币。你可以在某个服务提供者的 `boot` 方法中使用 `Cashier::useCurrency` 方法来修改默认的货币设置。`useCurrency` 方法接受两个字符串参数: 货币名称和货币符号。

```
use Laravel\Cashier\Cashier;

Cashier::useCurrency('eur', '€');
```

订阅列表

创建订阅

创建订阅, 首先需要获取到一个 `billabel` 模型实例, 一般情况下就是 `App\User` 实例。然后你可以使用 `newSubscription` 方法来创建该模型的订阅:

```
$user = User::find(1);

$user->newSubscription('main', 'monthly')->create($stripeToken);
```

`newSubscription` 方法的第一个参数应该是订阅的名称。如果你的应用程序只是提供一个简单的订阅, 你可以简单的设置为 `main` 或者 `primary`。第二个参数需要指定用户的 Stripe / Braintree 订阅计划。这里的值应该和在 Stripe 或 Braintree 的中的对应值保持一致。

`create` 方法会创建订阅并且更新客户 ID 和相关的支付信息到数据库。

Additional User Details

如果你需要添加额外的客户细节信息, 你可以在 `create` 方法的第二个参数中进行传递:

```
$user->newSubscription('main', 'monthly')->create($stripeToken, [
    'email' => $email,
]);
```

查阅一下 Stripe 的 [创建客户文档](#) 或对应的 [Braintree 文档](#) 了解更多支持的客户细节信息的内容。

优惠券

如果你想在创建订阅的时候使用优惠券，你可以使用 `withCoupon` 方法：

```
$user->newSubscription('main', 'monthly')
    ->withCoupon('code')
    ->create($stripeToken);
```

查询订阅状态

一旦用户在你的应用程序中完成了订阅，你可以使用大量便利的方法来查询他们的订阅状态。首先，`subscribed` 方法会返回 `true` 如果用户已经激活了订阅，即使该订阅正在试用期内。

```
if ($user->subscribed('main')) {
    //
}
```

也可以在 [路由中间件](#) 中使用 `subscribed` 方法，来基于用户的订阅状态过滤用户请求：

```
public function handle($request, Closure $next)
{
    if ($request->user() && ! $request->user()->subscribed('main')) {
        // 用户并没有完成支付...
        return redirect('billing');
    }

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

如果你想知道用户是否在一个常识周期内，你可以使用 `onTrial` 方法。该方法可以用来提示用户他们还在试用期之内：

```
if ($user->subscription('main')->onTrial()) {
    //
}
```

`subscribedToPlan` 方法可以基于给定的 Stripe / Braintree 计划 ID 来判断用户是否有订阅该计划。下面的示例，我们在判断用户的 `main` 订阅是否成功激活订阅了 `monthly` 计划。

```
if ($user->subscribedToPlan('monthly', 'main')) {
    //
}
```

取消订阅的状态

可以使用 `cancelled` 方法来判断用户是否之前已经完成了订阅，但是又已经取消了他们的订阅：

```
if ($user->subscription('main')->cancelled()) {
    //
}
```

可以去判断用户是否已经取消了他们的订阅，但是还是在订阅的「宽限期」直到订阅完全到期。比如说，如果一个用户在 3 月 5 号取消了订阅，但是原本订阅定于 3 月 10 号到期，那么该用户就会处于「宽限期」直到 3 月 10 号。需要注意的是，`subscribed` 方法在宽限期还是会返回 `true`：

```
if ($user->subscription('main')->onGracePeriod()) {
    //
}
```

更改订阅计划

在用户完成订阅之后，有时会更改订阅计划。将用户切换到一个新的订阅计划，需要传递订阅计划的 ID 给 `swap` 方法：

```
$user = App\User::find(1);

$user->subscription('main')->swap('provider-plan-id');
```

如果用户在试用期，试用期的期限会被保留。如果订阅有限量的「份额」存在，该份额数目也会被保留。

如果你想在更改用户订阅计划的时候取消用户当前订阅的试用期，可以使用 `skipTrial` 方法：

```
$user->subscription('main')
    ->skipTrial()
    ->swap('provider-plan-id');
```

订阅份额

{note} 按量计费订阅方式只支持 Stripe，因为 Braintree 并没有按量计费订阅之类的功能。

有些时候订阅是会受「数量」影响的。举个例子，你的应用程序的付费方式可能是每个账户 \$10 /人/月。你可以使用 `incrementQuantity` 和 `decrementQuantity` 方法轻松的增加或者减少你的订阅数量。

```
$user = User::find(1);

$user->subscription('main')->incrementQuantity();

// 当前的订阅数量加 5
```

```
$user->subscription('main')->incrementQuantity(5);

$user->subscription('main')->decrementQuantity();

// 当前的订阅数量减 5
$user->subscription('main')->decrementQuantity(5);
```

另外，你可以使用 `updateQuantity` 方法指定订阅的数量：

```
$user->subscription('main')->updateQuantity(10);
```

查阅 [Stripe 文档](#)，了解更多关于按量订阅的信息。

订阅税额设置

为了设置每笔订阅的税收比例，需要在 `billable` 模型内实现 `taxPercentage` 方法，该方法返回一个 0 到 100 的数字，精度不超过小数点后两位。

```
public function taxPercentage() {
    return 20;
}
```

`taxPercentage` 方法可以让你在模型基础上应用税率，这对于用户群跨越多个国家和税收的情况非常有帮助。

{note} 但是需要注意的是 `taxPercentage` 方法仅适用于订阅付费模式。如果你使用 Cashier 完成一次性收费的业务，你需要手动的指定税率。

取消订阅

取消订阅，可以直接在用户的订阅上调用 `cancel` 方法：

```
$user->subscription('main')->cancel();
```

当订阅被取消之后，Cashier 自动会填充数据库中的 `ends_at` 列。这列用来指定 `subscribed` 方法什么时候应该返回 `false`。比如，如果一个客户在 3 月 1 号取消了订阅，但是订阅并不会结束直到 3 月 5 号才到期，所以 `subscribed` 方法会继续返回 `true` 直到 3 月 5 号。

你也可以使用 `onGracePeriod` 方法来判断用户是否已经取消了订阅，但是还在一个「宽限期」：

```
if ($user->subscription('main')->onGracePeriod()) {
    //
}
```

如果你想直接取消一个订阅，可以直接调用 `cancelNow` 方法：

```
$user->subscription('main')->cancelNow();
```

恢复订阅

如果一个用户取消订阅之后，你可以使用 `resume` 方法来恢复他的订阅。该方法只适用于用户取消订阅之后的宽限期：

```
$user->subscription('main')->resume();
```

如果用户取消订阅之后，然后恢复订阅之时订阅已经到期，他们并不会被立即支付费用。代替的是，订阅会被简单进入重新激活的状态，需要按照原本的支付流程再次进行支付。

更新信用卡信息

`updateCard` 方法被用于更新客户的信用卡信息。该方法会接受一个 Stripe 的 token，并设置一个新的信用卡作为默认的结算来源：

```
$user->updateCard($stripeToken);
```

试用订阅

必须提供信用卡信息

如果你想用户在选择试用订阅计划的时候必须提供信用卡信息，可以使用在创建订阅时使用 `trialDays` 方法：

```
$user = User::find(1);

$user->newSubscription('main', 'monthly')
    ->trialDays(10)
    ->create($stripeToken);
```

该方法会设置一个试用期结束日期在数据库的订阅记录上，同时告知 Stripe / Braintree 在该日期之前并不开始结算。

{note} 如果客户没有在试用期结束之前取消订阅，订阅会被自动结算，所以需要在试用期结束之前通知你的客户。

你可以使用用户实例的 `onTrial` 方法来判断用户是否在试用期内，或者使用订阅实例上的 `onTrial` 方法也可以。下面是示例：

```
if ($user->onTrial('main')) {
    //
}
```

```
if ($user->subscription('main')->onTrial()) {
    //
}
```

非必须提供信用卡

如果你打算提供一个试用期，并且不需要用户立马提交信用卡信息，你可以简单的在数据库的用户表记录中设置 `trial_ends_at` 列为你需要的试用期结束日期。这是用户注册过程中的典型手法：

```
$user = User::create([
    // 填充其他用户属性...
    'trial_ends_at' => Carbon::now()->addDays(10),
]);
```

{note} 请确保在你的模型中已经为 `trial_ends_at` 添加[日期转换器](#)。

Cashier 把这种类型的试用引用为「generic trial」，因为它并没有关联任何已存在的订阅。`User` 实例上的 `onTrial` 会返回 `true` 值，只要当前的日期没有超过 `trial_ends_at` 的值：

```
if ($user->onTrial()) {
    // 用户在试用期内...
}
```

你可以使用 `onGenericTrial` 方法来判断用户是否在一个「generic」试用期间，其实并没有创建任何真实的订阅：

```
if ($user->onGenericTrial()) {
    // 用户在 「generic」 试用期内...
}
```

一旦你准备创建一个真实的订阅给用户，你通常可以使用 `newSubscription` 方法：

```
$user = User::find(1);

$user->newSubscription('main', 'monthly')->create($stripeToken);
```

处理 Stripe Webhooks

Stripe 和 Braintree 都能通过 webhooks 通知大量的事件给你的应用程序。为了处理 Stripe webhooks，定义一个路由指向 Cashier's webhook 控制器。该控制器会处理所有传来的请求并分发到合适的控制器方法：

```
Route::post(
    'stripe/webhook',
```

```
'\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook'
);
```

{note} 注册好路由之后，记得在 Stripe 的控制面板中配置好 webhook 跳转的 URL。

Cashier 的控制器默认会在多次失败的支付尝试后自动取消该订阅（取决于你的 Stripe 设置）；但是，稍后你会发现，你可以根据自己的需要扩展该控制器。

Webhooks & CSRF 保护

因为 Stripe webhooks 需要绕过 Laravel 的 [CSRF 保护](#)，所以需要确保将相关的 URI 作为例外添加到你的 `VerifyCsrfToken` 中间件或者在 `web` 中间件集合之外定义相关路由。

```
protected $except = [
    'stripe/*',
];
```

定义 Webhook 事件处理器

Cashier 对于失败的支付自动进行取消订阅的处理，但是如果你想对 Stripe webhook 事件进行额外的处理，可以简单的扩展一下 Webhook 控制器。你的方法名称应该与 Cashier 的预设惯例一致，特别是，方法名应该以 `handle` 为前缀，然后以「驼峰」命名的方式加上你要处理的 Stripe webhook 的名称。比如，如果你希望去处理 `invoice.payment_succeeded` webhook，你应该添加一个 `handleInvoicePaymentSucceeded` 方法到控制器：

```
<?php

namespace App\Http\Controllers;

use Laravel\Cashier\Http\Controllers\WebhookController as CashierController;

class WebhookController extends CashierController
{
    /**
     * 处理 Stripe webhook。
     *
     * @param array $payload
     * @return Response
     */
    public function handleInvoicePaymentSucceeded($payload)
    {
        // Handle The Event
    }
}
```

订阅失败

如果客户的信用卡过期了怎么办？不用担心 - Cashier 的 webhook 控制器会轻易的取消客户的订阅。像上面提到的，你所需要做的知识简单的指定一个路由到该控制器：

```
Route::post(
    'stripe/webhook',
    '\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook'
);
```

就这么简单！失败的支付会被控制器捕捉、并处理。Cashier 控制器会自动取消客户的订阅当 Stripe 判断该订阅已经失败（正常会经过三次错误的支付尝试）。

处理 Braintree Webhooks

Stripe 和 Braintree 都能通过 webhooks 通知大量的事件给你的应用程序。为了处理 Braintree webhooks，定义一个路由指向 Cashier's webhook 控制器。该控制器会处理所有传来的请求并分发到合适的控制器方法：

```
Route::post(
    'braintree/webhook',
    '\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook'
);
```

{note} 注册好路由之后，记得在 Braintree 的控制面板中配置好 webhook 跳转的 URL。

Cashier 的控制器默认会在多次失败的支付尝试后自动取消该订阅（取决于你的 Braintree 设置）；但是，稍后你会发现，你可以根据自己的需要扩展该控制器。

Braintree & CSRF 保护

因为 Stripe webhooks 需要绕过 Laravel 的 [CSRF 保护](#)，所以需要确保将相关的 URI 作为例外添加到你的 `VerifyCsrfToken` 中间件或者在 `web` 中间件集合之外定义相关路由。

```
protected $except = [
    'braintree/*',
];
```

定义 Webhook 事件处理器

Cashier 对于失败的支付自动进行取消订阅的处理，但是如果你想对 Braintree webhook 事件进行额外的处理，可以简单的扩展一下 Webhook 控制器。你的方法名称应该与 Cashier 的预设惯例一致，特别的是，方法名应该以 `handle` 为前缀，然后以「驼峰」命名的方式加上你要处理的 Braintree webhook 的名称。比如，如果你希望去处理 `dispute_opened` webhook，你应该添加一个 `handleDisputeOpened` 方法到控制器：

```
<?php
```

```

namespace App\Http\Controllers;

use Braintree\WebhookNotification;
use Laravel\Cashier\Http\Controllers\WebhookController as CashierController;

class WebhookController extends CashierController
{
    /**
     * 处理 Braintree webhook。
     *
     * @param  WebhookNotification  $webhook
     * @return Response
     */
    public function handleDisputeOpened(WebhookNotification $notification)
    {
        // Handle The Event
    }
}

```

订阅失败

如果客户的信用卡过期了怎么办？不用担心 - Cashier 的 webhook 控制器会轻易的取消客户的订阅。像上面提到的，你所需要做的知识简单的指定一个路由到该控制器：

```

Route::post(
    'braintree/webhook',
    '\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook'
);

```

就这么简单！失败的支付会被控制器捕捉、并处理。Cashier 控制器会自动取消客户的订阅当 Braintree 判断该订阅已经失败（正常会经过三次错误的支付尝试）。不要忘了：你需要在 Braintree 的控制面板正确配置 webhook URI。

一次性收费

单次付费

{note} 使用 Stripe 时，`charge` 方法是以你当前货币的最小面值作为单位的。但是，使用 Braintree 时，你应该传递完整的美元金额给 `charge` 方法：

如果你想对一个已订阅客户执行「单次」收费，你可以在 `billable` 模型实例上使用 `charge` 方法。

```

// Stripe 以美分结算...
$user->charge(100);

// Braintree 以美元结算...

```

```
$user->charge(1);
```

`charge` 方法可以接受一个数据作为第二个参数，允许你在 Stripe / Braintree 支付时传递任何可用的参数。查阅 Stripe 或 Braintree 的文档了解都有哪些可用的参数。

```
$user->charge(100, [
    'custom_option' => $value,
]);

```

当支付结算失败是 `charge` 方法会抛出一个异常。如果结算成功，会返回完成的 Stripe / Braintree 响应。

```
try {
    $response = $user->charge(100);
} catch (Exception $e) {
    //
}
```

结算时生成发票

有时候你只是需要完成一个一次性的支付，但是也需要为结算生成一个发票信息，这样你可以提供 PDF 的票据给你的客户。`invoiceFor` 方法就是做这个的。例如，让我们给客户开具一个 \$5.00 的「单次收费」发票：

```
// Stripe 以美分结算...
$user->invoiceFor('One Time Fee', 500);

// Braintree 以美元结算...
$user->invoiceFor('One Time Fee', 5);
```

生成发票将立即对用户的信用卡收取费用。`invoiceFor` 方法可以接受一个数据作为第三个参数，允许你在 Stripe / Braintree 结算时传递任何可用的参数。

```
$user->invoiceFor('One Time Fee', 500, [
    'custom-option' => $value,
]);

```

{note} `invoiceFor` 方法在当多次尝试失败支付后也会产生一个 Stripe 发票。如果你不需要为失败重试的支付生成发票，你需要在第一次失败结算时就调用 Stripe 的 API 关闭它们。

发票

你可以使用 `invoices` 方法轻松获取 `billable` 模型的所有发票信息：

```
$invoices = $user->invoices();

// 在结果中包含待开发票...
$invoices = $user->invoicesIncludingPending();
```

当将发票信息列出来给用户时，你可能需要使用发票的辅助函数来显示相关的发票信息。例如，你可能希望将发票列表以表格的形式显示，允许用户轻松的下载它们：

```
<table>
    @foreach ($invoices as $invoice)
        <tr>
            <td>{{ $invoice->date()->toFormattedDateString() }}</td>
            <td>{{ $invoice->total() }}</td>
            <td><a href="/user/invoice/{{ $invoice->id }}">Download</a></td>
        </tr>
    @endforeach
</table>
```

生成发票的 PDFs

在生成发票的 PDF 文件之前，你需要先安装 `dompdf` PHP 库：

```
composer require dompdf/dompdf
```

然后，在路由或控制器内，使用 `downloadInvoice` 方法来生成一个发票的 PDF 的下载响应。浏览器会自动开始下载该文件：

```
use Illuminate\Http\Request;

Route::get('user/invoice/{invoice}', function (Request $request, $invoiceId) {
    return $request->user()->downloadInvoice($invoiceId, [
        'vendor' => 'Your Company',
        'product' => 'Your Product',
    ]);
});
```

Laravel 的远程服务器任务处理器 Envoy

- 简介
 - 安装
- 编写任务
 - 任务启动
 - 任务变量
 - 任务故事
 - 多个服务器
- 运行任务
 - 任务确认
- 通知
 - Slack

简介

[Laravel Envoy](#) 为定义在远程服务器上运行的通用任务提供了一种简洁、轻便的语法。它使用了 Blade 风格的语法，让你可以很方便的启动任务来进行项目部署、Artisan 命令运行等操作。目前，Envoy 只支持 Mac 及 Linux 操作系统。

安装

首先，使用 Composer `global require` 命令来安装 Envoy：

```
composer global require "laravel/envoy=~1.0"
```

因为 Composer 的全局库有时会导致包的版本冲突，所以你可以考虑使用 `cgr`，它是 `composer global require` 命令的一种替代实现 `cgr` 库的安装指导可以在 [GitHub上找到](#)。

{note} 一定要确保 `~/.composer/vendor/bin` 目录加入到了你的 PATH 中，这样才能在命令行运行 `envoy`。

更新 Envoy

你也可以使用 Composer 来更新 Envoy 到最新版本。要注意 `composer global update` 命令是更新你所有在全局安装的包：

```
composer global update
```

编写任务

所有的 Envoy 任务都必须定义在项目根目录的 `Envoy.blade.php` 文件中，这里有个例子：

```
@servers(['web' => ['user@192.168.1.1']])

@task('foo', ['on' => 'web'])
    ls -la
@endtask
```

如你所见，`@servers` 的数组被定义在文件的起始位置处，让你在声明任务时可以在 `on` 选项里参考使用这些服务器。在你的 `@task` 声明里，你可以放置当任务运行时想要在远程服务器运行的 Bash 命令。

你可以通过指定服务器的 IP 地址为 `127.0.0.1` 来执行本地任务：

```
@servers(['localhost' => '127.0.0.1'])
```

任务启动

有时，你可能想在任务启动前运行一些 PHP 代码。这时可以使用 `@setup` 区块在 Envoy 文件中声明变量以及运行普通的 PHP 程序：

```
@setup
    $now = new DateTime();

    $environment = isset($env) ? $env : "testing";
@endsetup
```

如果你想在任务执行前引入其他 PHP 文件，可以直接在 `Envoy.blade.php` 文件起始位置使用

```
@include :
```

```
@include('vendor/autoload.php')

@task('foo')
    # ...
@endtask
```

任务变量

如果需要的话，你也可以通过命令行选项来传递变量至 Envoy 文件，以便自定义你的任务：

```
envoy run deploy --branch=master
```

你可以通过 Blade 的「echo」语法使用这些选项，当然也能在任务里用「if」和循环操作。举例来说，我们在执行 `git pull` 命令前，先检查 `$branch` 变量是否存在：

```
@servers(['web' => '192.168.1.1'])
```

```

@task('deploy', ['on' => 'web'])
    cd site

    @if ($branch)
        git pull origin
    @endif

    php artisan migrate
@endtask

```

任务故事

任务故事通过一个统一的、便捷的名字来划分一组任务，来让你把小而专的子任务合并到大的任务里。比如说，一个名为 `deploy` 的任务故事可以在它定义范围内列出子任务名字 `git` 和 `composer` 来运行各自对应的任务：

```

@servers(['web' => '192.168.1.1'])

@story('deploy')
    git
    composer
@endstory

@task('git')
    git pull origin master
@endtask

@task('composer')
    composer install
@endtask

```

当 `story` 写好后，像运行普通任务一样运行它就好了：

```
envoy run deploy
```

多个服务器

你可以在多个服务器上运行任务。首先，增加额外的服务器至你的 `@servers` 声明，每个服务器必须分配一个唯一的名称。一旦你定义好其它服务器，就能在任务声明的 `on` 数组中列出这些服务器：

```

@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
    php artisan migrate

```

```
@endtask
```

并行运行

默认情况下，任务会按照顺序在每个服务器上运行。这意味着任务会在第一个服务器运行完后才跳到第二个。如果你想在多个服务器上并行运行任务，只需简单的在任务声明里加上 `parallel` 选项即可：

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2'], 'parallel' => true])
    cd site
    git pull origin
    php artisan migrate
@endtask
```

运行任务

要想运行一个在 `Envoy.blade.php` 文件中定义好的任务或者故事，就执行 Envoy 的 `run` 命令，并将这个任务的名字传递给它。Envoy 会去执行这个任务并且把任务执行过程中的输出给打印出来：

```
envoy run task
```

任务确认

如果你想要在运行任务之前进行提示确认，则可以增加 `confirm` 命令到任务声明。这个选项对于破坏性的操作来说是相当有用的：

```
@task('deploy', ['on' => 'web', 'confirm' => true])
    cd site
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

[](#)

通知

Slack

Envoy 也支持任务执行完毕后发送通知至 [Slack](#)。`@slack` 命令接收 Slack hook 网址和频道名称。你可以通在在 Slack 的控制面板上创建「Incoming WebHooks」时来检索 webhook 网址。`webhook-url` 参数必须是 `@slack` 的 Incoming WebHooks 所提供的完整网址：

```
@finished
  @slack('webhook-url', '#bots')
@endfinished
```

你可以选择下方的任意一个来作为 channel 参数：

- 如果要发送通知至一个频道：`#channel`
- 如果要发送通知给一位用户：`@user`

Laravel 的搜索系统 Scout

- [简介](#)
- [安装](#)
 - [队列](#)
 - [驱动的必要设置](#)
- [配置](#)
 - [配置模型索引](#)
 - [配置可检索的数据](#)
- [索引](#)
 - [批量导入](#)
 - [添加记录](#)
 - [更新记录](#)
 - [删除记录](#)
 - [暂停索引](#)
- [检索](#)
 - [Where 语句](#)
 - [分页](#)
- [自定义引擎](#)

简介

Laravel Scout 是针对 [Eloquent 模型](#) 开发的一个简单的，基于驱动的全文检索系统。Scout 使用模型观察者时会自动保持你的检索索引与你的 Eloquent 记录同步。

目前，Scout 带着一个 [Algolia](#) 驱动；然而，扩展 Scout 并不难，你可以通过自定义驱动来自由的扩展 Scout。

安装

首先，使用 composer 包管理器来安装 Scout：

```
composer require laravel/scout
```

接下来，你需要将 `ScoutServiceProvider` 添加到你的 `config/app.php` 配置文件的 `providers` 数组中：

```
Laravel\Scout\ScoutServiceProvider::class,
```

注册好 Scout 的服务提供者之后，你可以使用 `vendor:publish` Artisan 命令生成 Scout 的配置文件。这个命令会在你的 `config` 目录下生成 `scout.php` 配置文件：

```
php artisan vendor:publish --provider="Laravel\Scout\ScoutServiceProvider"
```

最后，将 `Laravel\Scout\Searchable` trait 加到你想要做检索的模型，这个 trait 会注册一个模型观察者来保持模型同步到检索的驱动：

```
<?php

namespace App;

use Laravel\Scout\Searchable;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    use Searchable;
}
```

队列

虽然 Scout 没有限制你必须使用队列，但是建议你为 Scout 配置一个 [队列驱动](#)。使用队列来对处理 Scout 对数据模型的索引，将会极大的提高你的页面响应时间。

一旦你配置了队列驱动，在你的 `config/scout.php` 配置文件中设置 `queue` 的值为 `true`：

```
'queue' => true,
```

驱动必要设置

Algolia

当你使用 Algolia 驱动时，你需要在你的 `config/scout.php` 配置文件配置你的 Algolia `id` 和 `secret` 认证资料。配置好认证资料以后，你还需要使用 composer 包管理器安装 Algolia PHP SDK：

```
composer require algolia/algoliasearch-client-php
```

配置

配置模型索引

每一个 Eloquent 模型都会同步到对应的一个检索「索引」中，「索引」里包含了此模型的所有可检索的记录。你可以把每一个「索引」设想为一张 MySQL 数据表。默认情况下，「索引」的名称与模型对应数据表名称一致，也就是说，是模型名称的复数形式。当然，你也可以在模型类使用 `searchableAs` 方法来重写「索引」名称：

```
<?php

namespace App;

use Laravel\Scout\Searchable;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    use Searchable;

    /**
     * 得到该模型索引的名称。
     *
     * @return string
     */
    public function searchableAs()
    {
        return 'posts_index';
    }
}
```

配置可检索的数据

默认情况下，「索引」会从模型的 `toArray` 方法中读取数据来做数据持久化。你也可以通过重写 `toSearchableArray` 方法来自定义数据到「索引」的同步：

```
<?php

namespace App;

use Laravel\Scout\Searchable;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    use Searchable;

    /**
     * 得到该模型可索引数据的数组。
     *
     * @return array
     */
    public function toSearchableArray()
    {
        $array = $this->toArray();

        // 自定义数组数据...
    }
}
```

```

        return $array;
    }
}

```

索引

批量导入

如果你想要将 Scout 安装到已经存在的项目里，那你也需要将已经在数据库里的数据导入到搜索引擎里。你可以使用 Scout 提供的 `import` Artisan 命令把现有的模型数据导入到「索引」里：

```
php artisan scout:import "App\Post"
```

添加记录

当你将 `Laravel\Scout\Searchable` 添加到模型之后，你只需要 `save` 一个模型实例，它就会自动的添加到你检索的索引里。如果你的 Scout 配置里 [使用队列](#) 这个操作会在后台由你的 queue worker 执行：

```

$order = new App\Order;

// ...

$order->save();

```

使用队列添加

如果你想使用 Eloquent 构造器添加模型的集合到你的检索索引里，你也可以在 Eloquent 构造器上链式调用 `searchable` 方法。`searchable` 会把构造器的查询 [结果分块](#) 并且将记录添加到你的检索索引里。同样的，如果你配置 Scout 使用队列，所有的数据块将会在后台由你的 queue workers 添加：

```

// 使用 Eloquent 查询语句增加...
App\Order::where('price', '>', 100)->searchable();

// 你也可以使用模型关系增加记录...
$user->orders()->searchable();

// 你也可以使用集合增加记录...
$orders->searchable();

```

`searchable` 方法可以被看做是「增量更新」的操作。换句话说，如果一个模型的记录已经在你的索引里了，它就会被更新，如果不在，就会被插入。

更新记录

你只需要更新一个模型实例的属性并且 `save` 这个模型到你的数据库里，就更新了这个可检索的模型。Scout 会自动更新这个变化到你的检索索引里：

```
$order = App\Order::find(1);

// 更新 order...

$order->save();
```

你也可以在 Eloquent 查询语句上使用 `searchable` 方法来更新一个模型的集合。如果这个模型不存在你检索的索引里，就会被创建：

```
// 使用 Eloquent 查询语句更新...
App\Order::where('price', '>', 100)->searchable();

// 你也可以使用模型关系更新...
$user->orders()->searchable();

// 你也可以使用集合更新...
$orders->searchable();
```

删除记录

简单的使用 `delete` 从数据库删除模型就可以移除索引里的记录。这种移除的方式也兼容 软删除 模型：

```
$order = App\Order::find(1);

$order->delete();
```

或者说是你不想在删除记录之前检索模型，你也可以在 Eloquent 查询语句的实例或集合上使用 `unsearchable` 方法：

```
// 使用 Eloquent 查询语句移除索引...
App\Order::where('price', '>', 100)->unsearchable();

// 你也可以使用模型关系移除索引...
$user->orders()->unsearchable();

// 你也可以使用集合移除索引...
$orders->unsearchable();
```

暂停索引

当你想要对 Eloquent 模型完成一系列的操作并且不想将它们的数据同步到检索的索引里时，你也可以使用 `withoutSyncingToSearch` 方法。这个方法接受一个立即执行的回调函数。函数内部所有的操作都不会被同步到模型的索引里：

```
App\Order::withoutSyncingToSearch(function () {
    // Perform model actions...
});
```

检索

你可以对一个模型使用 `search` 方法来检索。这个 `search` 方法接受一个将要在模型里检索的简单的字符串。之后你可以在搜索语句上链式调用 `get` 方法得到匹配的 Eloquent 模型：

```
$orders = App\Order::search('Star Trek')->get();
```

当 Scout 检索到数据后，会返回一个 Eloquent 模型的集合，你也可以在路由或控制器上直接返回数据，这样它会被自动解析成 JSON 格式：

```
use Illuminate\Http\Request;

Route::get('/search', function (Request $request) {
    return App\Order::search($request->search)->get();
});
```

Where 语句

Scout 允许你增加一个简单的「where」语句链接到搜索语句上。目前，这些语句只支持简单的数字相等检查，并且主要用来查询范围内的拥有者的 ID。由于检索索引是非关系型数据库，更高级的「where」暂时不支持：

```
$orders = App\Order::search('Star Trek')->where('user_id', 1)->get();
```

分页

除了检索模型的集合，你也可以使用 `paginate` 方法对检索结果进行分页。这个方法会返回一个 `Paginator` 实例就像你 [对传统的 Eloquent 查询语句进行分页](#)：

```
$orders = App\Order::search('Star Trek')->paginate();
```

你可以通过通过 `paginate` 方法的第一个参数来指定检索的结果每页要显示的数量：

```
$orders = App\Order::search('Star Trek')->paginate(15);
```

一旦你获取到结果，就可以对结果进行显示，就像你对传统的 Eloquent 查询语句进行分页一样，使用 [Blade](#) 来渲染页面的链接：

```
<div class="container">
    @foreach ($orders as $order)
        {{ $order->price }}
    @endforeach
</div>

{{ $orders->links() }}
```

自定义引擎

写一个引擎

如果 Scout 内建的引擎不能满足你的需求，你可以写你自定义的引擎并且将它注册到 Scout。你的引擎需要扩展 `Laravel\Scout\Engines\Engine` 抽象类，这个抽象类包含了五种你自定义的引擎必须要实现的方法：

```
use Laravel\Scout\Builder;

abstract public function update($models);
abstract public function delete($models);
abstract public function search(Builder $builder);
abstract public function paginate(Builder $builder, $perPage, $page);
abstract public function map($results, $model);
```

在 `Laravel\Scout\Engines\AlgoliaEngine` 类里回顾这些方法会对你有较大的帮助。这个类将为你提供一个良好的学习起点，学习如何在你自己的引擎中实现这些方法。

注册引擎

一旦你写好了自己的引擎，你可以用 Scout 管理引擎的 `extend` 方法将它注册到 Scout。你只需要在你的 `AppServiceProvider` 下的 `boot` 方法调用 `extend` 方法，或者是你的应用下使用的其他任何一个服务提供者。举个例子，如果你写好了一个 `MySqlSearchEngine`，你可以像这样去注册它：

```
use Laravel\Scout\EngineManager;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    resolve(EngineManager::class)->extend('mysql', function () {
        return new MySqlSearchEngine;
```

```
    });
}
```

一旦你的引擎注册好了，你可以在 `config/scout.php` 配置文件中指定它为默认的 Scout driver：

```
'driver' => 'mysql',
```

Laravel 的 社会化登录功能

- 简介
- 授权
- 官方文档
- 配置
- 基本用法
 - 无状态身份验证
 - 检索用户详细信息
 - 从令牌检索用户详细信息

简介

Laravel 社会化登录通过 Facebook , Twitter , Google , LinkedIn , GitHub 和 Bitbucket 提供了一个富有表现力的，流畅的 OAuth 身份验证界面。它几乎能处理所有你害怕处理的各种样板社会认证代码。

翻译自 Readme: <https://github.com/laravel/socialite>

我们不接受新的适配器。

社区驱动的社会化登录提供商网站上可以找到为其他平台提供的适配器列表。

授权

Laravel 社会化登录是根据 [MIT 授权](#) 许可的开源软件

官方文档

除了常规的基于表单的身份验证之外， Laravel 也提供了一种简单，方便的办法来使用 [Laravel 社会化登录](#) 向 OAuth 提供程序进行身份验证。社会化登录目前支持 Facebook , Twitter , LinkedIn , Google , GitHub 和 Bitbucket 的身份验证。

要开始社会化登录，使用 composer 将相应包加入到你项目的依赖项中。

```
composer require laravel/socialite
```

配置

安装完社会化登录库之后，在你的 config/app.php 文件中注册 Laravel\\Socialite\\SocialiteServiceProvider 。

```
'providers' => [
```

```
// Other service providers...

Laravel\Socialite\SocialiteServiceProvider::class,
],
```

同时，在你的 `app` 配置文件中，把 `Socialite facade` 加入到 `aliases` 数组中。

```
'Socialite' => Laravel\Socialite\Facades\Socialite::class,
```

你还需要为你应用使用的 OAuth 服务加入凭据。这些凭据应该放在你的 `config/services.php` 文件中，并且使用 `facebook`，`twitter`，`linkedin`，`google`，`github` 或 `bitbucket` 作为键名，具体取决于在你的应用中由哪个程序来提供验证服务，比如：

```
'github' => [
    'client_id' => 'your-github-app-id',
    'client_secret' => 'your-github-app-secret',
    'redirect' => 'http://your-callback-url',
],
```

基本用法

接下来，你已经准备好了验证用户了！你需要两个路由：一个重定向用户到 OAuth 提供商，另一个在提供商验证之后接回调。我们用 `Socialite facade` 来访问：

```
<?php

namespace App\Http\Controllers\Auth;

use Socialite;

class LoginController extends Controller
{
    /**
     * Redirect the user to the GitHub authentication page.
     *
     * @return Response
     */
    public function redirectToProvider()
    {
        return Socialite::driver('github')->redirect();
    }

    /**
     * Obtain the user information from GitHub.
     *
     * @return Response
     */
}
```

```

public function handleProviderCallback()
{
    $user = Socialite::driver('github')->user();

    // $user->token;
}
}

```

`redirect` 方法负责发送用户到 OAuth 提供商，而 `user` 方法将读取传入的请求并从提供商处检索用户信息。在重定向用户之前，你还可以使用 `scope` 方法来设置请求的「scope」。这个方法会覆盖所有现有的范围。

```

return Socialite::driver('github')
    ->scopes(['scope1', 'scope2'])->redirect();

```

当然，你需要定义通往你的控制器方法的路由。

```

Route::get('login/github', 'Auth\LoginController@redirectToProvider');
Route::get('login/github/callback', 'Auth\LoginController@handleProviderCallback');

```

一部分 OAuth 提供商在重定向请求中支持携带可选参数。要在请求中包含任何可选参数，调用 `with` 方法时传入可选的数组即可。

```

return Socialite::driver('google')
    ->with(['hd' => 'example.com'])->redirect();

```

当使用 `with` 方法时，注意不要传递保留关键字，比如 `state` 或 `response_type`。

无状态身份验证

`stateless` 方法可以用于禁用 session 状态的验证，这个方法在向 API 添加社会化身份验证时非常有用。

```

return Socialite::driver('google')->stateless()->user();

```

检索用户详细信息

一旦你有了一个用户实例，你可以获取这个用户的更多详细信息：

```

$user = Socialite::driver('github')->user();

// OAuth Two Providers
$token = $user->token;
$refreshToken = $user->refreshToken; // not always provided
$expiresIn = $user->expiresIn;

```

```
// OAuth One Providers
$token = $user->token;
$tokenSecret = $user->tokenSecret;

// All Providers
$user->getId();
$user->getNickname();
$user->getName();
$user->getEmail();
$user->getAvatar();
```

从令牌检索用户详细信息

如果你已经有了一个用户的有效访问令牌，你可以使用 `userFromToken` 方法检索用户的详细信息。

```
$user = Socialite::driver('github')->userFromToken($token);
```