

Laravel 5.3 中文文档



本文档由 [Laravel 学院](#) 提供翻译支持

目录

1. 序言	3
1.1 发行版本说明	3
1.2 升级指南	13
1.3 贡献代码	30
2. 起步	33
2.1 安装	33
2.2 配置	36
2.3 目录结构	40
2.4 错误&日志	45
3. 开发环境	52
3.1 Laravel Homestead	52
3.2 Laravel Valet	62
4. 核心概念	71
4.1 服务容器	71
4.2 服务提供者	79
4.3 门面 (Facades)	84
4.4 契约 (Contracts)	92
5. HTTP 层	100
5.1 路由	100
5.2 中间件	110
5.3 CSRF 保护	119
5.4 控制器	122
5.5 请求	133
5.6 响应	145
5.7 Session	155
5.8 验证	164
6. 视图 & 模板	192
6.1 视图	192
6.2 Blade 模板	200
6.3 本地化	213
7. JavaScript & CSS	217
7.1 起步	217
7.2 编译资源 (Laravel Elixir)	220
8. 安全	231
8.1 用户认证	231
8.2 用户授权	249
8.3 密码重置	261
8.4 API 认证 (Passport)	264

8.5 加密	287
8.6 哈希 (Hashing)	289
9. 综合话题.....	292
9.1 事件广播	292
9.2 缓存	311
9.3 事件	321
9.4 文件存储	332
9.5 邮件	344
9.6 通知	360
9.7 队列	386
10. 数据库.....	407
10.1 起步	407
10.2 查询构建器.....	414
10.3 分页	433
10.4 迁移	439
10.5 填充数据	455
10.6 Redis	458
11. Eloquent ORM.....	463
11.1 起步	463
11.2 关联关系	488
11.3 集合	516
11.4 访问器&修改器	518
11.5 序列化	523
12. Artisan Console.....	528
12.1 控制台命令	528
12.2 任务调度	542
13. 测试	550
13.1 起步	550
13.2 应用测试	552
13.3 数据库	563
13.4 模拟	570
14. 官方包	582
14.1 Laravel Cashier	582
14.2 Envoy Task Runner	604
14.3 Laravel Scout	610
14.4 Laravel Socialite	622
15. 附录	626
15.1 集合	626
15.2 辅助函数	659
15.3 包开发	681

1. 序言

1.1 发行版本说明

1、支持政策

对于 LTS 版本，比如 [Laravel](#) 5.1，我们将会提供为期两年的 bug 修复和三年的安全修复支持。

LTS 版本将会提供最长时间的支持和维护。

对于其他通用版本，只提供六个月的 bug 修复和一年的安全修复支持，比如 Laravel [5.3](#)。

2、Laravel 5.3

Laravel 5.3 在 5.2 的基础上继续进行优化，提供了大量新功能和新特性：基于驱动的[通知](#)系统；

通过 Laravel [Echo](#) 提供强大的实时支持；通过 Laravel [Passport](#) 实现无痛的 [OAuth2](#) 服务器；通过

Laravel [Scout](#) 实现全文模型[搜索](#)；在 Laravel [Elixir](#) 中支持 Webpack；“可邮寄”的对象；明确分离

web 和 api [路由](#)；基于闭包的控制台命令；存储上传文件的辅助函数；支持 POPO 和单动作控制器；

以及优化[前端](#)脚手架；等等等等。

通知（[Notifications](#)）

注：Laracasts 上有关于此特性的免费[视频教程](#)。

Laravel Notifications 为我们提供了简单、优雅的 [API](#) 用于在不同的发行渠道中发送通知，例如[邮](#)

[件](#)、SMS、Slack 等等。例如，你可以定义一个单据已经支付的通知，然后通过邮件和 SMS 发送这

个通知，你可以使用一个很简单的来实现：

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

[Laravel 社区](#)已经为通知系统编写了各种各样的驱动，包括对 iOS 和 Android 通知的支持，要学习更多关于通知系统的细节，查看其[相应文档](#)。

WebSockets / 事件广播

事件广播在之前版本的 Laravel 中已经有了，Laravel 5.3 通过为已私有和已存在的 WebSocket 频道添加频道级认证对此功能进行了极大的优化和提升：

```
/*
 * Authenticate the channel subscription...
 */
Broadcast::channel('orders.*', function ($user, $orderId) {
    return $user->placedOrder($orderId);
});
```

Laravel Echo，通过 NPM 安装的全新的 JavaScript 包，将和 Laravel 5.3 一起发布，用于为订阅频道以及在客户端 JavaScript 应用中监听服务器端事件提供了简单、优美的 API，Echo 包含对 [Pusher](#) 和 [Socket.io](#) 的支持：

```
Echo.channel('orders.' + orderId)
    .listen('ShippingStatusUpdated', (e) => {
```

```
    console.log(e.description);  
});
```

为了订阅到传统频道， Laravel Echo 还使得订阅到提供谁在监听给定频道信息的已存在频道变得简单：

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

要学习更多关于 Echo 和事件广播的内容，请参考其[对应文档](#)。

Laravel Passport (OAuth2 服务器)

注：Laracasts 上提供了关于这一新特性的免费[视频教程](#)。

Laravel 5.3 使用 Laravel Passport 让 API 认证变得简单。 Laravel Passport 可以在几分钟内为应用提供一个完整的 OAuth2 服务器实现，Passport 基于 Alex Bilbie 维护的 [League OAuth2 server](#) 实现。

Passport 使得通过 OAuth2 授权码获取访问令牌（access token）变得轻松，你还可以允许用户通

过 Web UI 创建“个人访问令牌”。为了让你更快上手，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，欢迎提供你自己的用于管理客户端和访问令牌的前端后台。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。

当然，Passport 还让定义可能在应用消费你的 API 期间被请求的访问令牌域变得简单：

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

此外，Passport 还包含了用于验证访问令牌认证请求包含必要令牌域的中间件：

```
Route::get('/orders/{order}/status', function (Order $order) {
    // Access token has "check-status" scope...
})->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 令牌来实现这一功能，从而让开发者专注于业务开发。

想要学习更多 Passport 细节，请查看其[文档](#)。

搜索 (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 记录的搜索索引，目前，Scout 使用 [Algolia](#) 驱动，不过，编写自己的驱动很简单，你可以通过自己的搜索实现扩展 Scout。

你可以简单通过添加 Searchable trait 到模型让模型变得可搜索：

```
<?php

namespace App;

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

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

trait 被添加到模型之后，当保存模型实例的时候其信息将会被同步到搜索索引：

```
$order = new Order;

// ...
```

```
$order->save();
```

模型被索引之后，就可以通过模型进行全文搜索了，甚至还可以对搜索结果进行分页：

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

当然，Scout 还有很多其他特性，具体请查看其[文档](#)。

可邮寄对象

注：Laracasts 上有关于该特性的免费[视频教程](#)。

Laravel 5.3 支持可邮寄对象，这些对象可以以一个简单对象的形式表示邮件信息，而不再需要在闭包中自定义邮件信息，例如，你可以定义一个简单的邮寄对象用作欢迎邮件：

```
class WelcomeMessage extends Mailable  
{  
    use Queueable, SerializesModels;  
  
    /**  
     * Build the message.  
     *  
     * @return $this  
     */  
  
    public function build()
```

```
{  
    return $this->view('emails.welcome');  
}  
}
```

可邮寄对象被创建以后，你可以使用一个简单、优雅的 API 将其发送给用户。可邮寄对象可以在浏览代码的同时了解邮件信息：

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

当然，你还可以标记可邮寄对象为“队列化”，这样这封邮件就会在后台通过队列任务发送：

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

更多可邮寄对象细节请查看其[对应文档](#)。

存储上传文件

注：Laracasts 上有关于该特性的免费[视频教程](#)。

在 web 应用中，最常见的任务之一就是保存用户上传文件了，比如头像、照片、[文档](#)等等。Laravel 5.3 通过在上传文件实例上使用新的 `store` 方法让这一工作变得简单。只需要简单调用 `store` 方法并传入文件保存路径即可：

```
/**  
 * Update the avatar for the user.  
 *  
 * @param Request $request  
 * @return Response  
 */  
  
public function update(Request $request)  
{  
    $path = $request->file('avatar')->store('avatars', 's3');  
  
    return $path;  
}
```

更多细节请查看其[完整文档](#)。

Webpack & Laravel Elixir

Laravel Elixir 6.0 和 Laravel 5.3 一起发布，新版本将捆绑支持 Webpack 和 Rollup JavaScript 模块。

默认情况下，Laravel 5.3 的 `gulpfile.js` 文件现在已经使用 Webpack 来编译 JavaScript：

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

查看完整的 [Laravel Elixir 文档](#) 了解更多信息。

前端架构

注：Laracasts 有关于本特性的免费[视频教程](#)。

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` 命令生成的认证脚手架。

不再从 CDN 中加载前端资源，所有依赖都被定义在默认的 `package.json` 文件中。

此外，支持单文件的 Vue 组件现在已经开箱支持，`resources/assets/js/components` 目录下包含了一个简单的示例组件 `Example.vue`，新的 `resources/assets/js/app.js` 文件将会启动被配置你的 JavaScript 库以及 Vue 组件。

这种架构对开始开发现代的、强大的 JavaScript 应用提供了更好的指导，而不需要要求应用使用任何给定 JavaScript 或者 CSS 框架。关于如何进行现代 Laravel 前端开发，请查看[对应文档](#)。

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` 下包含两个 HTTP 路由文件。`web` 和 `api` 路由文件在如何分割 Web 界面和 API 路由方面提供了指导。`api` 路由文件中的路由会通过 `RouteServiceProvider` 自动添加 `api` 前缀和 `auth:api` 中间件。

闭包控制台命令

除了通过命令类定义之外，现在 Artisan 命令还可以在 `app\Console\Kernel.php` 文件的 `commands` 方法中以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...');  
});
```

更多详情请查看完整的 [Artisan 文档](#)。

\$loop 变量

注：Laracasts 中有关于此特性的免费[视频教程](#)。

当我们在 [Blade](#) 模板中[循环](#)遍历的时候，\$loop 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
```

更多详情请查看完整的 [Blade 文档](#)。

3、早期版本

更多早期版本发行说明：

- [Laravel 5.2 发行版本说明](#)
- [Laravel 5.1 发行版本说明](#)

1.2 升级指南

从 5.2 升级到 5.3

注：预计升级时间：2-3 小时

PHP & HHVM

[Laravel](#) 5.3 要求 PHP 5.6.4 及以上版本，官方将不再支持 HHVM，因为其不包含 PHP 5.6+新提供的语言特性。

废弃

所有罗列在 [Laravel 5.2 升级指南](#) 中的废弃功能都已从框架中移除，你需要查看这个列表以确定不再使用这些废弃功能。

数组

key/value 顺序更改

[Arr](#) 类上的 `first`、`last`、以及 `contains` 方法现在将“value”作为第一个参数传递给给定闭包，例如：

```
Arr::first(function ($value, $key) {
```

```
    return ! is_null($value);  
});
```

在 Laravel 之前版本中，`$key` 是第一个参数，但是由于大多数使用案例只对`$value` 感兴趣，所以我们将其放到第一个。你可以在应用中进行一次全局搜索以验证是否你在应用中通过旧的方式使用了这个函数。

Artisan

make:console 命令

`make:console` 命令现在被重命名为 `make:command`。

认证

认证脚手架

Laravel 框架提供的默认的两个认证控制器已经被分割成四个，这一更改让认证控制器变得更加清爽、责任更加明确。升级应用认证控制器到最新的最简单方法就是从 [GitHub 上将四个控制器代码](#) 拷贝过来复制到项目中。

你还要确保在路由文件中调用了 `Route::auth()` 方法，该方法在底层已经为新控制器注册了合适的路由。

这些新控制器拷贝到应用后，需要重新实现之前在认证控制器中实现的方法和业务。例如，如果你在自定义用于认证的 guard，需要重写控制器的 `guard` 方法，你可以检查每个认证控制器的 trait 以判断要重写哪些方法。

注：如果你没有自定义认证控制器，只需要将代码从 Github 拷贝到本地项目，并确保在路由文件中

调用了 `Route::auth` 方法。

密码重置邮件

密码重置邮件现在使用新的通知功能 (Laravel Notifications) , 如果你想要在发送密码重置链接的时候自定义通知发送 , 需要重写 `Illuminate\Auth\Passwords\CanResetPassword` trait 上的 `sendPasswordResetNotification` 方法。

`User` 模型必须使用新的 `Illuminate\Notifications\Notifiable` trait 以便邮件重置链接可以正常发送 :

```
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;
}
```

注 : 不要忘了在配置文件 `config/app.php` 的 `providers` 数组中注册服务提供者 `Illuminate\Notifications\NotificationServiceProvider`。

POST 到 Logout

`Route::auth` 方法现在为 `/logout` 注册了一个 POST 路由取代之前的 GET 路由。这可以防止其它

应用让用户从应用中退出。想要升级的话，需要将原来的退出请求转化为 POST 请求方式，或者

为 `/logout` URI 自定义 GET 路由：

```
Route::get('/logout', 'Auth\LoginController@logout');
```

授权

使用类名调用策略方法

有时候一些策略方法只接收当前认证用户而不是授权模型实例，最常见的场景就是授权 `create` 动作。例如，如果你在创建一篇博客，你可能希望检查当前用户是否被授权创建文章。

当定义一个不接收模型实例的策略方法时，比如 `create` 方法，对应类名就不再需要以第二个参数的方式传入，只需要传入认证用户实例即可：

```
/**
 * Determine if the given user can create posts.
 *
 * @param \App\User $user
 * @return bool
 */
public function create(User $user)
{
    //
}
```

AuthorizesResources Trait

`AuthorizesResources` trait 已经和 `AuthorizesRequests` trait 合并到一起，你需要从 `app/Http/Controllers/Controller.php` 中移除 `AuthorizesResources` trait。

Blade

自定义指令

在之前版本的 Laravel 中，我们使用 `directive` 方法注册自定义的 Blade 指令，传递给指回调的 `$expression` 参数包含了最外层的括号。在 Laravel 5.3 中，这些最外层的括号将不再包含在传递给指回调的表达式中，请查看 Blade [文档](#) 确保自定义的 Blade 指令还能正常工作。

广播

服务提供者

Laravel 5.3 对事件广播进行了显著的优化，需要添加新的 `BroadcastServiceProvider` (从 [GitHub 下载文件](#)) 到 `app/Providers` 目录，然后将这个新的服务提供者注册到配置文件 `config/app.php` 的 `providers` 数组中。

缓存

扩展闭包绑定`&$this`

使用闭包调用 `Cache::extend` 方法时，`$this` 会被绑定到 `CacheManager` 实例，从而允许你在扩展闭包中调用其提供的方法：

```
Cache::extend('memcached', function ($app, $config) {
    try {
```

```
        return $this->createMemcachedDriver($config);  
    } catch (Exception $e) {  
        return $this->createNullDriver($config);  
    }  
});
```

Cashier

如果你在使用 Cashier，需要升级 `laravel/cashier` 扩展包到 7.0 版本，这一版本的 Cashier 只修改了一些内置方法以便兼容于 Laravel 5.3，并没有什么重大更新。

集合

key/value 顺序调整

集合方法 `first`, `last` 和 `contains` 都将“value”作为第一个参数传递给相应的回调闭包，例如：

```
$collection->first(function ($value, $key) {  
    return ! is_null($value);  
});
```

在 Laravel 之前的版本中，`$key` 是第一个参数，由于大部分使用案例只对`$value` 感兴趣，所以将其调整为第一个，你需要在应用中对这些方法做一个全局搜索，以验证`$value` 是否按照期望的方式以第一个参数传入闭包。

where 默认使用非严格比较

`where` 现在默认使用非严格比较而不是之前的严格比较，如果你想要进行严格比较，可以使用 `whereStrict` 方法。

`where` 方法也不再接收第三个参数标识“严格”，你需要基于应用的需求区分调用 `where` 或 `whereStrict`。

控制器

构造函数当中使用 Session

在 Laravel 以前的版本中，你可以在控制器构造函数中获取 session 变量或者认证后的用户实例。框架从未打算具有如此明显的特性。在 Laravel 5.3 中，你在控制器构造函数中不再能够直接获取到 session 变量或认证后的用户实例，因为中间件还未启动。

仍然有替代方案，那就是在控制器构造函数中使用 Closure 来直接定义中间件。请注意，在使用这个方案的时候，确保你所使用的 Laravel 版本高于 5.3.4：

```
<?php

namespace App\Http\Controllers;

use App\User;
use Illuminate\Support\Facades\Auth;
use App\Http\Controllers\Controller;

class ProjectController extends Controller
{
    /**
     * 当前用户的的所有 Projects.
     */
}
```

```
protected $projects;

/**
 * 创建新的控制器实例.
 *
 * @return void
 */

public function __construct()
{
    $this->middleware(function ($request, $next) {
        $this->projects = Auth::user()->projects;
        return $next($request);
    });
}

}
```

当然，你可以在控制器的其他方法中依靠 `\Illuminate\Http\Request` 类获取到 session 以及认证用户信息。

```
/*
 * 获取当前用户的所有 Projects.
 *
 * @param \Illuminate\Http\Request $request
 * @return Response
 */

public function index(Request $request)
```

```
{  
    $projects = $request->user()->projects;  
  
    $value = $request->session()->get('key');  
  
    //  
  
}
```

声明：控制器 session 部分由网友 AC1982 (微信号) 提供翻译支持。

数据库

集合

查询构建器现在返回 `Illuminate\Support\Collection` 实例而不是原生数组，以便保持和 Eloquent 返回结果类型一致。

如果你不想要迁移查询构建器结果到 Collection 实例，可以在查询构建器的 `get` 方法后调用 `call` 方法，这将会返回原生的 PHP 数组结果，从而保证向后兼容：

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

Eloquent `$morphClass` 属性

可以在 Eloquent 模型上定义的 `$morphClass` 属性已经被移除，以便定义一个“morph map”（变形映射），定义变形映射可以支持渴求式加载并且解决使用多态关联关系引起的额外 bugs，如果你之前使用了 `$morphClass` 属性，需要使用如下语法将其迁移到 `morphMap`：

```
Relation::morphMap([  
    'YourCustomMorphName' => YourModel::class,
```

```
]);
```

例如，如果你之前定义了如下 `$morphClass`：

```
class User extends Model
{
    protected $morphClass = 'user'
}
```

现在则需要在 `AppServiceProvider` 的 `boot` 方法中定义如下 `morphMap`：

```
use Illuminate\Database\Eloquent\Relations\Relation;

Relation::morphMap([
    'user' => User::class,
]);
```

Eloquent save 方法

如果模型在上一次获取或保存之后被改变过，调用 Eloquent 的 `save` 方法会返回 `false`。

Eloquent 作用域

Eloquent 作用域现在会以第一个作用域约束的布尔值为准，例如，如果你的作用域以 `orWhere` 开头，则将不再被转化为正常的 `where`。如果你现在的代码中使用了这个特性（在循环中添加了多个 `orWhere` 约束），需要验证第一个条件是否是正常的 `where` 以避免布尔逻辑问题。

如果你的作用域约束都是以 `where` 开头则不需要做任何调整，你可以通过 `toSql` 方法查看当前的 SQL 语句：

```
User::where('foo', 'bar')->toSql();
```

join 语句

`JoinClause` 类被重写以便统一查询构建器的语法，`on` 方法上可选的 `$where` 参数已被移除，要添加 `where` 条件需要显式使用查询构建器提供的 `where` 方法：

```
$query->join('table', function($join) {
    $join->on('foo', 'bar')->where('bar', 'baz');
});
```

`$bindings` 属性也被移除，要直接操作 `join` 绑定可以使用 `addBinding` 方法：

```
$query->join(DB::raw('.'.$subquery->toSql().' table'), function($join) use ($subquery) {
    $join->addBinding($subquery->getBindings(), 'join');
});
```

加密

Mcrypt 加密被移除

Laravel 5.1.0 版本中 Mcrypt 加密就已经被废弃，在 Laravel 5.3 中该功能被完全移除，以便于统一使用基于 OpenSSL 的新的加密实现，该实现从 Laravel 5.1.0 开始就已经作为默认的加密实现方式。

如果你还在配置文件 `config/app.php` 中使用基于 `cipher` 的 Mcrypt 加密，需要将 `cipher` 更新到

AES-256-CBC 并且将 key 设置为 32 位随机字符串（这可以通过 Artisan 命令 `php artisan key:generate` 生成）。

如果你在使用 Mcrypt 加密保存加密数据到数据库，则需要安装包含 Mcrypt 加密实现的扩展包 `laravel/legacy-encrypter`，你需要通过该扩展包解密数据然后通过新的 OpenSSL 加密方式对数据进行重新加密。例如，你可能在自定义 Artisan 命令中这么做：

```
$legacy = new McryptEncrypter($encryptionKey);

foreach ($records as $record) {
    $record->encrypted = encrypt(
        $legacy->decrypt($record->encrypted)
    );

    $record->save();
}
```

异常处理器

构造函数

异常处理器基类现在需要传递一个 `Illuminate\Container\Container` 实例到构造函数，这只有当你在 `app/Exception/Handler.php` 中定义了自定义的 `__construct` 方法时才会对应用产生影响，如果你这么做了，需要传递一个容器实例到 `parent::__construct` 方法：

```
parent::__construct(app());
```

中间件

can 中间件命名空间修改

罗列在 HTTP Kernel 的 `$routeMiddleware` 属性中的 `can` 中间件需要作如下修改：

```
'can' => \Illuminate\Auth\Middleware\Authorize::class,
```

can 中间件认证异常

如果用户没有认证的话 `can` 中间件会抛出 `Illuminate\Auth\AuthenticationException` 异常实例，

如果你手动捕获了其它异常类型，需要修改为捕获这个异常，在大多数案例中，这一修改对应用不会造成影响。

绑定替代中间件

路由模型绑定现在通过中间件来完成，所有应用都需要在 `app/Http/Kernel.php` 文件的 `web` 中间件组中添加 `Illuminate\Routing\Middleware\SubstituteBindings`：

```
\Illuminate\Routing\Middleware\SubstituteBindings::class,
```

你还需要在 HTTP Kernel 的 `$routeMiddleware` 属性中注册路由中间件用于绑定替代：

```
'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
```

这个路由中间件注册后，还要将其添加到 `api` 中间件组：

```
'api' => [
```

```
'throttle:60,1',  
'bindings',  
,
```

通知 (Notifications)

安装

Laravel 5.3 内置了一个新的、基于驱动的通知系统，你需要在配置文件 `config/app.php` 的 `providers` 数组中注册服务提供者 `Illuminate\Notifications\NotificationServiceProvider`。

你还需要在配置文件 `config/app.php` 的 `aliases` 数组中注册门面 `Illuminate\Support\Facades\Notification`。

最后，你可以在 `User` 模型或其它你希望接收通知的模型中使用 `Illuminate\Notifications\Notifiable` trait。

分页

自定义

与之前版本的 Laravel 相比，Laravel 5.3 中自定义分页生成的 HTML 变得更加简单，你只需要定义一个简单的 Blade 模板，而不需要定义一个“Presenter”类。自定义分页视图最简单的方式就是通过 `vendor:publish` 命令将它们导出到 `resources/views/vendor` 目录：

```
php artisan vendor:publish --tag=laravel-pagination
```

这个命令会将视图放置到 `resources/views/vendor/pagination` 目录，该目录下的 `default.blade.php` 文件会对应到默认的分页视图。只需要编辑这个文件就可以自定义分页 HTML。

阅读[完整的分页文档](#)了解更多实现细节。

队列

配置

在队列配置中，所有配置项 `expire` 需要重命名为 `retry_after`，类似的，Beanstalk 配置的配置项 `ttr` 也需要重命名为 `retry_after`。这一命名更改让配置项的意义更加明确。

闭包

队列闭包不再支持，如果你在应用中将闭包添加到队列，需要将闭包转化为一个类，然后将类实例添加到队列。

集合序列化

`Illuminate\Queue\SerializesModels` trait 现在适当实例化了 `Illuminate\Database\Eloquent\Collection`。对绝大多数应用来说这不算重大更新，但是，如果你的应用绝对依赖于不是通过队列任务从数据库重新获取的集合，那么就要验证这一改动对应用没有负面影响。

后台 worker

在使用 Artisan 命令 `queue:work` 的时候不再需要指定 `--daemon` 选项，运行 `php artisan queue:work` 命令的时候自动判断在后台运行。如果你想要处理单个任务，可以在命令后加上 `--once` 选项：

```
// Start a daemon queue worker...
php artisan queue:work
```

```
// Process a single job...
php artisan queue:work --once
```

事件数据修改

多个队列任务事件如 `JobProcessing` 和 `JobProcessed` 将不再包含 `$data` 属性，你需要更新应用调用 `$event->job->payload()` 来获取对应数据。

失败任务表

如果你的应用有了 `failed_jobs` 表，需要添加 `exception` 字段到这张表，`exception` 字段应该是 `TEXT` 类型，用于保存导致队列任务失败的异常字符串。

在传统风格队列任务上序列化模型

通常，Laravel 的队列任务通过传递任务实例到 `Queue::push` 方法添加到队列，不过，一些应用会通过如下这种传统的方式添加任务到队列：

```
Queue::push('ClassName@method');
```

如果你在使用这种语法，Eloquent 模型将不再会被自动序列化然后通过队列重新获取，如果你想让 Eloquent 模型继续被队列自动序列化，需要在任务类上使用 `Illuminate\Queue\SerializesModels` trait 并使用新的方法将任务推送到队列：

```
Queue::push(new ClassName);
```

路由

资源路由参数默认是单数

之前版本的 Laravel 中，使用 `Route::resource` 注册的路由参数并没有“单数化”，这可能会在注册路由模型绑定的时候引起一些非预期的行为，例如，给定如下 `Route::resource` 调用：

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

`show` 路由的 URI 将会定义如下：

```
/photos/{photos}
```

在 Laravel 5.3 中，所有资源路由参数默认都是单数化的，因此，同样调用 `Route::resource`，将会注册 URI 如下：

```
/photos/{photo}
```

如果你想要继续维护之前版本的行为而不是自动单数化资源路由参数，可以在

`AppServiceProvider` 中这样调用 `singularResourceParameters` 方法：

```
use Illuminate\Support\Facades\Route;  
  
Route::singularResourceParameters(false);
```

资源路由名称不再受路由前缀影响

使用 `Route::resource` 的时候 URI 前缀将不再影响分配给路由的路由名称 ,如果在 `Route::group`

中使用 `Route::resource` 时调用了指定的 `prefix` 选项 , 则需要检查所有对 `route` 辅助函数的调用以验证不再追加 URI 的 `prefix` 到路由名称。

如果这一改动导致同一名称下有两个路由 , 可以在调用 `Route::resource` 的时候使用 `names` 选项为给定路由指定一个自定义路由 , 更多信息请查看完整的路由文档。

验证

表单请求异常

如 果 一 个 表 单 请 求 验 证 失 败 , Lar avel 现 在 会 抛 出 一 个 `Illuminate\Validation\ValidationException` 实例而不是 `HttpException` 实例 , 如果你手动捕获了表单请求的 `HttpException` 实例 , 需要修改 `catch` 区块代码为捕获 `ValidationException`。

支持空的原生数值

当验证数组、布尔值、整型、数字、字符串时 , `null` 不会被当作有效值 , 除非在约束条件中设置包含 `nullable` :

```
Validate::make($request->all(), [
    'string' => 'nullable|max:5',
]);
```

1.3 贡献代码

1、缺陷报告

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

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

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

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

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

2、核心开发讨论

你可以在 [issue board](#) 上提议新功能或者优化已有功能，如果是新功能的话请至少实现部分代码以便完成新功能开发。

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

3、哪个分支？

所有的 bug 修复应该被提交到最新的稳定分支，永远不要把 bug 修复提交到 `master` 分支，除非它们能够修复下个发行版本中的特性。

当前版本中完全向后兼容的次要特性也可以提交到最新的稳定分支。

重要的新特性总是要被提交到 `master` 分支，包括下个发行版本。

如果你不确定是重要特性还是次要特性，请在 Slack 小组 [LaraChat](#) 的 `#internals` 频道咨询 Taylor Otwell

4、安全漏洞

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

5、编码风格

Laravel 遵循 [PSR-2](#) 编码标准和 [PSR-4](#) 自动载入标准。

PHPDoc

下面是一个有效的 Laravel [文档](#) 区块示例，注意到 `@param` 属性前面有两个空格，参数类型前有两个空格，最后是参数名称，也有两个空格：

```
/**  
 * Register a binding with the container.  
 */
```

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

StyleCI

如果你的代码格式不是很完美，不必担心，[StyleCI](#)会在提交代码时自动为我们修正代码风格以保持和 Laravel 仓库代码一致，从而让我们更加专注于代码内容而非风格。

2. 起步

2.1 安装

1、服务器要求

[Laravel](#) 框架有对服务器有少量要求，当然，Laravel Homestead 已经满足所有这些要求，所以我们强烈推荐使用 Homestead 作为 Laravel 本地开发环境（Mac 的话还可以使用 Valet 作为本地开发环境）。

不过，如果你没有使用 Homestead，那么需要保证开发环境满足以下要求：

- PHP 版本 $\geq 5.6.4$
- PHP 扩展：OpenSSL
- PHP 扩展：PDO
- PHP 扩展：Mbstring
- PHP 扩展：Tokenizer

2、安装 Laravel

Laravel 使用 [Composer](#) 管理依赖，因此，使用 Laravel 之前，确保机器上已经安装了 Composer。

通过 Laravel 安装器

首先，通过 Composer 安装 Laravel 安装器：

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

确保 `~/.composer/vendor/bin` 在系统路径中，否则不能在任意路径调用 `laravel` 命令。

安装完成后，通过简单的 `laravel new` 命令即可在当前目录下创建一个新的 Laravel 应用，例如，

`laravel new blog` 将会创建一个名为 `blog` 的新应用，且包含所有 Laravel 依赖。该安装方法比

通过 Composer 安装要快很多：

```
laravel new blog
```

通过 Composer Create-Project

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

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

注 如果要下载其他版本，比如 5.2 版本，可以使用这个命令：`composer create-project --prefer-dist laravel/laravel blog 5.2.*`。

3、配置

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

Public 目录

安装完 Laravel 后，需要将 HTTP 服务器的 web 根目录指向 `public` 目录，该目录下的 `index.php` 文件将作为前端控制器，所有 HTTP 请求都会通过该文件进入应用。

配置文件

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

目录权限

安装完 Laravel 后，需要配置一些目录的读写权限：`storage` 和 `bootstrap/cache` 目录应该是可

写的，如果你使用 Homestead 虚拟机做为开发环境，这些权限已经设置好了。

应用 Key

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

通常，该字符串应该是 32 位长，通过 `.env` 文件中的 APP_KEY 进行配置，如果你还没有将 `.env.example` 文件重命名为 `.env`，现在立即这样做。如果应用 `key` 没有被设置，用户 Session 和其它加密数据将会有安全隐患。

更多配置

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

你可能还想要配置 Laravel 的一些其它组件，比如缓存、数据库、Session 等，关于这些我们将会在后续[文档](#)——探讨。

安装完成后，即可进入下一步——配置 Laravel。

2.2 配置

1、简介

Laravel 的所有配置文件都存放在 `config` 目录下，每个配置项都有注释，以保证浏览任意配置文

件的配置项都能直观了解该配置项的作用及用法。

2、访问配置值

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

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

如果要在运行时设置配置值，传递数组参数到 `config` 方法即可：

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

3、环境配置

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

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

获取环境变量配置值

在应用每次接受请求时，`.env` 中列出的所有配置及其值都会被载入到 PHP 超全局变量 `$_ENV` 中，然后你就可以在应用中通过辅助函数 `env` 来获取这些配置值。实际上，如果你去查看 Laravel 的

配置文件，就会发现很多地方已经在使用这个辅助函数了：

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

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

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

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

判断当前应用环境

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

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

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

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

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

应用实例也可以通过辅助函数 `app` 来访问：

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

4、配置缓存

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

应用一旦上线，就要运行一次 `php artisan config:cache`，但是在本地开发时，没必要经常运行该命令，因为配置值经常需要改变。

5、维护模式

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

要开启维护模式，只需执行 Artisan 命令 `down` 即可：

```
php artisan down
```

要关闭维护模式，对应的 Artisan 命令是 `up`：

```
php artisan up
```

维护模式响应模板

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

维护模式 & 队列

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

维护模式的替代方案

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

2.3 目录结构

1、简介

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

Models 目录在哪里？

许多初学者都会困惑 Laravel 为什么没有 `models` 目录，我可以负责任的告诉大家，这是故意的。

因为 `models` 这个词对不同人而言有不同的含义，容易造成歧义，有些开发者认为应用的模型指

的是业务逻辑，另外一些人则认为模型指的是与关联数据库的交互。

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

2、根目录

App 目录

`app` 目录包含了应用的核心代码，此外你为应用编写的代码绝大多数也会放到这里；

Bootstrap 目录

`bootstrap` 目录包含了少许文件，用于框架的启动和自动载入配置，还有一个 `cache` 文件夹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` 目录包含了应用所有的配置文件，建议通读一遍这些配置文件以便熟悉所有配置项；

Database 目录

`database` 目录包含了数据迁移及填充文件，如果你喜欢的话还可以将其作为 SQLite 数据库存放目录；

Public 目录

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

Resources 目录

`resources` 目录包含了视图文件及原生资源文件（LESS、SASS、CoffeeScript），以及本地化文件；

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` 文件包含的路由都会应用 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` 文件包含的路由应用了 `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 token 进行认证并且不能访问 Session 状态。

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

Storage 目录

`storage` 目录包含了编译过的 Blade 模板、基于文件的 session、文件缓存，以及其它由框架生成的文件，该目录被细分为成 `app`、`framework` 和 `logs` 子母录，`app` 目录用于存放应用要使用的文件，`framework` 目录用于存放框架生成的文件和缓存，最后，`logs` 目录包含应用的日志文件；
`storage/app/public` 目录用于存储用户生成的文件，比如可以被公开访问的用户头像，要达到被访问的目的，你还需要在 `public` 目录下生成一个软连接 `storage` 指向这个目录。你可以通过 `php artisan storage:link` 命令生成这个软链接。

Tests 目录

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

Vendor 目录

`vendor` 目录包含 [Composer](#) 依赖。

3、App 目录

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

`app` 目录下包含多个子目录，如 `Console`、`Http`、`Providers` 等。`Console` 和 `Http` 目录提供了进入应用核心的 API，HTTP 协议和 CLI 是和应用进行交互的两种机制，但实际上并不包含应用逻辑。换句话说，它们只是两个向应用发布命令的方式。`Console` 目录包含了所有的 Artisan 命令，`Http` 目录包含了控制器、中间件和请求等。

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` 目录直到你执行 `make:job` 命令生成任务类时才会出现在 `app` 目录下。

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

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` 核心类，在这里可以注册自定义的 Artisan 命令以及定义调度任务。

Events 目录

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

Exceptions 目录

`Exceptions` 目录包含应用的异常处理器，同时还是处理应用抛出的任何异常的好地方。如果你想自定义异常如何记录异常或渲染，需要修改 `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

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

Listeners 目录

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

Mail 目录

这个目录默认不存在，但是可以通过执行 `make:mail` 命令生成，`Mail` 目录包含邮件发送类，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

这个目录默认不存在，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

这个目录默认不存在，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` 目录包含应用的所有服务提供者。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以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。

2.4 错误&日志

1、简介

Laravel 默认已经为我们配置好了错误和异常处理，我们在 `App\Exceptions\Handler` 类中触发异常并将响应返回给用户。本教程我们将深入探讨这个类。

此外，Laravel 还集成了 [Monolog 日志](#) 库以便提供各种功能强大的日志处理器。默认情况下，Laravel 已经为我们配置了一些处理器，我们可以选择单个日志文件，也可以选择记录错误信息到系统日志。

2、配置

错误详情显示

配置文件 `config/app.php` 中的 `debug` 配置项控制浏览器显示的错误详情数量。默认情况下，该配置项通过 `.env` 文件中的环境变量 `APP_DEBUG` 进行设置。

对本地开发而言，你应该设置环境变量 `APP_DEBUG` 值为 `true`。在生产环境，该值应该被设置为 `false`。如果在生产环境被设置为 `true`，就有可能将一些敏感的配置值暴露给终端用户。

日志存储

默认情况下，Laravel 支持日志方法 `single`, `daily`, `syslog` 和 `errorlog`。如果你想要日志文件按日生成而不是生成单个文件，应该在配置文件 `config/app.php` 中设置 `log` 值如下：

```
'log' => 'daily'
```

日志文件最大生命周期

使用 `daily` 日志模式的时候，Laravel 默认最多为我们保留最近 5 天的日志，如果你想要修改这个时间，需要添加一个配置 `log_max_files` 到 `app` 配置文件：

```
'log_max_files' => 30
```

日志错误级别

使用 `Monolog` 的时候，日志消息可能有不同的错误级别，默认情况下，Laravel 将所有日志写到

`storage` 目录，但是在生产环境中，你可能想要配置最低错误级别，这可以通过在配置文件 `app.php` 中通过添加配置项 `log_level` 来实现。

该配置项被配置后，Laravel 会记录所有错误级别大于等于这个指定级别的日志，例如，默认 `log_level` 是 `error`，则将会记录 `error`、`critical`、`alert` 以及 `emergency` 级别的日志信息：

```
'log_level' => env('APP_LOG_LEVEL', 'error'),
```

注：Monolog 支持以下错误级别 ——`debug`、`info`、`notice`、`warning`、`error`、`critical`、`alert`、`emergency`。

自定义 Monolog 配置

如果你想要在应用中完全控制 Monolog 的配置，可以使用 `configureMonologUsing` 方法。你需要在 `bootstrap/app.php` 文件返回 `$app` 变量之前调用该方法：

```
$app->configureMonologUsing(function($monolog) {
    $monolog->pushHandler(...);
});

return $app;
```

3、异常处理器

所有异常都由类 `App\Exceptions\Handler` 处理，该类包含两个方法：`report` 和 `render`。下面我们详细阐述这两个方法。

report 方法

`report` 方法用于记录异常并将其发送给外部服务如 `Bugsnag` 或 `Sentry`，默认情况下，`report` 方

法只是将异常传递给异常被记录的基类，当然你也可以按自己的需要记录异常并进行相关处理。

例如，如果你需要以不同方式报告不同类型的异常，可使用 PHP 的 `instanceof` 比较操作符：

```
/*
 * 报告或记录异常
 *
 * This is a great spot to send exceptions to Sentry, Bugsnag, etc.
 *
 * @param \Exception $e
 * @return void
 */
public function report(Exception $e){
    if ($e instanceof CustomException) {
        //
    }

    return parent::report($e);
}
```

通过类型忽略异常

异常处理器的 `$dontReport` 属性包含一个不会被记录的异常类型数组，默认情况下，`404` 错误异常

不会被写到日志文件，如果需要的话你可以添加其他异常类型到这个数组：

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

```
* @var array
*/
protected $dontReport = [
    \Illuminate\Auth\AuthenticationException::class,
    \Illuminate\Auth\Access\AuthorizationException::class,
    \Symfony\Component\HttpKernel\Exception\HttpException::class,
    \Illuminate\Database\Eloquent\ModelNotFoundException::class,
    \Illuminate\Validation\ValidationException::class,
];
];
```

render 方法

`render` 方法负责将给定异常转化为发送给浏览器的 HTTP 响应，默认情况下，异常被传递给为你生成响应的基类。当然，你也可以按照自己的需要检查异常类型或者返回自定义响应：

```
/**
 * 将异常渲染到 HTTP 响应中
 *
 * @param \Illuminate\Http\Request $request
 * @param \Exception $e
 * @return \Illuminate\Http\Response
 */
public function render($request, Exception $e){
    if ($e instanceof CustomException) {
        return response()->view('errors.custom', [], 500);
    }
}
```

```
    return parent::render($request, $e);  
}
```

4、HTTP 异常

有些异常描述来自服务器的 HTTP 错误码，例如，这可能是一个“页面未找到”错误（`404`），“认证失败错误”（`401`）亦或是程序出错造成的 `500` 错误，为了在应用中生成这样的响应，可以使用 `abort` 方法：

```
abort(404);
```

`abort` 方法会立即引发一个会被异常处理器渲染的异常，此外，你还可以像这样提供响应描述：

```
abort(403, 'Unauthorized action.');
```

该方法可在请求生命周期的任何时间点使用。

自定义 HTTP 错误页面

在 Laravel 中，返回不同 HTTP 状态码的错误页面很简单，例如，如果你想要自定义 `404` 错误页面，创建一个 `resources/views/errors/404.blade.php` 文件，该视图文件用于渲染程序返回的所有 `404` 错误。需要注意的是，该目录下的视图命名应该和相应的 HTTP 状态码相匹配。

5、日志

Laravel 基于强大的 Monolog 库提供了简单的日志抽象层，默认情况下，Laravel 被配置为在

`storage/logs` 目录下每天为应用生成日志文件，你可以使用 `Log` 门面记录日志信息到日志中：

```
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller{
    /**
     * 显示指定用户的属性
     *
     * @param int $id
     * @return Response
     */
    public function showProfile($id)
    {
        Log::info('Showing user profile for user: '.$id);
        return view('user.profile', ['user' => User::findOrFail($i
d)]);
    }
}
```

该日志记录器提供了 [RFC 5424](#) 中定义的八种日志级别：`emergency`、`alert`、`critical`、`error`、`warning`、`notice`、`info` 和 `debug`。

```
Log::emergency($error);
Log::alert($error);
Log::critical($error);
Log::error($error);
Log::warning($error);
Log::notice($error);
Log::info($error);
```

```
Log::debug($error);
```

上下文信息

上下文数据也会以数组形式传递给日志方法，然后和日志消息一起被格式化和显示：

```
Log::info('User failed to login.', ['id' => $user->id]);
```

访问底层 Monolog 实例

Monolog 有多个可用于日志的处理器，如果需要的话，你可以访问 Laravel 使用的底层 Monolog

实例：

```
$monolog = Log::getMonolog();
```

3. 开发环境

3.1 Laravel Homestead

1、简介

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

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

数分钟内销毁并重新创建该 Vagrant 盒子！

Homestead 可以运行在 Windows、Mac 以及 Linux 系统上 ,其中已经安装好了 [Nginx](#)、PHP7.0、

MySQL、Postgres、Redis、Memcached、Node 以及很多其它开发 Laravel 应用所需要的东西。

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

预装软件

- Ubuntu 16.04
- Git
- PHP 7.0
- HHVM
- Xdebug
- Nginx
- MySQL
- MariaDB
- SQLite 3
- Postgres
- Composer
- Node (With PM2, Bower, Grunt, and Gulp)
- Redis
- Memcached
- Beanstalkd

2、安装 & 设置

首次安装

在使用 Homestead 之前 ,需要先安装 Virtual Box 5.x/VMWare 和 Vagrant ,所有这些软件包都为常用操作系统提供了一个便于使用的可视化安装器。

安装 Homestead Vagrant 盒子

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

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

如果上述命令执行失败，需要确认 Vagrant 是否是最新版本。

通过 GitHub 安装 Homestead

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

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

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

```
bash init.sh
```

配置 Homestead

设置 Provider

`Homestead.yaml` 文件中的 `provider` 键表示使用哪个 Vagrant 提供者：`virtualbox`、

`vmware_fusion` 或者 `vmware_workstation`，你可以将其设置为自己喜欢的提供者：

```
provider: virtualbox
```

配置共享文件夹

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

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

如果要开启 [NFS](#)，只需简单添加一个标识到同步文件夹配置：

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

配置 Nginx 站点

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

站点：

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

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

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

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

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

Hosts 文件

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

```
192.168.10.10 homestead.app
```

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

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

启动 Vagrant Box

配置好 `Homestead.yaml` 文件后，在 Homestead 目录下运行 `vagrant up` 命令，Vagrant 将会启动虚拟机并自动配置共享文件夹以及 Nginx 站点。

销毁该机器，可以使用 `vagrant destroy -force` 命令。

为指定项目安装 Homestead

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

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

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

Mac/Linux：

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

Windows:

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

接下来，在终端中运行 `vagrant up` 命令然后在浏览器中通过 `http://homestead.app` 访问站点。

不要忘记在 `/etc/hosts` 文件中添加域名 `homestead.app`。

安装 MariaDB

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

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

3、日常使用

全局访问 Homestead

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

通过 SSH 连接

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

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

连接到数据库

Homestead 默认已经在虚拟机中为 MySQL 和 Postgres 数据库做好了配置，更方便的是，这些配置值就是 Laravel 的 `.env` 中默认提供的。

想要通过本地的 Navicat 或 Sequel Pro 连接到 Homestead 上的 MySQL 或 Postgres 数据库，可以通过新建连接来实现，主机 IP 都是 `127.0.0.1`，对于 MySQL 而言，端口号是 `33060`，对 Postgres 而言，端口号是 `54320`，用户名/密码是 `homestead/secret`。

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

添加更多站点

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

配置 Cron 调度任务

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

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

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

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

端口转发配置

默认情况下，Homestead 端口转发配置如下：

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

转发更多端口

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

```
ports:
```

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

4、网络接口

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

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

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

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

要开启 `DHCP`，只需要从配置中移除 `ip` 选项即可：

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

3.2 Laravel Valet

1、简介

Valet 是为 Mac 提供的极简主义[开发环境](#)，没有 Vagrant、Apache、Nginx，也没有`/etc/hosts`文件，甚至可以使用本地隧道公开共享你的站点。

在你启动 Mac 后，[Laravel](#) Valet 会在后台静默运行 [Caddy](#)，然后通过使用 [DnsMasq](#)，Valet 将所有请求代理到`*.dev` 域名并指向本地安装的站点目录。这样一个极速的 Laravel 开发环境只需要占用 7M 内存。

Valet 并不是想要替代 Vagrant 或者 Homestead，只是提供了另外一种选择，更加灵活、极速、以及占用更小的内存空间。

Valet 为我们支持但不限于以下软件和工具：

- Laravel
- Lumen
- [Symfony](#)
- [Zend](#)
- [CakePHP 3](#)
- [WordPress](#)
- [Bedrock](#)
- [Craft](#)
- [Statamic](#)
- [Jigsaw](#)
- 静态 HTML

当然，你还可以通过自定义驱动扩展 Valet。

Valet Or Homestead

正如你所知道的，Laravel 还提供了另外一个开发环境 [Homestead](#)。Homestead 和 Valet 的不同之

处在于两者的目标受众和本地开发方式。Homestead 提供了一个完整的包含自动化 Nginx 配置的 Ubuntu 虚拟机，如果你需要一个完整的虚拟化 Linux 开发环境或者使用的是 Windows/Linux 操作系统，那么 Homestead 无疑是最佳选择。

Valet 只支持 Mac，并且要求本地安装了 PHP 和数据库服务器，这可以通过使用 [Homebrew](#) 命令轻松实现（`brew install php70` 以及 `brew install mariadb`），Valet 通过最小的资源消耗提供了一个极速的本地开发环境，如果你只需要 PHP/MySQL 并且不需要完整的虚拟化开发环境，那么 Valet 将是最好的选择。

最后，Valet 和 Homestead 都是配置本地 Laravel 开发环境的好帮手，选择使用哪一个取决于你个人的喜好或团队的需求。

2、安装

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

- 使用 `brew update` 安装或更新 Homebrew 到最新版本
- 通过运行 `brew services list` 确保 `brew services` 有效并且能获取到正确的输出，如果无效，则需要[添加](#)。
- 通过 Homebrew 安装 PHP 7.0：`brew install homebrew/php/php70`。
- 通过 Composer 安装 Valet：`composer global require laravel/valet`（确保 `~/.composer/vendor/bin` 在系统路径中）

- 运行 `valet install` 命令，这将会配置并安装 Valet 和 DnsMasq，然后注册 Valet 后台随机启动。

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

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

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

使用其他域名

默认情况下，Valet 使用.dev 域名后缀，如果你想要使用其他域名，可以使用 `valet domain tld-name` 命令。例如，你想使用.app 域名后缀取代.dev，运行 `valet domain app`，Valet 将会自动将站点域名后缀改为*.app。

数据库

如果你需要数据库，可以在命令行通过 `brew install mariadb` 安装 MariaDB，安装完成后就可以在本机通过用户名 `root` 和一个空密码连接到数据库。

3、服务站点

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

park 命令

- 在 Mac 中创建一个新目录，例如 `mkdir ~/Sites`，然后进入这个目录并运行 `valet park`。这个命令会将当前所在目录作为 web 根目录。
- 接下来，在新建的目录中创建一个新的 Laravel 站点：`laravel new blog`。
- 在浏览器中访问 `http://blog.dev`。

页面截图如下：



Laravel 5

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

link 命令

`link` 命令也可以用于本地 Laravel 站点，当你想要在目录中提供单个站点时 这个命令 很有用。

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

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

通过 TLS 让站点更安全

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

```
valet secure laravel
```

要回到原生 HTTP，使用 `unsecure` 命令。和 `secure` 命令一样，该命令接收主机名作为参数：

```
valet unsecure laravel
```

4、分享站点

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

要共享站点，切换到站点所在目录并运行 `valet share`，这会生成一个可以公开访问的 URL 并插

入剪贴板，以便你直接复制到浏览器地址栏，就是这么简单。

要停止共享站点，使用 `Control + C` 即可。

5、查看日志

如果你想要在终端显示所有站点的日志，可以运行 `valet logs` 命令，这会在终端显示新的日志。

6、自定义 Valet 驱动

你还可以编写自定义的 Valet 驱动为运行于非 Valet 原生支持的 PHP 应用提供服务。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，这个文件中有一个演示如何编写自定义驱动的示例。编写一个驱动只需要实现三个方法：`serves`、`isStaticFile` 和 `frontControllerPath`。

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

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

下面我们来具体讨论并演示自定义 Valet 驱动需要实现的三个方法。

`serves` 方法

如果自定义驱动需要继续处理输入请求，`serves` 方法会返回 `true`，否则该方法返回 `false`。因此，

在这个方法中应该判断给定的 `$sitePath` 是否包含你服务类型的项目。

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

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

isStaticFile 方法

`isStaticFile` 方法会判断输入请求是否是静态文件，例如图片或样式文件，如果文件是静态的，

该方法会返回磁盘上的完整路径，如果输入请求不是请求静态文件，则返回 `false`：

```
/*
 * Determine if the incoming request is for a static file.
 *
 * @param string $sitePath
```

```
* @param string $siteName
* @param string $uri
* @return string|false
*/
public function isStaticFile($sitePath, $siteName, $uri)
{
    if (file_exists($staticFilePath = $sitePath . '/public/' . $uri)) {
        return $staticFilePath;
    }

    return false;
}
```

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

frontControllerPath 方法

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

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

```
{  
    return $sitePath.'/public/index.php';  
}
```

7、其他 Valet 命令

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

4. 核心概念

4.1 服务容器

1、简介

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

让我们看一个简单的例子：

```
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller
{
    /**
     * The user repository implementation.
     *
     * @var UserRepository
     */
}
```

```
protected $users;

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

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

在本例中，`UserController` 需要从数据源获取用户，所以，我们注入了一个可以获取用户的服
务 `UserRepository`，其扮演的角色类似使用 Eloquent 从数据库获取用户信息。注

入 `UserRepository` 后，我们可以在其基础上封装其他实现，也可以模拟或者创建一个假的 `UserRepository` 实现用于测试。

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

2、绑定

绑定基础

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

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

简单的绑定

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

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

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

对象的子依赖。

绑定一个单例

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

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

绑定实例

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

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

绑定原始值

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

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

绑定接口到实现

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

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

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

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

上下文绑定

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

简单、平滑的接口：

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

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

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

标签

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

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

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

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

这些服务被打上标签后，可以通过 `tagged` 方法来轻松解析它们：

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

3、解析

make 方法

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

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

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

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

自动注入

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

象从容器中解析的方式。

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

```
<?php

namespace App\Http\Controllers;

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

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

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

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

```
}
```

4、容器事件

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

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

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

4.2 服务提供者

1、简介

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

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

如果你打开 Laravel 自带的 `config/app.php` 文件，将会看到一个 `providers` 数组，这里就是应用

所要加载的所有服务提供者类，当然，其中很多是延迟加载的，也就是说不是每次请求都会被加载，只有真的用到它们的时候才会加载。

本章节里你将会学习如何编写自己的服务提供者并在 Laravel 应用中注册它们。

2、编写服务提供者

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

通过 Artisan 命令 `make:provider` 可以简单生成一个新的提供者：

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

register 方法

正如前面所提到的，在 `register` 方法中只绑定事物到服务容器，而不要做其他事情，否则，一不小心就能用到一个尚未被加载的服务提供者提供的服务。

现在让我们来看看一个基本的服务提供者长什么样：

```
<?php

namespace App\Providers;

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

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

```

```
* @return void
*/
public function register()
{
    $this->app->singleton('Riak\Contracts\Connection', function
($app) {
    return new Connection(config('riak'));
});
}
```

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

boot 方法

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

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

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

```
}

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

boot 方法的依赖注入

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

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

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

3、注册服务提供者

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

要注册你自己的服务提供者，只需要将其追加到该数组中即可：

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

4、延迟加载服务提供者

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

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

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

```
<?php

namespace App\Providers;

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

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

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

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

}
```

4.3 门面 (Facades)

1、简介

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

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

到门面：

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

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

在整个 Laravel [文档](#) 中，很多例子使用了门面来演示框架的各种功能特性

2、何时使用门面

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

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

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

门面 vs 依赖注入

依赖注入的最大优点是可以替换注入类的实现，这在测试时很有用，因为你可以注入一个模拟或

存根并且在存根上断言不同的方法。

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

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

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

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

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

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

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

门面 vs 辅助函数

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

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

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

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

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

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

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

3、门面工作原理

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

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

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

```
<?php
```

```
namespace App\Http\Controllers;

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

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

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

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

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

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

`Cache` 门面继承 `Facade` 基类并定义了 `getFacadeAccessor` 方法，该方法的工作就是返回服务容器

绑定类的别名 ,当用户引用 `Cache` 类的任何静态方法时 ,Laravel 从服务容器中解析 `cache` 绑定 ,然后在解析出的对象上调用所有请求方法 (本例中是 `get`)。

4、门面类列表

下面列出了每个门面及其对应的底层类 ,这对深入给定根门面的 API 文档而言是个很有用的工具。

服务容器绑定键也被包含进来 :

门面	类	服务容器绑定
App	Illuminate\Foundation\Application	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	encrypter
DB	Illuminate\Database\DatabaseManager	db
DB (Instance)	Illuminate\Database\Connection	
Event	Illuminate\Events\Dispatcher	events
File	Illuminate\Filesystem\Filesystem	files

门面	类	服务容器绑定
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
Password	Illuminate\Auth\Passwords\PasswordBroker	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

门面	类	服务容器绑定
Validator	Illuminate\Validation\Factory	<code>validator</code>
Validator (Instance)	Illuminate\Validation\Validator	
View	Illuminate\View\Factory	<code>view</code>
View (Instance)	Illuminate\View\View	

4.4 契约 (Contracts)

1、简介

Laravel 中的 契约 是指框架提供的一系列定义核心服务的 接口。例如，

`Illuminate\Contracts\Queue\Queue` 契约定义了队列任务需要实现的方法，

`Illuminate\Contracts\Mail\Mailer` 契约定义了发送邮件所需要实现的方法。

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

由 `SwiftMailer` 驱动 实现。

所有 Laravel 契约都有其对应的 [GitHub 库](#)，这为所有有效的契约提供了快速入门指南，同时也可

以作为独立、解耦的包被开发者使用。

契约 VS 门面

Laravel 门面 为 Laravel 服务的使用提供了便捷方式——不再需要从 服务容器 中类型提示和契约

解析即可直接通过静态门面调用。有些开发者喜欢这种便捷，不过也有开发者倾向于使用契约，

他们喜欢定义明确的依赖。

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

2、何时使用契约

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

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

松耦合

首先，让我们看看一些缓存实现的紧耦合代码：

```
<?php  
  
namespace App\Orders;  
  
class Repository{  
    /**  
     * 缓存  
     */  
    protected $cache;
```

```
/*
 * 创建一个新的 Repository 实例
 *
 * @param \SomePackage\Cache\Memcached $cache
 * @return void
 */

public function __construct(\SomePackage\Cache\Memcached $cache)
{
    $this->cache = $cache;
}

/**
 * 通过 ID 获取订单
 *
 * @param int $id
 * @return Order
 */

public function find($id)
{
    if ($this->cache->has($id)) {
        //
    }
}

}
```

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

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

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

```
<?php

namespace App\Orders;

use Illuminate\Contracts\Cache\Repository as Cache;

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

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

简单

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

此外，基于简单接口，代码也更容易理解和维护。在一个庞大而复杂的类中，与其追踪哪些方法

是有效的，不如转向简单、干净的接口。

3、如何使用契约

那么，如何实现契约呢？这很简单。

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

例如，看看下面这个事件监听器：

```
<?php

namespace App\Listeners;

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

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

    /**
     * Create a new event handler instance.
    
```

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

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

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

4、契约列表

下面是 Laravel 契约列表，以及其对应的“门面”：

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

Contract	References Facade
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

5.1 路由

1、基本路由

最基本的 Laravel 路由只接收一个 URI 和一个闭包，并以此提供一个非常简单且优雅的定义路由

方法：

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

默认路由文件

所有 Laravel 路由都定义位于 `routes` 目录下的路由文件中，这些文件通过框架自动加载。

`routes/web.php` 文件定义了 `web` 界面的路由，这些路由被分配给 `web` 中间件组，从而可以提供

`session` 和 `csrf` 防护等功能。`routes/api.php` 中的路由是无状态的，被分配到 `api` 中间件组。

对大多数应用而言，都是从 `routes/web.php` 文件开始定义路由。

有效的路由方法

我们可以注册路由来响应任何 HTTP 请求：

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

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

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

```
Route::match(['get', 'post'], '/', function () {
    //
});
Route::any('foo', function () {
    //
});
```

CSRF 防护

在 `web` 路由文件中所有请求方式为 `PUT`、`POST` 或 `DELETE` 的 HTML 表单都会包含一个 CSRF 令牌字
段，否则，请求会被拒绝。关于 CSRF 的更多细节，可以参考其[文档](#)：

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

2、路由参数

必选参数

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

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

可以按需要在路由中定义多个路由参数：

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

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

注意：路由参数不能包含 `-` 字符，需要的话可以使用 `_` 替代。

可选参数

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

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

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

3、命名路由

命名路由为生成 URL 或重定向提供了便利。实现也很简单，在路由定义之后使用 `name` 方法链的方式来实现：

```
Route::get('user/profile', function () {  
    //  
})->name('profile');
```

还可以为控制器动作指定路由名称：

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

为命名路由生成 URL

为给定路由分配名称之后，就可以通过辅助函数 `route` 为该命名路由生成 URL：

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

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

会自动插入到 URL 中：

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

4、路由群组

路由群组允许我们在多个路由中共享路由属性，比如中间件和命名空间等，这样的话我们就不必为每一个路由单独定义属性。共享属性以数组的形式作为第一个参数被传递给 `Route::group` 方法。

中间件

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

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

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

```
});
```

命名空间

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

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

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

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

子域名路由

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

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

```
//  
});  
});
```

路由前缀

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

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

5、路由模型绑定

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

隐式绑定

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

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

在这个例子中，由于类型声明了 Eloquent 模型 `App\User`，对应的变量名 `$user` 会匹配路由片段中的 `{user}`，这样，Laravel 会自动注入与请求 URI 中传入的 ID 对应的用户模型实例。

如果数据库中找不到对应的模型实例，会自动生成 HTTP 404 响应。

自定义键名

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

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

显式绑定

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

绑定参数到模型

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

接下来，定义一个包含 `{user}` 参数的路由：

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

由于我们已经绑定 `{user}` 参数到 `App\User` 模型，User 实例会被注入到该路由。因此，如果请求

URL 是 `profile/1`，就会注入一个用户 ID 为 1 的 User 实例。

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

自定义解析逻辑

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

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

6、表单方法伪造

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

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

还可以使用辅助函数 `method_field` 来实现这一目的：

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

7、访问当前路由

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

```
$route = Route::current();  
$name = Route::currentRouteName();  
$action = Route::currentRouteAction();
```

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

5.2 中间件

1、简介

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

当然，除了认证之外，中间件还可以被用来处理更多其它任务。比如：CORS 中间件可以用于为离开站点的响应添加合适的头（跨域）；日志中间件可以记录所有进入站点的请求。

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

2、定义中间件

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

```
php artisan make:middleware CheckAge
```

这个命令会在 [app/Http/Middleware](#) 目录下创建一个新的中间件类 `CheckAge`，在这个中间件中，我们只允许提供的 `age` 大于 `200` 的请求访问[路由](#)，否则，我们将用户重定向到 `home`：

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

```
use Closure;

class CheckAge
{
    /**
     * 返回请求过滤器
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if ($request->input('age') <= 200) {
            return redirect('home');
        }

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

正如你所看到的，如果 `age<=200`，中间件会返回一个 HTTP 重定向到客户端；否则，请求会被传递下去。将请求往下传递可以通过调用回调函数 `$next` 并传入 `$request`。

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

中间件之前/之后

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

```
<?php

namespace App\Http\Middleware;

use Closure;

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

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

然而，下面这个中间件则会在请求处理后执行其任务：

```
<?php

namespace App\Http\Middleware;

use Closure;
```

```
class AfterMiddleware

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

        // 执行动作

        return $response;
    }
}
```

3、注册中间件

全局中间件

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

分配中间件到路由

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

```
// Within App\Http\Kernel Class...
```

```
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,
];
```

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

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

使用数组分配多个中间件到路由：

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

分配中间件的时候还可以传递完整的类名：

```
use App\Http\Middleware\CheckAge;
```

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

中间件组

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

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

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

```
'throttle:60,1',
'auth:api',
],
];
```

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

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

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

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

4、中间件参数

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

额外的中间件参数会在 `$next` 参数之后传入中间件：

```
<?php
```

```
namespace App\Http\Middleware;

use Closure;

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

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

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

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

5、可终止的中间件

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

```
<?php

namespace Illuminate\Session\Middleware;

use Closure;

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

    public function terminate($request, $response)
    {
```

```
// 存储 session 数据...
}

}
```

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

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

5.3 CSRF 保护

1、简介

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

Laravel 自动为每一个被应用管理的有效用户会话生成一个 [CSRF “令牌”](#)，该令牌用于验证授权用户和发起请求者是否是同一个人。

任何时候在 Laravel 应用中定义 HTML 表单，都需要在表单中引入 CSRF 令牌字段，这样 CSRF 保护中间件才能够正常验证请求。想要生成包含 CSRF 令牌的隐藏输入字段，可以使用辅助函数 `csrf_field` 来实现：

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

```
...  
</form>
```

中间件组 web 中的中间件 `VerifyCsrfToken` 会自动为我们验证请求输入的 `token` 值和 Session 中存储的 `token` 是否一致。

2、从 CSRF 保护中排除指定 URL

有时候我们需要从 CSRF 保护中排除一些 URL，例如，如果你使用了 Stripe 来处理支付并用到他们的 webhook 系统，这时候就需要从 Laravel 的 CSRF 保护中排除 webhook 处理器路由，因为 Stripe 并不知道要传什么 token 值给我们定义的路由。

通常我们需要将这种类型的路由放到文件 `routes/web.php` 里，中间件组 `web` 之外。此外，你也可以在 `VerifyCsrfToken` 中间件中将要排除的 URL 添加到 `$except` 属性数组：

```
<?php  
  
namespace App\Http\Middleware;  
  
use Illuminate\Foundation\Http\Middleware\VerifyCsrfToken as BaseVerifier;  
  
class VerifyCsrfToken extends BaseVerifier  
{  
    /**  
     * 从 CSRF 验证中排除的 URL  
     *  
     * @var array
```

```
*/  
protected $except = [  
    'stripe/*',  
];  
}
```

3、X-CSRF-Token

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

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

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

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

4、X-XSRF-Token

Laravel 还会将 CSRF 令牌保存到名为 `XSRF-TOKEN` 的 Cookie 中，你可以使用该 Cookie 值来设

置 `X-XSRF-TOKEN` 请求头。一些 JavaScript 框架，比如 Angular，会为你自动进行设置，基本上你不太需要手动设置这个值。

5.4 控制器

1、简介

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

2、基本控制器

定义控制器

下面是一个基本控制器类的例子。所有的 Laravel 控制器应该继承自 Laravel 自带的控制器基类 `Controller`，控制器基类提供了一些很方便的方法如 `middleware`，用于添加中间件到控制器动作：

```
<?php

namespace App\Http\Controllers;

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

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

我们可以像这样定义指向该控制器动作的路由：

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

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

注：控制器并不是一定要继承自基类，不过，那样的话就不能使用那些便利的方法了，比如 `middleware`、`validate` 和 `dispatch` 等。

控制器 & 命名空间

你应该注意到我们在定义控制器路由的时候没有指定完整的控制器命名空间，而只是定义了 `App\Http\Controllers` 之后的部分。默认情况下，`RouteServiceProvider` 将会在一个路由群组

中载入 `routes.php` 文件，并且该路由群组指定定了群组中路由控制器所在的命名空间。

如果你在 `App\Http\Controllers` 目录下选择使用 PHP 命名空间嵌套或组织控制器，只需要使用

相对于 `App\Http\Controllers` 命名空间的指定类名即可。因此，如果你的完整控制器类是

`App\Http\Controllers\Photos\AdminController`，你可以像这样注册路由：

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

单动作控制器

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

```
<?php

namespace App\Http\Controllers;

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

class ShowProfile extends Controller
{
    /**
     * Show the profile for the given user.
     *
     * @param int $id
     * @return Response
     */
}
```

```
public function __invoke($id)
{
    return view('user.profile', ['user' => User::findOrFail($id)]);
}
```

当你为这个单动作控制器注册路由的时候，不需要指定方法：

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

3、控制器中间件

中间件可以像这样分配给控制器路由：

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

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

```
class UserController extends Controller
{
    /**
     * 实例化一个新的 UserController 实例
     *
     * @return void
     */
    public function __construct()
```

```
{  
    $this->middleware('auth');  
  
    $this->middleware('log')->only('index');  
  
    $this->middleware('subscribed')->except('store');  
}  
}
```

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

4、资源控制器

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

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

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

接下来，可以为该控制器注册一个资源路由：

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

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

资源控制器处理的动作

方法	路径	动作	路由名称
GET	/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

伪造表单方法

由于 HTML 表单不能发起 `PUT`、`PATCH` 和 `DELETE` 请求，需要添加一个隐藏的 `_method` 字段来伪造

HTTP 请求方式，辅助函数 `method_field` 可以帮我们做这件事：

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

部分资源路由

声明资源路由时可以指定该路由处理的动作子集：

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

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

命名资源路由

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

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

命名资源路由参数

默认情况下，`Route::resource` 将会基于资源名称的单数格式为资源路由参数，你可以通过在选项数组中传递 `parameters` 来覆盖这一默认设置。`parameters` 应该是资源名称和参数名称的关联数组：

```
Route::resource('user', 'AdminUserController', ['parameters' => [
    'user' => 'admin_user'
]]);
```

上面的示例代码会为资源的 `show` 路由生成如下 URL：

```
/user/{admin_user}
```

补充资源控制器

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

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

注：注意保持控制器的单一职责，如果你发现指向控制器动作的路由超过默认提供的资源控制器动作集合了，考虑将你的控制器分割成更多、更小的控制器。

5、依赖注入 & 控制器

构造函数注入

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

```
<?php  
  
namespace App\Http\Controllers;  
  
use Illuminate\Routing\Controller;  
use App\Repositories\UserRepository;
```

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

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

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

方法注入

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

```
<?php
```

```
namespace App\Http\Controllers;

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

class UserController extends Controller

{
    /**
     * 存储新用户
     *
     * @param Request $request
     * @return Response
     */

    public function store(Request $request)
    {
        $name = $request->input('name');

        //
    }
}
```

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

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

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

```
<?php

namespace App\Http\Controllers;

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

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

6、路由缓存

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

由。

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

```
php artisan route:cache
```

运行完成后，每次请求都会从缓存中读取路由，记住，如果你添加新的路由需要重新生成路由缓存。因此，只有在项目部署阶段才需要运行 `route:cache` 命令。

想要移除缓存路由文件，使用 `route:clear` 命令即可：

```
php artisan route:clear
```

5.5 请求

1、访问请求实例

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

```
<?php  
  
namespace App\Http\Controllers;  
  
use Illuminate\Http\Request;
```

```
use Illuminate\Routing\Controller;

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

        //
    }
}
```

依赖注入 & 路由参数

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

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

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

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller

{
    /**
     * 更新指定用户
     *
     * @param Request $request
     * @param int $id
     * @return Response
     */

    public function update(Request $request,$id)
    {
        //
    }
}
```

通过路由闭包访问请求

还可以在路由闭包上类型提示 `Illuminate\Http\Request` 类，在执行的时候服务容器会自动注入 输入 的请求到闭包：

```
use Illuminate\Http\Request;
```

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

请求路径 & 方法

`Illuminate\Http\Request` 实例提供了多个方法来检测应用的 HTTP 请求，

Laravel 的 `Illuminate\Http\Request` 继承自 `Symfony\Component\HttpFoundation\Request` 类，

下面演示了该类提供的一些有用方法：

获取请求路径

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

则 `path` 方法将会返回 `foo/bar`：

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

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

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

获取请求 URL

想要获取完整的 URL，而不仅仅是路径信息，可以使用请求实例提供的 `url` 或 `fullUrl` 方

法，`url` 方法将会返回不带查询字符串的 URL，而 `fullUrl` 方法返回结果则包含查询字符串：

```
//不包含查询字符串  
$url=$request->url();  
  
//包含查询字符串  
$url = $request->fullUrl();
```

获取请求方法

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

```
$method=$request->method();  
if($request->isMethod('post')){  
    //  
}
```

PSR-7 请求

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

```
composer require symfony/psr-http-message-bridge  
composer require zendframework/zend-diactoros
```

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

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

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

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

2、获取请求输入

获取所有输入值

你可以使用 `all` 方法以数组格式获取所有输入值：

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

获取单个输入值

使用一些简单的方法，就可以从 `Illuminate\Http\Request` 实例中访问用户输入。你不需要关心请求所使用的 HTTP 请求方法，因为对所有请求方式的输入都是通过 `input` 方法获取用户输入：

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

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

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

处理表单数组输入时，可以使用“.”来访问数组输入：

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

通过动态属性获取输入

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

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

使用动态属性的时候，Laravel 首先会在请求中查找参数的值，如果不存在，还会到路由参数中查找。

获取 JSON 输入值

发送 JSON 请求到应用的时候，只要 `Content-Type` 请求头被设置为 `application/json`，都可以通过 `input` 方法获取 JSON 数据，还可以通过“.”号解析数组：

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

获取输入的部分数据

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

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

判断输入值是否出现

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

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

上一次请求输入

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

将输入存储到一次性 Session

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

依然有效：

```
$request->flash();
```

你还可以使用 `flashOnly` 和 `flashExcept` 方法将输入数据子集存放到 Session 中，这些方法在 session 之外保存敏感信息时很有用：

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

将输入存储到一次性 Session 然后重定向

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

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

取出上次请求数据

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

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

Laravel 还提供了一个全局的辅助函数 `old`，如果你是在 Blade 模板中显示上次输入数据，使用

辅助函数 `old` 更方便，如果给定参数没有对应输入，返回 `null`：

```
<input type="text" name="username" value="{{ old('username') }}">
```

Cookies

从请求中取出 Cookies

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

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

新增 Cookie 到响应

你可以使用 `cookie` 方法将一个 `cookie` 附加到输出的 `Illuminate\Http\Response` 实例，你需要传递名称、值、以及 `cookie` 有效期（分钟）到这个方法：

```
return response('Hello World')->cookie(
    'name', 'value', $minutes
);
```

`cookie` 方法可以接收一些使用频率较低的参数，一般来说，这些参数和 PHP 原生函数 `setcookie` 作用和意义一致：

```
return response('Hello World')->cookie(  
    'name', 'value', $minutes, $path, $domain, $secure, $httpOnly  
) ;
```

生成 Cookie 实例

如果你想要生成一个 `Symfony\Component\HttpFoundation\Cookie` 实例以便后续附加到响应实例，

可以使用一个全局的辅助函数 `cookie`，这个 cookie 只有在附加到响应实例上才会发送到客户端：

```
$cookie = cookie('name', 'value', $minutes);  
  
return response('Hello World')->cookie($cookie);
```

3、文件上传

获取上传的文件

可以使用 `Illuminate\Http\Request` 实例提供的 `file` 方法或者动态属性来访问上传文件，`file` 方法返回 `Illuminate\Http\UploadedFile` 类的一个实例，该类继承自 PHP 标准库中提供与文件交互方法的 `SplFileInfo` 类：

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

你可以使用 `hasFile` 方法判断文件在请求中是否存在：

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

```
//  
}
```

验证文件是否上传成功

使用 `isValid` 方法判断文件在上传过程中是否出错：

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

文件路径 & 扩展名

`UploadedFile` 类还提供了访问上传文件绝对路径和扩展名的方法。`extension` 方法可以基于文件内容判断文件扩展名，该扩展名可能会和客户端提供的扩展名不一致：

```
$path = $request->photo->path();  
$extension = $request->photo->extension();
```

其他文件方法

`UploadedFile` 实例上还有很多其他可用方法，查看[该类的 API 文档](#)了解更多信息。

保存上传的文件

要保存上传的文件，通常需要使用你所配置的其中一个文件系统，`UploadedFile` 类有一个 `store` 方法，该方法会将上传文件移动到相应的磁盘路径上，该路径可以是本地文件系统的某个位置，也可以是云存储（如 Amazon S3）上的路径。

`store` 方法接收一个文件保存的相对路径（相对于文件系统配置的根目录），该路径不应该包含文件名，因为文件名会通过对文件内容进行 md5 自动生成。

`store` 方法还接收一个可选的参数——用于存储文件的磁盘名称作为第二个参数，该方法会返回相对于根目录的文件路径：

```
$path = $request->photo->store('images');  
$path = $request->photo->store('images', 's3');
```

如果你不想自动生成文件名，可以使用 `storeAs` 方法，该方法接收保存路径、文件名和磁盘名作为参数：

```
$path = $request->photo->storeAs('images', 'filename.jpg');  
$path = $request->photo->storeAs('images', 'filename.jpg', 's3');
```

5.6 响应

1、创建响应

字符串&数组

所有路由和控制器都会返回一个被发送到用户浏览器的响应，Laravel 提供了多种不同的方式来返回响应，最基本的响应就是从路由或控制器返回一个简单的字符串，框架会将这个字符串转化为一个完整的 HTTP 响应：

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

```
});
```

除了从路由或控制器返回字符串之外，还可以返回数组。框架会自动将数组转化为一个 [JSON](#) 响应：

```
Route::get('/', function () {
    return [1, 2, 3];
});
```

注：你知道还可以从路由或控制器返回 Eloquent 集合吗？这也会被自动转化为 JSON，试一试吧。

[Response 对象](#)

通常，我们并不只是从路由动作简单返回字符串和数组，大多数情况下，都会返回一个完整的 [Illuminate\Http\Response](#) 实例或[视图](#)。

返回一个完整的 Response 实例允许你自定义响应的 HTTP 状态码和头信息。Response 实例继承自 [Symfony\Component\HttpFoundation\Response](#) 类，该类提供了一系列方法用于创建 HTTP 响应：

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

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

添加响应头

大部分响应方法都是可以以方法链形式调用的，从而可以平滑地构建响应（流接口模式）。例如，

在发送响应给用户前可以使用 `header` 方法来添加一系列响应头：

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

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

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

添加 Cookie 到响应

使用响应实例上的 `cookie` 方法可以轻松添加 Cookie 到响应：

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

`cookie` 方法还可以接收更多使用频率较低的额外可选参数，一般来说，这些参数和 PHP 原生提供的 `setcookie` 方法目的和意义差不多：

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

Cookie & 加密

默认情况下，Laravel 框架生成的 Cookie 都经过了加密和签名，以免在客户端被篡改。如果你想让特定的 Cookie 子集在生成时取消加密，可以使用 `app/Http/Middleware` 目录下的中间件 `App\Http\Middleware\EncryptCookies` 提供的 `$except` 属性来排除这些 Cookie：

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

2、重定向

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

```
Route::get('dashboard', function () {
    return redirect('home/dashboard');
});
```

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

```
Route::post('user/profile', function () {
    // 验证请求...
    return back()->withInput();
});
```

重定向到命名路由

如果调用不带参数的 `redirect` 方法，会返回一个 `Illuminate\Routing\Redirector` 实例，然后就可以调用 `Redirector` 实例上的所有方法。例如，要生成一个 `RedirectResponse` 到命名路由，可以使用 `route` 方法：

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

如果路由中有参数，可以将其作为第二个参数传递到 `route` 方法：

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

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

通过 Eloquent 模型填充参数

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

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

如果你想要自定义这个路由参数中的默认值(默认是 id), 需要重写模型实例上的 `getRouteKey` 方法 :

```
/**  
 * Get the value of the model's route key.  
 *  
 * @return mixed  
 */  
public function getRouteKey()  
{  
    return $this->slug;  
}
```

重定向到控制器动作

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

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

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

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

带一次性 `Session` 数据的重定向

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

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

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

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

```
@endif
```

3、其它响应类型

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

视图响应

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

```
return response()->view('hello', $data, 200)->header('Content-Type', $type);
```

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

JSON 响应

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

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

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

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

文件下载

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

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

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

文件响应

`file` 方法可用于直接在用户浏览器显示文件，例如图片或 PDF，而不需要下载，该方法接收文件路径作为第一个参数，头信息数组作为第二个参数：

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

4、响应宏

如果你想要定义一个自定义的可以在多个路由和控制器中复用的响应，可以使用 `Response` 门面上的 `macro` 方法。

例如，在某个服务提供者的 `boot` 方法中编写代码如下：

```
<?php

namespace App\Providers;

use Response;
use Illuminate\Support\ServiceProvider;

class ResponseMacroServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Response::macro('caps', function ($value) {
            return Response::make(strtoupper($value));
        });
    }
}
```

`macro` 方法接收响应名称作为第一个参数，闭包函数作为第二个参数，`macro` 的闭包

在 `ResponseFactory` 实现类或辅助函数 `response` 中调用 `macro` 名称的时候被执行：

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

5.7 Session

1、简介

由于 HTTP 驱动的应用是无状态的，所以我们使用 [Session](#) 来存储用户请求信息。Laravel 通过干净、统一的 API 处理后端各种 Session 驱动，目前支持的流行后端驱动包括 [Memcached](#)、[Redis](#) 和[数据库](#)。

配置

Session 配置文件位于 `config/session.php`。默认情况下，Laravel 使用的 `session` 驱动为文件驱动，这对许多应用而言是没有什么问题的。在生产环境中，你可能考虑使用 `memcached` 或者 `redis` 驱动以便获取更快的 session 性能。

`session` 驱动定义请求的 Session 数据存放在哪里，Laravel 可以处理多种类型的驱动：

- `file` – session 数据存储在 `storage/framework/sessions` 目录下；
- `cookie` – session 数据存储在经过加密的安全的 cookie 中；
- `database` – session 数据存储在数据库中
- `memcached` / `redis` – session 数据存储在 memcached/redis 中；
- `array` – session 数据存储在简单 PHP 数组中，在多个请求之间是非持久化的。

注意：数组驱动通常用于运行测试以避免 session 数据持久化。

Session 驱动预备知识

数据库

当使用 `database` session 驱动时，需要设置表包含 `session` 项，下面是该数据表的表结构声明：

```
Schema::create('sessions', function ($table) {
    $table->string('id')->unique();
    $table->integer('user_id')->nullable();
    $table->string('ip_address', 45)->nullable();
    $table->text('user_agent')->nullable();
    $table->text('payload');
    $table->integer('last_activity');
});
```

你可以使用 Artisan 命令 `session:table` 来生成迁移：

```
php artisan session:table
php artisan migrate
```

Redis

在 Laravel 中使用 Redis 作为 `session` 驱动前，需要通过 Composer 安装 `predis/predis` 包。可以在 `database` 配置文件中配置 Redis 连接，在 `session` 配置文件中，`database` 选项用于指定 session 使用哪一个 redis 连接。

2、使用 Session

获取数据

在 Laravel 中主要有两种方式处理 Session 数据：全局的辅助函数 `session`，或者通过 Request 实例。首先，我们通过 Request 实例来访问 Session 数据，可以在控制器方法中通过类型提示引入该实例，记住，控制器方法依赖通过 Laravel [服务容器](#) 自动注入：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserController extends Controller{

    /**
     * 显示指定用户的属性
     *
     * @param Request $request
     * @param int $id
     * @return Response
     */

    public function showProfile(Request $request, $id)
    {
        $value = $request->session()->get('key');

        //
    }
}
```

从 session 中获取数据的时候，还可以传递默认值作为第二个参数到 `get` 方法，默认值在指定键在 session 中不存在时返回。如果你传递一个闭包作为默认值到 `get` 方法，该闭包会执行并返回执行结果：

```
$value = $request->session()->get('key', 'default');

$value = $request->session()->get('key', function() {
    return 'default';
});
```

全局辅助函数 session

还可以使用全局的 PHP 函数 `session` 来获取和存储 session 中的数据，如果只传递一个字符串参数到 `session` 方法，则返回该 session key 对应的值；如果传递的参数是 key/value 键值对数组，则将这些数据保存到 session：

```
Route::get('home', function () {
    // 从 session 中获取数据...
    $value = session('key');
    // 指定默认值...
    $value = session('key', 'default');
    // 存储数据到 session...
    session(['key' => 'value']);
});
```

注：通过 HTTP 请求实例和辅助函数 `session` 处理数据并无实质性差别，这两个方法在测试用例中都可以通过 `assertSessionHas` 方法进行测试。

获取所有 Session 值

如果你想要从 session 中获取所有数据，可以使用 `all` 方法：

```
$data = $request->session()->all();
```

判断 session 中是否存在指定项

`has` 方法可用于检查数据项在 session 中是否存在。如果存在并且不为 null 的话返回 `true`：

```
if ($request->session()->has('users')) {  
    //  
}
```

要判断某个值在 session 中是否存在，即使是 null 的话也无所谓，则可以使用 `exists` 方法。如果值存在的话 `exists` 返回 `true`：

```
if ($request->session()->exists('users')) {  
    //  
}
```

存储数据

获取到 session 实例后，就可以调用多个方法来与底层数据进行交互，例如，`put` 方法存储新的数据到 session 中：

```
//通过 put 方法  
  
$request->session()->put('key', 'value');  
  
//通过全局辅助函数  
  
session(['key' => 'value']);
```

推送数据到数组 session

`push` 方法可用于推送数据到值为数组的 session，例如，如果 `user.teams` 键包含团队名数组，可以像这样推送新值到该数组：

```
$request->session()->push('user.teams', 'developers');
```

获取并删除数据

`pull` 方法将会从 session 获取并删除数据：

```
$value = $request->session()->pull('key', 'default');
```

一次性数据

有时候你可能想要在 session 中存储只在下个请求中有效的数据，可以通过 `flash` 方法来实现。

使用该方法存储的 session 数据只在随后的 HTTP 请求中有效，然后将会被删除：

```
$request->session()->flash('status', 'Task was successful!');
```

如果你需要在更多请求中保持该一次性数据，可以使用 `reflash` 方法，该方法将所有一次性数据

保留到下一个请求，如果你只是想要保存特定一次性数据，可以使用 `keep` 方法：

```
$request->session()->reflash();
$request->session()->keep(['username', 'email']);
```

删除数据

`forget` 方法从 session 中移除指定数据，如果你想要从 session 中移除所有数据，可以使用 `flush`

方法：

```
$request->session()->forget('key');
```

```
$request->session()->flush();
```

重新生成 Session ID

重新生成 Session ID 经常用于阻止恶意用户对应用进行 session fixation 攻击(关于 session fixation 攻击 可参考这篇文章 :

http://www.360doc.com/content/11/1028/16/1542811_159889635.shtml)。

如果你使用内置的 `LoginController` 的话 , Laravel 会在认证期间自动重新生成 session id , 如果你需要手动重新生成 session ID , 可以使用 `regenerate` 方法 :

```
$request->session()->regenerate();
```

3、添加自定义Session 驱动

实现驱动

自定义 session 驱动需要实现 `SessionHandlerInterface` 接口 , 该接口包含少许我们需要实现的方法 , 比如一个 MongoDB 的实现如下 :

```
<?php

namespace App\Extensions;

class MongoHandler implements SessionHandlerInterface
{
    public function open($savePath, $sessionId) {}
```

```
public function close() {}

public function read($sessionId) {}

public function write($sessionId, $data) {}

public function destroy($sessionId) {}

public function gc($lifetime) {}

}
```

注 : Laravel 默认并没有附带一个用于包含扩展的目录 , 你可以将扩展放置在任何地方 , 这里我们创建一个 `Extensions` 目录用于存放 `MongoHandler`。

由于这些方法并不是很容易理解 , 所以我们接下来快速过一遍每一个方法 :

- `open` 方法用于基于文件的 session 存储系统 , 由于 Laravel 已经有了一个 `file` session 驱动 , 所以在该方法中不需要放置任何代码 , 可以将其置为空方法。
- `close` 方法和 `open` 方法一样 , 也可以被忽略 , 对大多数驱动而言都用不到该方法。
- `read` 方法应该返回与给定`$sessionId` 相匹配的 session 数据的字符串版本 , 从驱动中获取或存储 session 数据不需要做任何序列化或其它编码 , 因为 Laravel 已经为我们做了序列化。
- `write` 方法应该讲给定`$data` 写到持久化存储系统相应的`$sessionId` , 例如 MongoDB , Dynamo 等等。
- `destroy` 方法从持久化存储中移除 `$sessionId` 对应的数据。
- `gc` 方法销毁大于给定`$lifetime` 的所有 session 数据 , 对本身拥有过期机制的系统如 Memcached 和 Redis 而言 , 该方法可以留空。

注册驱动

驱动被实现后，需要准备将其注册到框架，要添加额外驱动到 Laravel session 后端，可以使用 Session 门面上的 `extend` 方法。我们在服务提供者的 `boot` 方法中调用该方法：

```
<?php

namespace App\Providers;

use App\Extensions\MongoSessionStore;
use Illuminate\Support\Facades\Session;
use Illuminate\Support\ServiceProvider;

class SessionServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Session::extend('mongo', function($app) {
            // Return implementation of SessionHandlerInterface...
            return new MongoSessionStore();
        });
    }
}
```

```
/**
 * Register bindings in the container.
 *
 * @return void
 */
public function register()
{
    //
}
```

session 驱动被注册之后，就可以在配置文件 `config/session.php` 中使用 `mongo` 驱动了。

5.8 验证

1、简介

Laravel 提供了多种方法来验证应用输入数据。默认情况下，Laravel 的控制器基类使用 `ValidatesRequests` trait，该 trait 提供了便利的方法通过各种功能强大的验证规则来验证输入的 HTTP 请求。

2、快速入门

要掌握 Laravel 强大的验证特性，让我们先看一个完整的验证表单并返回错误信息给用户的例子。

定义路由

首先，我们假定在 routes/web.php 文件中包含如下路由：

```
// 显示创建博客文章表单...
Route::get('post/create', 'PostController@create');
// 存储新的博客文章...
Route::post('post', 'PostController@store');
```

当然，GET 路由为用户显示了一个创建新的博客文章的表单，POST 路由将新的博客文章存储到数据库。

创建控制器

接下来，让我们看一个处理这些路由的简单控制器示例。我们先将 `store` 方法留空：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PostController extends Controller{
    /**
     * 显示创建新的博客文章的表单
     *
     * @return Response
     */
    public function create()
    {
        return view('post.create');
    }

    /**
     * 存储新的博客文章
     *
```

```
* @param Request $request
* @return Response
*/
public function store(Request $request)
{
    // 验证并存储博客文章...
}
```

编写验证逻辑

现在我们准备用验证新博客文章输入的逻辑填充 `store` 方法。如果你检查应用的控制器基类

(`App\Http\Controllers\Controller`) ,你会发现该类使用了 `ValidatesRequests` trait ,这个 trait 在所有控制器中提供了一个便利的 `validate` 方法。

`validate` 方法接收一个 HTTP 请求输入数据和验证规则 ,如果验证规则通过 ,代码将会继续往下执行 ;然而 ,如果验证失败 ,将会抛出一个异常 ,相应的错误响应也会自动发送给用户。在一个传统的 HTTP 请求案例中 ,将会生成一个重定向响应 ,如果是 AJAX 请求则会返回一个 JSON 响应。

要更好的理解 `validate` 方法 ,让我们回到 `store` 方法 :

```
/**
 * 存储博客文章
 *
 * @param Request $request
 * @return Response
 */
public function store(Request $request){
    $this->validate($request, [
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ]);

    // 验证通过, 存储到数据库...
}
```

```
}
```

正如你所看到的，我们只是传递输入的 HTTP 请求和期望的验证规则到 `validate` 方法，在强调一次，如果验证失败，相应的响应会自动生成。如果验证通过，控制器将会继续正常执行。

首次验证失败后中止后续规则验证

有时候你可能想要在首次验证失败后停止检查属性其它验证规则，要实现这个功能，可以在属性中分配 `bail` 规则：

```
$this->validate($request, [
    'title' => 'bail|required|unique:posts|max:255',
    'body' => 'required',
]);
```

在这个例子中，如果 `title` 属性上的 `required` 规则验证失败，则不会检查 `unique` 规则，规则会按照分配顺序依次进行验证。

嵌套属性注意事项

如果 HTTP 请求中包含“嵌套”参数，可以使用“.”在验证规则中指定它们：

```
$this->validate($request, [
    'title' => 'required|unique:posts|max:255',
    'author.name' => 'required',
    'author.description' => 'required',
]);
```

显示验证错误信息

那么，如果请求输入参数没有通过给定验证规则怎么办？正如前面所提到的，Laravel 将会自动将用户重定向回上一个位置。此外，所有验证错误信息会自动一次性存放到 session。

注意我们并没有在 GET 路由中明确绑定错误信息到视图。这是因为 Laravel 总是从 session 数据

中检查错误信息，而且如果有的话会自动将其绑定到视图。所以，值得注意的是每次请求的所有视图中总是存在一个 `$errors` 变量，从而允许你在视图中方便而又安全地使用。`$errors` 变量是的一个 `Illuminate\Support\MessageBag` 实例。想要了解更多关于该对象的信息，[查看其文档](#)。

注：`$errors` 变量会通过 web 中间件组中的 `Illuminate\View\Middleware\ShareErrorsFromSession` 中间件绑定到视图，如果使用了该中间件，那么 `$errors` 变量在视图中总是有效，从而方便你随时使用。

所以，在我们的例子中，验证失败的话用户将会被重定向到控制器的 `create` 方法，从而允许我们在视图中显示错误信息：

```
<!-- /resources/views/post/create.blade.php -->

<h1>Create Post</h1>

@if (count($errors) > 0)
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

<!-- Create Post Form -->
```

自定义错误格式

如果你想要自定义保存在 session 中的验证错误信息的格式，需要在控制器基类中重写

`formatValidationErrors` 方法（不要忘了在该控制器类的顶部导入 `Illuminate\Contracts\Validation\Validator` 类）：

```
<?php
```

```
namespace App\Http\Controllers;

use Illuminate\Foundation\Bus\DispatchesJobs;
use Illuminate\Contracts\Validation\Validator;
use Illuminate\Routing\Controller as BaseController;
use Illuminate\Foundation\Validation\ValidatesRequests;

abstract class Controller extends BaseController{
    use DispatchesJobs, ValidatesRequests;

    /**
     * {@inheritDoc}
     */
    protected function formatValidationErrors(Validator $validator)
    {
        return $validator->errors()->all();
    }
}
```

AJAX 请求&验证

在这个例子中，我们使用传统的表单来发送数据到应用。然而，很多应用使用 AJAX 请求。在 AJAX 请求中使用 `validate` 方法时，Laravel 不会生成重定向响应。取而代之的，Laravel 生成一个包含验证错误信息的 JSON 响应。该 JSON 响应会带上一个 HTTP 状态码 `422`。

3、表单请求验证

创建表单请求

对于更复杂的验证场景，你可能想要创建一个“表单请求”。表单请求是包含验证逻辑的自定义请求类，要创建表单验证类，可以使用 Artisan 命令 `make:request`：

```
php artisan make:request StoreBlogPost
```

生成的类位于 `app/Http/Requests` 目录下，接下来我们添加少许验证规则到 `rules` 方法：

```
/**  
 * 获取应用到请求的验证规则  
 *  
 * @return array  
 */  
public function rules(){  
    return [  
        'title' => 'required|unique:posts|max:255',  
        'body' => 'required',  
    ];  
}
```

那么，验证规则如何生效呢？你所要做的就是在控制器方法中类型提示该请求。表单输入请求会在控制器方法被调用之前被验证，这就是说你不需要将控制器和验证逻辑杂糅在一起：

```
/**  
 * 存储输入的博客文章  
 *  
 * @param StoreBlogPostRequest $request  
 * @return Response  
 */  
public function store(StoreBlogPost $request){  
    // The incoming request is valid...  
}
```

如果验证失败，重定向响应会被生成并将用户退回上一个位置，错误信息也会被一次性存储到 `session` 以便在视图中显示。如果是 AJAX 请求，带 `422` 状态码的 HTTP 响应将会返回给用户，该

响应数据中还包含了 JSON 格式的验证错误信息。

认证表单请求

表单请求类还包含了一个 `authorize` 方法，你可以检查认证用户是否有资格更新指定资源。例如，

如果用户尝试更新一个博客评论，那么他是否是评论的所有者呢？举个例子：

```
/**  
 * Determine if the user is authorized to make this request.  
 *  
 * @return bool  
 */  
public function authorize()  
{  
    $comment = Comment::find($this->route('comment'));  
  
    return $comment && $this->user()->can('update', $comment);  
}
```

由于所有请求都继承自 Laravel 请求基类，我们可以使用 `user` 方法获取当前认证用户，还要注意

上面这个例子中对 `route` 方法的调用。该方法赋予用户访问被调用路由 URI 参数的权限，比如下

面这个例子中的 `{comment}` 参数：

```
Route::post('comment/{comment}');
```

如果 `authorize` 方法返回 `false`，一个包含 `403` 状态码的 HTTP 响应会自动返回而且控制器方法将不会被执行。

如果你计划在应用的其他部分包含认证逻辑，只需在 `authorize` 方法中简单返回 `true` 即可：

```
/**
```

```
* 判断请求用户是否经过认证
*
* @return bool
*/
public function authorize(){
    return true;
}
```

自定义错误格式

如果你想要自定义验证失败时一次性存储到 `session` 中验证错误信息的格式，重写请求基类

(`App\Http\Requests\Request`) 中的 `formatErrors` 方法即可。不要忘记在文件顶部导入

`Illuminate\Contracts\Validation\Validator` 类：

```
/**
 * {@inheritDoc}
 */
protected function formatErrors(Validator $validator){
    return $validator->errors()->all();
}
```

自定义错误消息

你可以通过重写 `messages` 方法自定义表单请求使用的错误消息，该方法应该返回属性/规则对数

组及其对应错误消息：

```
/**
 * Get the error messages for the defined validation rules.
 *
 * @return array
 */
```

```
public function messages(){
    return [
        'title.required' => 'A title is required',
        'body.required'  => 'A message is required',
    ];
}
```

4、 手动创建验证器

如果你不想使用 `ValidatesRequests` trait 提供的 `validate` 方法，可以使用 `Validator` 门面手动创建一个验证器实例，该门面上的 `make` 方法用于生成一个新的验证器实例：

```
<?php

namespace App\Http\Controllers;

use Validator;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PostController extends Controller{
    /**
     * 存储新的博客文章
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        $validator = Validator::make($request->all(), [
            'title' => 'required|unique:posts|max:255',
            'body'  => 'required',
        ]);

        if ($validator->fails()) {
            return redirect('post/create')
        }
    }
}
```

```
->withErrors($validator)
->withInput();
}

// 存储博客文章...
}

}
```

传递给 `make` 方法的第一个参数是需要验证的数据，第二个参数是要应用到数据上的验证规则。

检查请求是否通过验证后，可以使用 `withErrors` 方法将错误数据一次性存放到 session，使用该方法时，`$errors` 变量重定向后自动在视图间共享，从而允许你轻松将其显示给用户，`withErrors` 方法接收一个验证器、或者一个 `MessageBag`，又或者一个 PHP 数组。

自动重定向

如果你想要手动创建一个验证器实例，但仍然使用 `ValidatesRequest` trait 提供的自动重定向，可以调用已存在验证器上的 `validate` 方法，如果验证失败，用户将会被自动重定向，或者，如果是 AJAX 请求的话，返回 JSON 响应：

```
Validator::make($request->all(), [
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
])->validate();
```

命名错误包

如果你在单个页面上有多个表单，可能需要命名 `MessageBag`，从而允许你为指定表单获取错误信息。只需要传递名称作为第二个参数给 `withErrors` 即可：

```
return redirect('register')
    ->withErrors($validator, 'login');
```

然后你就可以从 `$errors` 变量中访问命名的 `MessageBag` 实例：

```
{{ $errors->login->first('email') }}
```

验证钩子之后

验证器允许你在验证完成后添加回调，这种机制允许你轻松执行更多验证，甚至添加更多错误信息到消息集合。使用验证器实例上的 `after` 方法即可：

```
$validator = Validator::make(...);

$validator->after(function($validator) {
    if ($this->somethingElseIsInvalid()) {
        $validator->errors()->add('field', 'Something is wrong with this field!');
    }
});

if ($validator->fails()) {
    //
}
```

5、处理错误信息

调用 `Validator` 实例上的 `errors` 方法之后，将会获取一个 `Illuminate\Support\MessageBag` 实例，该实例中包含了多种处理错误信息的便利方法。在所有视图中默认有效的 `$errors` 变量也是

一个 MessageBag 实例。

获取某字段的第一条错误信息

要获取指定字段的第一条错误信息，可以使用 `first` 方法：

```
$messages = $validator->errors();
echo $messages->first('email');
```

获取指定字段的所有错误信息

如果你想要简单获取指定字段的所有错误信息数组，使用 `get` 方法：

```
foreach ($messages->get('email') as $message) {
    //
}
```

如果是一个数组表单字段，可以使用 *`获取所有数组元素错误信息`：

```
foreach ($errors->get('attachments.*') as $message) {
    //
}
```

获取所有字段的所有错误信息

要获取所有字段的所有错误信息，可以使用 `all` 方法：

```
foreach ($messages->all() as $message) {
    //
}
```

判断消息中是否存在某字段的错误信息

`has` 方法可用于判断错误信息中是否包含给定字段：

```
if ($messages->has('email')) {  
    //  
}
```

获取指定格式的错误信息

```
echo $messages->first('email', '<p>:message</p>');
```

获取指定格式的所有错误信息

```
foreach ($messages->all('<li>:message</li>') as $message) {  
    //  
}
```

自定义错误信息

如果需要的话，你可以使用自定义错误信息替代默认的，有多种方法来指定自定义信息。首先，

你可以传递自定义信息作为第三方参数给 `Validator::make` 方法：

```
$messages = [  
    'required' => 'The :attribute field is required.',  
];  
  
$validator = Validator::make($input, $rules, $messages);
```

在本例中，`:attribute` 占位符将会被验证时实际的字段名替换，你还可以在验证消息中使用其他

占位符，例如：

```
$messages = [  
    'same'      => 'The :attribute and :other must match.',  
    'size'      => 'The :attribute must be exactly :size.',  
    'between'   => 'The :attribute must be between :min - :max.',  
    'in'        => 'The :attribute must be one of the following types:  
    :values',  
];
```

为给定属性指定自定义信息

有时候你可能只想为特定字段指定自定义错误信息，可以通过“.”来实现，首先指定属性名，然后是规则：

```
$messages = [
    'email.required' => 'We need to know your e-mail address!',
];
```

在语言文件中指定自定义消息

在很多案例中，你可能想要在语言文件中指定属性特定自定义消息而不是将它们直接传递给 `Validator`。要实现这个，添加消息到 `resources/lang/xx/validation.php` 语言文件的 `custom` 数组：

```
'custom' => [
    'email' => [
        'required' => 'We need to know your e-mail address!',
    ],
],
```

6、验证规则大全

下面是有效规则及其函数列表：

- [Accepted](#)
- [Active URL](#)
- [After \(Date\)](#)
- [Alpha](#)
- [Alpha Dash](#)
- [Alpha Numeric](#)
- [Array](#)
- [Before \(Date\)](#)
- [Between](#)
- [Boolean](#)
- [Confirmed](#)
- [Date](#)
- [Date Format](#)

- [Different](#)
 - [Digits](#)
 - [Digits Between](#)
 - [Dimensions \(图片文件\)](#)
 - [Distinct](#)
 - [E-Mail](#)
 - [Exists \(Database\)](#)
 - [file](#)
 - [filled](#)
 - [Image \(File\)](#)
 - [In](#)
 - [In Array](#)
 - [Integer](#)
 - [IP Address](#)
 - [JSON](#)
 - [Max](#)
 - [MIME Types \(File\)](#)
 - [MIME Type By File Extension](#)
 - [Min](#)
 - [Nullable](#)
 - [Not In](#)
 - [Numeric](#)
 - [Present](#)
 - [Regular Expression](#)
 - [Required](#)
 - [Required If](#)
 - [Required Unless](#)
 - [Required With](#)
 - [Required With All](#)
 - [Required Without](#)
 - [Required Without All](#)
 - [Same](#)
 - [Size](#)
 - [String](#)
 - [Timezone](#)
 - [Unique \(Database\)](#)
 - [URL](#)
- accepted**

在验证中该字段的值必须是 `yes`、`on`、`1` 或 `true`，这在“同意服务协议”时很有用。

active_url

该字段必须是一个基于 PHP 函数 `checkdnsrr` 的有效 URL

after:date

该字段必须是给定日期后的一个值，日期将会通过 PHP 函数 `strtotime` 传递：

```
'start_date' => 'required|date|after:tomorrow'
```

你可以指定另外一个比较字段而不是使用 `strtotime` 验证传递的日期字符串：

```
'finish_date' => 'required|date|after:start_date'
```

alpha

该字段必须是字母

alpha_dash

该字段可以包含字母和数字，以及破折号和下划线

alpha_num

该字段必须是字母或数字

array

该字段必须是 PHP 数组

before:date

验证字段必须是指定日期之前的一个数值，该日期将会传递给 PHP `strtotime` 函数。

between:min, max

验证字段尺寸在给定的最小值和最大值之间，字符串、数值和文件都可以使用该规则

boolean

验证字段必须可以被转化为 `boolean`，接收 `true`, `false`, `1`, `0`, `"1"`, 和 `"0"` 等输入。

confirmed

验证字段必须有一个匹配字段 `foo_confirmation`，例如，如果验证字段是 `password`，必须输入一

个与之匹配的 `password_confirmation` 字段

date

验证字段必须是一个基于 PHP `strtotime` 函数的有效日期

`date_format:format`

验证字段必须匹配指定格式，该格式将使用 PHP 函数 `date_parse_from_format` 进行验证。你应该

在验证字段时使用 `date` 或 `date_format`

`different:field`

验证字段必须是一个和指定字段不同的值

`digits:value`

验证字段必须是数字且长度为 `value` 指定的值

`digits_between:min, max`

验证字段数值长度必须介于最小值和最大值之间

`dimensions`

验证的图片尺寸必须满足该规定参数指定的约束条件：

```
'avatar' => 'dimensions:min_width=100,min_height=200'
```

有效的约束条件包括：`min_width, max_width, min_height, max_height, width, height, ratio`

`ratio` 约束应该是宽度/高度，这可以通过表达式 `3/2` 或浮点数 `1.5` 来表示：

```
'avatar' => 'dimensions:ratio=3/2'
```

`distinct`

处理数组时，验证字段不能包含重复值：

```
'foo.*.id' => 'distinct'
```

`email`

验证字段必须是格式化的电子邮件地址

`exists:table, column`

验证字段必须存在于指定数据表

基本使用：

```
'state' => 'exists:states'
```

指定自定义列名：

```
'state' => 'exists:states,abbreviation'
```

还可以添加更多查询条件到 `where` 查询子句：

```
'email' => 'exists:staff,email,account_id,1'
```

这些条件还可以包含`!`：

```
'email' => 'exists:staff,email,role,!admin'
```

还可以传递 `NULL` 或 `NOT NULL` 到 `where` 子句：

```
'email' => 'exists:staff,email,deleted_at,NULL'  
'email' => 'exists:staff,email,deleted_at,NOT_NULL'
```

有时，你可能需要为 `exists` 查询指定要使用的数据库连接，这可以通过在表名前通过`.前置数据`

库连接来实现：

```
'email' => 'exists:connection.staff,email'
```

file

该验证字段必须是上传成功的文件

filled

该验证字段如果存在则不能为空

image

验证文件必须是图片 (jpeg、png、bmp、gif 或者 svg)

in:foo, bar...

验证字段值必须在给定的列表中

in_array:另一个字段

验证字段必须在另一个字段中存在

integer

验证字段必须是整型

ip

验证字段必须是 IP 地址

JSON

验证字段必须是有效的 JSON 字符串

max:value

验证字段必须小于等于最大值，和字符串、数值、文件字段的 size 规则一起使用

mimetypes: text/plain...

验证文件必须匹配给定的 MIME 文件类型之一：

```
'video' => 'mimetypes:video/avi,video/mpeg,video/quicktime'
```

为了判断上传文件的 MIME 类型，框架将会读取文件内容来猜测 MIME 类型，这可能会和客户端 MIME 类型不同。

mimes:foo, bar, ...

验证文件的 MIMIE 类型必须是该规则列出的扩展类型中的一个

MIMIE 规则的基本使用：

```
'photo' => 'mimes:jpeg,bmp,png'
```

尽管你只需要指定扩展，该规则实际上验证的是通过读取文件内容获取到的文件 MIME 类型。

完整 的 MIME 类 型 列 表 及 其 相 应 的 扩 展 可 以 在 这 里 找 到：

<http://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types>
min:value

验证字段的最小值，对字符串、数值、文件字段而言，和 `size` 规则使用方式一致。

nullable

验证字段必须为 `null`，这在验证一些可以为 `null` 的原生数据如整型或字符串时很有用。

not_in:foo, bar, ...

验证字段值不在给定列表中

numeric

验证字段必须是数值

present

验证字段必须出现在输入数据中但可以为空。

regex:pattern

验证字段必须匹配给定正则表达式

注：使用 `regex` 模式时，规则必须放在数组中，而不能使用管道分隔符，尤其是正则表达式中使用管道符号时。

required

输入字段值不能为空，以下情况字段值都为空：

- 值为 null
- 值是空字符串
- 值是空数组或者空的 Countable 对象
- 值是上传文件但路径为空

required_if:anotherfield, value, ...

验证字段在另一个字段等于指定值 `value` 时是必须的

required_unless:anotherfield, value, ...

除了 `anotherfield` 字段等于 `value`，验证字段不能为空

required_with:foo, bar, ...

验证字段只有在任一其它指定字段存在的话才是必须的

required_with_all:foo, bar, ...

验证字段只有在所有指定字段存在的情况下才是必须的

required_without:foo, bar, ...

验证字段只有当任一指定字段不存在的情况下才是必须的

required_without_all:foo, bar, ...

验证字段只有当所有指定字段不存在的情况下才是必须的

same:field

给定字段和验证字段必须匹配

size:value

验证字段必须有和给定值 `value` 相匹配的尺寸，对字符串而言，`value` 是相应的字符数目；对数

值而言，`value` 是给定整型值；对文件而言，`value` 是相应的文件字节数

string

验证字段必须是字符串

`timezone`

验证字符必须是基于 PHP 函数 `timezone_identifiers_list` 的有效时区标识

`unique:table, column, except, idColumn`

验证字段在给定数据表上必须是唯一的，如果不指定 `column` 选项，字段名将作为默认 `column`。

指定自定义列名：

```
'email' => 'unique:users,email_address'
```

自定义数据库连接

有时候，你可能需要自定义验证器生成的数据库连接，正如上面所看到的，设置 `unique:users` 作

为验证规则将会使用默认数据库连接来查询数据库。要覆盖默认连接，在数据表名后使用“:”指定

连接：

```
'email' => 'unique:connection.users,email_address'
```

强制一个唯一规则来忽略给定 ID：

有时候，你可能希望在唯一检查时忽略给定 ID，例如，考虑一个包含用户名、邮箱地址和位置的“

更新属性“界面，当然，你将会验证邮箱地址是唯一的，然而，如果用户只改变用户名字段而并没

有改变邮箱字段，你不想要因为用户已经拥有该邮箱地址而抛出验证错误，你只想要在用户提供

的邮箱已经被别人使用的情况下才抛出验证错误，要告诉唯一规则忽略用户 ID，可以传递 ID 作

为第三个参数：

```
'email' => 'unique:users,email_address,'.$user->id
```

如果你的数据表使用主键字段不是 `id`，可以指定第四个输入参数：

```
'email' => 'unique:users,email_address,'.$user->id.',user_id'
```

添加额外的 where 子句：

还可以指定更多条件给 `where` 子句：

```
'email' => 'unique:users,email_address,NULL,id,account_id,1'
```

在上述规则中，只有 `account_id` 为 1 记录才会进行唯一性检查。

`url`

验证字段必须是基于 PHP 函数 `filter_var` 过滤的有效 URL

7、添加条件规则

验证何时出现

在某些场景下，你可能想要只有某个字段存在的情况下运行验证检查，要快速完成这个，添加

`sometimes` 规则到规则列表：

```
$v = Validator::make($data, [
    'email' => 'sometimes|required|email',
]);
```

在上例中，`email` 字段只有存在于 `$data` 数组时才会被验证。

复杂条件验证

有时候你可能想要基于更复杂的条件逻辑添加验证规则。例如，你可能想要只有在另一个字段值

大于 `100` 时才要求一个给定字段是必须的，或者，你可能需要只有当另一个字段存在时两个字段

才都有给定值。添加这个验证规则并不是一件头疼的事。首先，创建一个永远不会改变的静态规

则到 `Validator` 实例：

```
$v = Validator::make($data, [
    'email' => 'required|email',
    'games' => 'required|numeric',
]);
```

让我们假定我们的 web 应用服务于游戏收集者。如果一个游戏收集者注册了我们的应用并拥有超

过 `100` 个游戏，我们想要他们解释为什么他们会有这么多游戏，例如，也许他们在运营一个游戏

二手店，又或者他们只是喜欢收集。要添加这种条件，我们可以使用 `Validator` 实例上的 `sometimes`

方法：

```
$v->sometimes('reason', 'required|max:500', function($input) {
    return $input->games >= 100;
});
```

传递给 `sometimes` 方法的第一个参数是我们需要有条件验证的名称字段，第二个参数是我们想要

添加的规则，如果作为第三个参数的闭包返回 `true`，规则被添加。该方法让构建复杂条件验证变

得简单，你甚至可以一次为多个字段添加条件验证：

```
$v->sometimes(['reason', 'cost'], 'required', function($input) {
    return $input->games >= 100;
});
```

注：传递给闭包的 `$input` 参数是 `Illuminate\Support\Fluent` 的一个实例，可用于访问输入和文件。

8、验证数组输入

验证表单数组输入字段不再是件痛苦的事情，例如，要验证给定数组输入中每个 `email` 是否是唯一的，可以这么做：

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

类似地，在语言文件中你也可以使用`*`字符指定验证消息，从而可以使用单个消息定义基于数组字段的验证规则：

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

9、自定义验证规则

Laravel 提供了多种有用的验证规则；然而，你可能还是想要指定一些自己的验证规则。注册验证规则的一种方法是使用 `Validator` 门面的 `extend` 方法。让我们在[服务提供者](#)中使用这种方法来注册一个自定义的验证规则：

```
<?php

namespace App\Providers;

use Validator;
```

```
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * 启动应用服务
     *
     * @return void
     */
    public function boot()
    {
        Validator::extend('foo', function($attribute, $value, $parameters, $validator) {
            return $value == 'foo';
        });
    }

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

自定义验证器闭包接收四个参数：要验证的属性名称、属性值、传递给规则的参数数组以及

Validator 实例。

你还可以传递类和方法到 `extend` 方法而不是闭包：

```
Validator::extend('foo', 'FooValidator@validate');
```

定义错误信息

你还需要为自定义规则定义错误信息。你可以使用内联自定义消息数组或者在验证语言文件中添加条目来实现这一目的。消息应该被放到数组的第一维，而不是在只用于存放属性指定错误信息的 `custom` 数组内：

```
"foo" => "Your input was invalid!",  
"accepted" => "The :attribute must be accepted.",  
// 验证错误信息其它部分...
```

当创建一个自定义验证规则时，你可能有时候需要为错误信息定义自定义占位符，可以通过创建自定义验证器然后调用 Validator 门面上的 `replacer` 方法来实现。可以在服务提供者的 `boot` 方法中编写代码：

```
/**  
 * 启动应用服务  
 *  
 * @return void  
 */  
public function boot(){  
    Validator::extend(...);  
    Validator::replacer('foo', function($message, $attribute, $rule, $parameters) {  
        return str_replace(...);  
    });  
}
```

隐式扩展

默认情况下，被验证的属性如果没有提供或者验证规则为 `required` 而值为空，那么正常的验证

规则，包括自定义扩展将不会执行。例如，`unique` 规则将不会检验 `null` 值：

```
$rules = ['name' => 'unique'];
$input = ['name' => null];
Validator::make($input, $rules)->passes(); // true
```

如果要求即使为空时也要验证属性，则必须要暗示属性是必须的，要创建一个隐式扩展，可以使
用 `Validator::extendImplicit()` 方法：

```
Validator::extendImplicit('foo', function($attribute, $value, $parameters, $validator) {
    return $value == 'foo';
});
```

注：一个隐式扩展仅仅暗示属性是必须的，至于它到底是缺失的还是空值这取决于你。

6. 视图 & 模板

6.1 视图

1、创建视图

视图包含应用的 HTML 代码并将应用的控制器逻辑和表现逻辑进行分离。视图文件存放
在 `resources/views` 目录。

下面是一个简单视图：

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

```
<html>
  <body>
    <h1>Hello, {{ $name }}</h1>
  </body>
</html>
```

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

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

正如你所看到的，传递给 `view` 方法的第一个参数是 `resources/views` 目录下相应的视图文件的名字，第二个参数是一个数组，该数组包含了在该视图中所有有效的[数据](#)。在这个例子中，我们传递了一个 `name` 变量，在视图中通过执行 `echo` 将其显示出来。

当然，视图还可以嵌套在 `resources/views` 的子目录中，用“.”号来引用嵌套视图，比如，如果视图存放路径是 `resources/views/admin/profile.php`，那我们可以这样引用它：

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

判断视图是否存在

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

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

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

2、传递数据到视图

在上述例子中可以看到，我们可以简单通过数组方式将数据传递到视图：

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

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

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

在视图间共享数据

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

```
<?php

namespace App\Providers;

class AppServiceProvider extends ServiceProvider
{
    /**
     * 启动所有应用服务
     *
     * @return void
     */

    public function boot()
    {
        view()->share('key', 'value');
    }

    /**
     * 注册服务提供者
     *
     * @return void
     */

    public function register()
    {
        //
    }
}
```

3、视图 Composer

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

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

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class ComposerServiceProvider extends ServiceProvider
{
    /**
     * 在容器中注册绑定.
     *
     * @return void
     * @author http://laravelacademy.org
     */
    public function boot()
    {
        // 使用基于类的 composers...
    }
}
```

```
view()->composer(  
    'profile', 'App\Http\ViewComposers\ProfileComposer'  
)  
  
// 使用基于闭包的 composers...  
  
view()->composer('dashboard', function ($view) {});  
  
}  
  
/**  
 * 注册服务提供者.  
 *  
 * @return void  
 */  
  
public function register()  
{  
    //  
}  
}
```

注 :如果创建一个新的服务提供者来包含视图 Composer 注册 ,需要添加该服务提供者到配置文

件 `config/app.php` 的 `providers` 数组中。

现在我们已经注册了 Composer , 每次 `profile` 视图被渲染时都会执

行 `ProfileComposer@compose` , 接下来我们来定义该 Composer 类 :

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

```
use Illuminate\Contracts\View\View;

use Illuminate\Users\Repository as UserRepository;

class ProfileComposer

{
    /**
     * 用户仓库实现.
     *
     * @var UserRepository
     */

    protected $users;

    /**
     * 创建一个新的属性 composer.
     *
     * @param UserRepository $users
     * @return void
     */

    public function __construct(UserRepository $users)
    {
        // Dependencies automatically resolved by service container...
        $this->users = $users;
    }

    /**
     * 绑定数据到视图.
     *
     */

}
```

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

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

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

添加 Composer 到多个视图

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

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

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

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

```
//  
});
```

视图创建器

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

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

6.2 Blade 模板

1、简介

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

2、模板继承

定义布局

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

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

```
<html>

    <head>
        <title>App Name - @yield('title')</title>
    </head>

    <body>
        @section('sidebar')
            This is the master sidebar.
        @show

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

正如你所看到的，该文件包含典型的 HTML 标记，然而，注意 `@section` 和 `@yield` 指令，前者正

如其名字所暗示的，定义了一个内容的片段，而后者用于显示给定片段的内容。

现在我们已经为应用定义了一个布局，接下来让我们定义继承该布局的子页面吧。

扩展布局

定义子页面的时候，可以使用 Blade 的 `@extends` 指令来指定子页面所继承的布局，继承一个

Blade 布局的视图将会使用 `@section` 指令注入内容到布局的片段中，记住，如上面例子所示，这

些片段的内容将会显示在布局中使用`@yield` 的地方：

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

```
@extends('layouts.master')

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

@section('sidebar')
    @parent
    <p>This is appended to the master sidebar.</p>
@endsection

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

在本例中，`sidebar` 片段使用 `@parent` 指令来追加(而非覆盖)内容到布局中 `sidebar`，`@parent` 指令在视图渲染时将会被布局中的内容替换。

当然，和原生 PHP 视图一样，Blade 视图可以通过 `view` 方法直接从路由中返回：

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

3、数据显示

可以通过两个花括号包裹变量来显示传递到视图的数据，比如，如果给出如下路由：

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

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

那么可以通过如下方式显示 `name` 变量的内容：

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

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

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

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

输出存在的数据

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

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

除了使用三元运算符，Blade 还提供了更简单的方式：

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

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

显示原生数据

默认情况下，Blade 的 `{{ }}` 语句已经通过 PHP 的 `htmlentities` 函数处理以避免 XSS 攻击，

如果你不想要数据被处理，可以使用如下语法：

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

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

Blade & JavaScript 框架

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

```
<h1>Laravel</h1>

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

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

@verbatim 指令

如果你在模板中很大一部分显示 JavaScript 变量，那么可以将这部分 HTML 封装在 @verbatim 指令中，这样就不需要在每个 Blade 输出表达式前加上 @ 前缀：

```
@verbatim

<div class="container">

Hello, {{ name }}.
```

```
</div>  
@endverbatim
```

4、流程控制

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

if 语句

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

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

为方便起见，Blade 还提供了 `@unless` 指令：

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

循环

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

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

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

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

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

在循环的时候可以使用`$loop` 变量获取循环信息 , 例如是否是循环的第一个或最后一个迭代 :

```
@foreach ($users as $user)
    @if ($user->type == 1)
```

```
@continue  
@endif  
  
<li>{{ $user->name }}</li>  
  
@if ($user->number == 5)  
    @break  
@endif  
@endforeach
```

还可以使用指令声明来引入条件：

```
@foreach ($users as $user)  
    @continue($user->type == 1)  
  
    <li>{{ $user->name }}</li>  
  
    @break($user->number == 5)  
@endforeach
```

\$loop 变量

在循环的时候，可以在循环体中使用 `$loop` 变量，该变量提供了一些有用的信息，比如当前循环索引，以及当前循环是不是第一个或最后一个迭代：

```
@foreach ($users as $user)  
    @if ($loop->first)
```

```
This is the first iteration.  
@endif  
  
@if ($loop->last)  
    This is the last iteration.  
@endif  
  
<p>This is user {{ $user->id }}</p>  
@endforeach
```

如果你身处嵌套循环，可以通过 `$loop` 变量的 `parent` 属性访问父级循环：

```
@foreach ($users as $user)  
    @foreach ($user->posts as $post)  
        @if ($loop->parent->first)  
            This is first iteration of the parent loop.  
        @endif  
    @endforeach  
@endforeach
```

`$loop` 变量还提供了一些有用的其他属性：

属性	描述
<code>\$loop->index</code>	当前循环迭代索引 (从 0 开始).
<code>\$loop->iteration</code>	当前循环迭代 (从 1 开始).

属性	描述
<code>\$loop->remaining</code>	当前循环剩余的迭代
<code>\$loop->count</code>	迭代数组元素的总数量
<code>\$loop->first</code>	是否是当前循环的第一个迭代
<code>\$loop->last</code>	是否是当前循环的最后一个迭代
<code>\$loop->depth</code>	当前循环的嵌套层级
<code>\$loop->parent</code>	嵌套循环中的父级循环变量

注释

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

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

5、包含子视图

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

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

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

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

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

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

为集合渲染视图

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

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

该指令的第一个参数是数组或集合中每个元素要渲染的局部视图，第二个参数是你希望迭代的数

组或集合，第三个参数是要分配给当前视图的变量名。举个例子，如果你要迭代一个 `jobs` 数组，

通常你需要在局部视图中访问 `$job` 变量。在局部视图中可以通过 `key` 变量访问当前迭代的键。

你还可以传递第四个参数到 `@each` 指令，该参数用于指定给定数组为空时渲染的视图：

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

6、堆栈

Blade 允许你推送内容到命名堆栈，以便在其他视图或布局中渲染。这在子视图中引入指定 JavaScript 库时很有用：

```
@push('scripts')
<script src="/example.js"></script>
@endpush
```

推送次数不限，要渲染完整的堆栈内容，传递堆栈名称到 `@stack` 指令即可：

```
<head>
  <!-- Head Contents -->

  @stack('scripts')
</head>
```

7、服务注入

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

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

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

8、扩展 Blade

Blade 甚至还允许你自定义指令，可以使用 `directive` 方法来注册一个指令。当 Blade 编译器遇到该指令，将会传入参数并调用提供的回调。

下面的例子创建了一个 `@datetime($var)` 指令格式化给定的 DateTime 的实例 `$var`：

```
<?php

namespace App\Providers;

use Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Blade::directive('datetime', function($expression) {
```

```
        return "<?php echo $expression->format('m/d/Y H:i'); ?>";  
    };  
  
}  
  
/**  
 * 在容器中注册绑定.  
 *  
 * @return void  
 */  
public function register()  
{  
    //  
}  
}
```

正如你所看到的，我们可以将 `format` 方法应用到任何传入指令的表达式上。最终该指令生成的

PHP 代码如下：

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

注：更新完 Blade 指令逻辑后，必须删除所有的 Blade 缓存视图。缓存的 Blade 视图可以通过

Artisan 命令 `view:clear` 移除。

6.3 本地化

1、简介

Laravel 的本地化特性允许你在应用中轻松实现多种语言支持。语言字符串默认存放在

`resources/lang` 目录中，在该目录中应该包含应用支持的每种语言的子目录：

```
/resources
  /lang
    /en
      messages.php
    /es
      messages.php
```

所有语言文件都返回一个键值对数组，例如：

```
<?php

return [
    'welcome' => 'Welcome to our application'
];
```

配置 Locale 选项

应用默认语言存放在配置文件 `config/app.php` 中，当然，你可以修改该值来匹配应用需要。你还

可以在运行时使用 `App` 门面上的 `setLocale` 方法改变当前语言：

```
Route::get('welcome/{locale}', function ($locale) {
    App::setLocale($locale);
    //
});
```

你还可以配置一个“备用语言”，当当前语言不包含给定语言时备用语言被返回。和默认语言一

样，备用语言也在配置文件 `config/app.php` 中配置：

```
'fallback_locale' => 'en',
```

判断当前的本地化语言

你可以使用 `App` 门面上的 `getLocale` 和 `isLocale` 方法来获取当前的本地化语言或者检查是否与

给定本地化匹配：

```
$locale = App::getLocale();

if (App::isLocale('en')) {
    //
}
```

2、获取语言行

你可以使用帮助函数 `trans` 从语言文件中获取行，该方法接收文件和语言行的键作为第一个参数，

例如，让我们在语言文件 `resources/lang/messages.php` 中获取语言行 `welcome`：

```
echo trans('messages.welcome');
```

当然如果你使用 [Blade 模板引擎](#)，可以使用`{{ }}`语法打印语言行或者使用 `@lang` 指令：

```
{{ trans('messages.welcome') }}
```

```
@lang('messages.welcome')
```

如果指定的语言行不存在，`trans` 函数将返回语言行的键，所以，使用上面的例子，如果语言行

不存在的话，`trans` 函数将返回 `messages.welcome`。

替换语言行中的参数

如果需要的话，你可以在语言行中定义占位符，所有的占位符都有一个`:前缀`，例如，你可以用占

位符名称定义一个 `welcome` 消息：

```
'welcome' => 'Welcome, :name',
```

要在获取语言行的时候替换占位符，传递一个替换数组作为 `trans` 函数的第二个参数：

```
echo trans('messages.welcome', ['name' => 'Dayle']);
```

如果占位符都是大写的，或者首字母是大写的，那么相应的，传入的值也会保持和占位符格式一致：

```
'welcome' => 'Welcome, :NAME', // Welcome, DAYLE  
'goodbye' => 'Goodbye, :Name', // Goodbye, Dayle
```

复数

复数是一个复杂的问题，因为不同语言对复数有不同的规则，通过使用管道字符“|”，你可以区分一个字符串的单数和复数形式：

```
'apples' => 'There is one apple|There are many apples',
```

然后，你可以使用 `trans_choice` 函数获取给定行数的语言行，在本例中，由于行数大于 1，将会返回语言行的复数形式：

```
echo trans_choice('messages.apples', 10);
```

Laravel [翻译](#)器由 Symfony 翻译组件提供，因此你可以创建更复杂的复数规则：

```
'apples' => '{0} There are none|[1,19] There are some|[20,Inf] Ther  
e are many',
```

3、覆盖 Vendor 包的语言文件

有些包可以处理自己的语言文件。你可以通过将自己的文件放在 `resources/lang/vendor/{package}/{locale}` 目录下来覆盖它们而不是破坏这些包的核心文件来调整这些句子。

所以，举个例子，如果你需要覆盖名为 `skyrim/hearthfire` 的包中的 `messages.php` 文件里的英文句子，可以创建一个 `resources/lang/vendor/hearthfire/en/messages.php` 文件。在这个文件中只需要定义你想要覆盖的句子，你没有覆盖的句子仍然从该包原来的语言文件中加载。

7. JavaScript & CSS

7.1 起步

1、简介

Laravel 并不强制你使用什么 `JavaScript` 或者 `CSS` 预处理器，不过也确实提供了对很多应用都很有用的 `Bootstrap` 和 `Vue` 的一些基本使用。默认情况下，Laravel 使用 `NPM` 来安装这些前端包。`CSS`

Laravel `Elixir` 提供了干净的、优雅的 API 用于编译 SASS 或 Less，SASS 和 Less 都是在原生 CSS 的基础上新增了变量、混合 (MixIn) 以及其它强大的功能特性，从而让我们在使用 CSS 的时候更加享受。

在本[文档](#)中，我们会简要讨论 CSS 的编译，不过，你最好参考完整的 Laravel Elixir 文档了解更多 SASS 或 Less 的编译细节。

JavaScript

Laravel 并不强制你使用指定的 JavaScript 框架或库来构建应用，事实上，你也可以完全不使用 JavaScript，不过， Laravel 还是引入了一些基本的脚手架：使用 Vue 库让我们更轻松地编写现代 JavaScript。Vue 提供了优雅的 API 让我们可以通过组件构建强大的 JavaScript 应用。

2、编写 CSS

Laravel 应用根目录下的 `package.json` 文件包含了 `bootstrap-sass` 扩展包以便我们使用 [Bootstrap](#) 构建前端原型，不过，你也可以按照自己应用的需要来增删 `package.json` 文件中的扩展包。此外，并不是必须要使用 Bootstrap 框架来构建 Laravel 应用——这只是为选择使用 Bootstrap 的开发者提供一个好的起点。

编译 CSS 之前，使用 [NPM](#) 安装应用的前端依赖：

```
npm install
```

使用 `npm install` 安装好前端依赖之后，可以使用 [Gulp](#) 编译 SASS 文件为原生的 CSS，`gulp` 命令会处理 `gulpfile.js` 文件中的声明。通常，编译好的 CSS 文件会被放置到 `public/css` 目录下：

```
gulp
```

Laravel 自带的默认 `gulpfile.js` 文件会编译 SASS 文件 `resources/assets/sass/app.scss`，这个 `app.scss` 文件将会导入一个包含 SASS 变量的文件并加载 Bootstrap，从而助力我们快速在应用中引入 Bootstrap 资源，你也可以按照自己的需要自定义 `app.scss` 文件，甚至可以通过配置 Laravel Elixir 使用一个完全不同的预处理器。

3、编写 JavaScript

应用所需要的所有 JavaScript 依赖都可以在应用根目录下的 `package.json` 中找到，这个文件和 `composer.json` 类似，只不过它指定的是 JavaScript 依赖而不是 PHP 依赖。你可以使用 NPM 来安装这些依赖：

```
npm install
```

默认情况下，Laravel 自带的 `package.json` 文件引入了一些扩展包，比如 `vue` 和 `vue-resource`，以便你快速构建 JavaScript 应用，同样，你可以按照应用的需要随意增删 `package.json` 中的扩展包。

扩展包安装好之后，可以使用 `gulp` 命令来编译前端资源，[Gulp](#) 是一个 JavaScript 命令行构建工具，当你执行 `gulp` 命令的时候，Gulp 将会执行 `gulpfile.js` 中的声明：

```
gulp
```

默认情况下，Laravel 自带的 `gulpfile.js` 将会编译 SASS 和 `resources/assets/js/app.js` 文件，在 `app.js` 文件中你可以注册 Vue 组件，或者你倾向于其它 JavaScript 框架，配置你自己的 JavaScript 应用。你所编译的 JavaScript 文件通常会存放在 `public/js` 目录下。

注：`app.js` 文件会加载 `resources/assets/js/bootstrap.js` 以便启动和配置 Vue，Vue 资源，jQuery 以及所有其它 JavaScript 依赖，如果你有额外的 JavaScript 依赖需要配置，请在这里操作。

编写 Vue 组件

默认情况下，新安装的 Laravel 应用将会在 `resources/assets/js/components` 目录下包含一个 Vue 组件 `Example.vue`，这个 Vue 组件是一个单文件 Vue 组件示例，其中定义了相关的 JavaScript 和 HTML 模板，单文件组件为构建 JavaScript 驱动的应用提供了便利。这个示例组件在 `app.js` 中注册：

```
Vue.component('example', require('./components/Example.vue'));
```

要在应用中使用这个组件，只需要将其丢到某个 HTML 模板中。例如，在运行完 Artisan 命令 `make:auth` 创建登录和注册视图之后，就可以将这个组件丢到 Blade 模板 `home.blade.php` 中：

```
@extends('layouts.app')

@section('content')
<example></example>
@endsection
```

注：记住，每次修改 Vue 组件后都要运行一次 `gulp` 命令，或者，你也可以运行 `gulp watch` 命令进行监听，一旦组件被修改后可以自动进行重新编译。

如果你对编写 Vue 组件感兴趣，可以去阅读 [Vue 文档](#)，从而对 Vue 框架有更加全面的认识。

7.2 编译资源 (Laravel Elixir)

1、简介

Laravel Elixir 提供了一套干净、平滑的 API 用于为 Laravel 应用定义基本的 [Gulp](#) 任务。Elixir 支持一些通用的 CSS 和 JavaScript 预处理器，甚至测试工具。使用方法链，Elixir 允许你平滑的定义资源管道。例如：

```
elixir(function(mix) {  
    mix.sass('app.scss')  
        .coffee('app.coffee');  
});
```

如果你曾经对如何使用 [Gulp](#) 和[编译前端资源](#)感到困惑，那么你会爱上 Laravel Elixir。不过，并不是强制要求在开发期间使用它。你可以自由选择使用任何前端资源管道工具，或者压根不使用。

2、安装 & 设置

安装 Node

在开始 Elixir 之前，必须首先确保 [Node.js](#) 在机器上已经安装：

```
node -v  
npm -v
```

默认情况下，Laravel Homestead 包含你需要的一切；然而，如果你不使用 Vagrant，你也可以通过访问 [Node 的下载页面](#) 轻松的安装 Node。

Gulp

接下来，需要安装 Gulp 作为全局 NPM 包：

```
npm install --global gulp-cli
```

Laravel Elixir

最后，在新安装的 Laravel 根目录下，你会发现有一个 `package.json` 文件。该文件和 `composer.json` 一样，只不过是用来定义 Node 依赖而非 PHP 依赖，你可以通过运行如下命令来安装需要的依赖：

```
npm install
```

如果你正在 Windows 系统上开发，需要在运行 `npm install` 命令时带上 `--no-bin-links`：

```
npm install --no-bin-links
```

3、运行 Elixir

Elixir 基于 Gulp，所以要运行 Elixir 命令你只需要在终端中运行 `gulp` 命令即可。添加 `--production` 标识到命令将会最小化 CSS 和 JavaScript 文件：

```
// Run all tasks...
gulp

// Run all tasks and minify all CSS and JavaScript...
gulp --production
```

运行这个命令的时候，你会看到一个显示刚刚发生事件的格式良好的表格。

监控前端资源改变

由于每次修改前端资源后都要运行 `gulp` 很不方便，可以使用 `gulp watch` 命令。该命令将会一直在终端运行并监控前端文件的改动。当改变发生时，新文件将会自动被编译：

```
gulp watch
```

4、处理样式表

项目根目录下的 `gulpfile.js` 文件包含了所有的 Elixir 任务。Elixir 任务可以使用方法链的方式链接起来用于定义前端资源如何被编译。

Less

要将 `Less` 编译成 `CSS`，可以使用 `less` 方法。`less` 方法假定你的 `Less` 文件都放在 `resources/assets/less`。默认情况下，本例中该任务会将编译后的 `CSS` 放到 `public/css/app.css`：

```
elixir(function(mix) {  
    mix.less('app.less');  
});
```

你还可以将多个 `Less` 文件编译成单个 `CSS` 文件。同样，该文件会被放到 `public/css/app.css`：

```
elixir(function(mix) {  
    mix.less([  
        'app.less',  
        'controllers.less'  
    ]);  
});
```

如果你想要自定义编译后文件的输出位置，可以传递第二个参数到 `less` 方法：

```
elixir(function(mix) {  
    mix.less('app.less', 'public/stylesheets');  
});  
  
// Specifying a specific output filename...  
elixir(function(mix) {  
    mix.less('app.less', 'public/stylesheets/style.css');  
});
```

Sass

`sass` 方法允许你将 `Sass` 编译成 `CSS`。假定你的 `Sass` 文件存放在 `resources/assets/sass`，你可以像这样使用该方法：

```
elixir(function(mix) {  
    mix.sass('app.scss');  
});
```

同样，和 `less` 方法一样，你可以将多个脚本编译成单个 `CSS` 文件，甚至自定义结果 `CSS` 的输出路径：

```
elixir(function(mix) {  
    mix.sass([  
        'app.scss',  
        'controllers.scss'  
    ], 'public/assets/css');  
});
```

自定义路径

尽管我们推荐你使用默认的前端资源目录，但是如果你确实需要设置其它目录，可以在相应文件路径前加上 `./`，这表明 Elixir 将会从项目根目录开始寻找文件，而不是使用默认的根目录。

例如，要编译 `app/assets/sass/app.scss` 然后将编译后文件输出到 `public/css/app.css`，可以这样调用 `sass` 方法：

```
elixir(function(mix) {  
    mix.sass('./app/assets/sass/app.scss');  
});
```

Stylus

`stylus` 方法可用于将 `Stylus` 编译成 `CSS`。假设你的 `Stylus` 文件存放在 `resources/assets/stylus`，

那么可以这样调用该方法：

```
elixir(function(mix) {  
    mix.styles('app.styl');  
});
```

注：该方法的调用和 `mix.less()`、`mix.sass()` 类似。

原生 CSS

如果你只想要将多个原生 CSS 样式文件合并到一个文件，可以使用 `styles` 方法。传递给该方法的路径相对于 `resources/assets/css` 目录，结果 CSS 被存放在 `public/css/all.css`：

```
elixir(function(mix) {  
    mix.styles([  
        'normalize.css',  
        'main.css'  
    ]);  
});
```

当然，你还可以通过传递第二个参数到 `styles` 方法来输出结果文件到一个自定义路径：

```
elixir(function(mix) {  
    mix.styles([  
        'normalize.css',  
        'main.css'  
    ], 'public/assets/css');  
});
```

源地图

在 Elixir 中，源地图 默认 被启用，以便在编译前端资源的时候为浏览器开发者工具提供更好的

调试信息。对于每一个被编译过的文件都可以在同一目录下找到一个对应的 `*.css.map` 或 `*.js.map` 文件。

如果你不想让应用 生成源地图，可以通过 `sourcemaps` 配置选项关闭它们：

```
elixir.config.sourcemaps = false;

elixir(function(mix) {
    mix.sass('app.scss');
});
```

5、处理脚本

Elixir 还提供了多个函数帮助你处理 JavaScript 文件，例如编译 ECMAScript 2015，模块管理，最小化以及合并原生 JavaScript 文件。

使用模块编写 ES2015 的时候，可以选择 [Webpack](#) 或 [Rollup](#)，如果你对这些工具很陌生，别担心，Elixir 会为你处理所有背后的复杂逻辑。默认情况下，Laravel 的 `gulpfile` 使用 `webpack` 来编译 JavaScript，当然，你也可以选择使用自己喜欢的模块管理器。

Webpack

`webpack` 方法用于将 [ECMAScript 2015](#) 编译打包成原生 JavaScript，该方法接收一个相对于 `resources/assets/js` 目录的文件路径，然后在 `public/js` 目录下生成单个打包文件：

```
elixir(function(mix) {
    mix.webpack('app.js');
});
```

要选择其他输出目录，只需在目标路径前加上`.`前缀，然后指定一个相对于应用根目录的相对路径。例如，要编译`app/assets/js/app.js` 到`public/dist/app.js`：

```
elixir(function(mix) {  
  mix.webpack(  
    './resources/assets/js/app.js',  
    './public/dist'  
  );  
});
```

如果你想要使用 Webpack 的更多功能，可以利用应用根目录下的`webpack.config.js` 文件，Elixir 将会读取该文件并将其中的配置用于构建过程。

Rollup

和 Webpack 相似，Rollup 是为 ES2015 准备的下一代模块管理器，该方法接收一个相对于`resources/assets/js` 目录的文件数组，然后在`public/js` 目录下生成单个文件：

```
elixir(function(mix) {  
  mix.rollup('app.js');  
});
```

和`webpack`方法一样，你也可以自定义传递给`rollup`方法的输入输出文件路径：

```
elixir(function(mix) {  
  mix.rollup(  
    ...  
  );  
});
```

```
'./resources/assets/js/app.js',  
'./public/dist'  
);  
});
```

脚本

如果你有多个 JavaScript 文件想要编译成单个文件，可以使用 `scripts` 方法。

`scripts` 方法假定所有路径相对于 `resources/assets/js` 目录，而且所有结果 JavaScript 默认存放在 `public/js/all.js`：

```
elixir(function(mix) {  
  mix.scripts([  
    'order.js',  
    'forum.js'  
  ]);  
});
```

如果你需要将多个脚本集合合并到不同的文件，需要多次调用 `scripts` 方法。该方法的第二个参数决定每个合并的结果文件名：

```
elixir(function(mix) {  
  mix.scripts(['app.js', 'controllers.js'], 'public/js/app.js')  
  .scripts(['forum.js', 'threads.js'], 'public/js/forum.js');  
});
```

如果你需要将多个脚本合并到给定目录，可以使用 `scriptsIn` 方法。结果 JavaScript 会被存放到

`public/js/all.js`：

```
elixir(function(mix) {  
    mix.scriptsIn('public/js/some/directory');  
});
```

注 如果你要合并多个已经最小化的 vendor 库文件 ,例如 jQuery ,可以考虑使用 `mix.combine()` ,

这将会合并这些文件 ,略过源地图和最小化步骤。最终 ,编译消耗时间将会大幅减小。

6、拷贝文件/目录

你可以使用 `copy` 方法拷贝文件/目录到新路径 ,所有操作都相对于项目根目录 :

```
elixir(function(mix) {  
    mix.copy('vendor/foo/bar.css', 'public/css/bar.css');  
});
```

7、版本号/缓存刷新

很多开发者会给编译的前端资源添加时间戳或者唯一令牌后缀以强制浏览器加载最新版本而不是代码的缓存副本。Elixir 可以使用 `version` 方法为你处理这种情况。

`version` 方法接收相对于 `public` 目录的文件名 ,附加唯一 hash 到文件名 ,从而实现缓存刷新。

例如 ,生成的文件名看上去是这样——`all-16d570a7.css` :

```
elixir(function(mix) {  
    mix.version('css/all.css');  
});
```

生成版本文件后 ,可以在视图中使用 Elixir 全局的 PHP 帮助函数 `elixir` 方法来加载相应的带 hash 值的前端资源 , `elixir` 函数会自动判断 hash 文件名 :

```
<link rel="stylesheet" href="{{ elixir('css/all.css') }}">
```

给多个文件加上版本号

你可以传递一个数组到 `version` 方法来为多个文件添加版本号：

```
elixir(function(mix) {  
    mix.version(['css/all.css', 'js/app.js']);  
});
```

文件被加上版本号后，就可以使用辅助函数 `elixir` 来生成指向该 hash 文件的链接。记住，你只需要传递没有 hash 值的文件名到 `elixir` 方法。该帮助函数使用未加 hash 值的文件名来判断文件当前的 hash 版本：

```
<link rel="stylesheet" href="{{ elixir('css/all.css') }}">  
  
<script src="{{ elixir('js/app.js') }}"></script>
```

8、BrowserSync

BrowserSync 会在你修改前端资源后自动刷新浏览器，`browserSync` 方法接收一个 JavaScript 对象，该对象有一个包含本地应用 URL 的属性——`proxy`。当你运行 `gulp watch` 命令时，可以通过 3000 端口 (<http://project.dev:3000>) 享受浏览器同步：

```
elixir(function(mix) {  
    mix.browserSync({  
        proxy: 'project.dev'  
    });  
});
```

8. 安全

8.1 用户认证

1、简介

注：想要更快上手？只需要在新安装的 [Laravel](#) 应用下运行 `php artisan make:auth`，然后在浏

览器中访问 <http://your-app.dev/register>，该命令会生成[用户登录注册](#)所需要的所有东西。

Laravel 中实现用户[认证](#)非常简单。实际上，几乎所有东西都已经为你配置好了。配置文件位于

`config/auth.php`，其中包含了用于调整认证服务行为的、[文档](#)友好的选项配置。

在底层代码中，Laravel 的认证组件由“guards”和“providers”组成，[Guard](#) 定义了用户在每个请求

中如何实现认证，例如，Laravel 通过 `session` guard 来维护 Session 存储的状态和 cookies。

Provider 定义了如何从持久化存储中获取用户信息，Laravel 底层支持通过 Eloquent 和[数据库](#)

查询构建器两种方式来获取用户，如果需要的话，你还可以定义额外的 Provider。

如果看到这些名词觉得云里雾里，大可不必太过担心，因为对绝大多数应用而言，只需使用默认

认证配置即可，不需要做什么改动。

数据库考量

默认情况下，Laravel 在 `app` 目录下包含了一个 Eloquent 模型 `App\User`，这个模型可以和默认的

Eloquent 认证驱动一起使用。如果你的应用不使用 Eloquent，你可以使用 `database` 认证驱动，

该驱动使用了 Laravel 查询构建器。

为 `App\User` 模型构建数据库表结构的时候，确保 `password` 字段长度至少有 60 位。

还有，你应该验证 `users` 表包含了可以为空的、字符串类型的 `remember_token` 字段长度为 100，该字段用于在登录时存储被应用维护的“记住我”的 Session 令牌。

2、快速入门

Laravel 提供了几个预置的认证控制器，位于 `App\Http\Controllers\Auth` 命名空间下，`RegisterController` 用于处理新用户注册，`LoginController` 用于处理用户登录认证，`ForgotPasswordController` 用于处理重置密码邮件链接，`ResetPasswordController` 包含重置密码逻辑，每个控制器都使用 trait 来引入它们需要的方法。对很多应用而言，你根本不需要修改这些控制器。

路由

Laravel 通过运行如下命令可快速生成认证所需要的路由和视图：

```
php artisan make:auth
```

运行该命令会生成注册和登录视图，以及所有的认证路由，同时生成 `HomeController` 用于处理应用的登录请求。

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。

所有这些视图都使用了 Bootstrap CSS 框架，你也可以根据需要对其进行自定义。

认证

现在你已经为自带的认证控制器设置好了路由和视图，接下来我们来实现新用户注册和登录认证。

你可以在浏览器中访问定义好的路由，认证控制器默认已经包含了注册及登录逻辑(通过 trait)。

自定义路径

当一个用户成功进行登录认证后，默认将会跳转到 `/home`，你可以通过在 `LoginController`，`RegisterController` 和 `ResetPasswordController` 中定义 `redirectTo` 属性来自定义登录认证成功之后的跳转路径：

```
protected $redirectTo = '/';
```

当一个用户登录认证失败后，默认将会跳转回登录表单对应的页面。

自定义 Guard

你还可以自定义用于实现用户注册登录的“guard”，要实现这一功能，需要在 `LoginController`，`RegisterController` 和 `ResetPasswordController` 中定义 `guard` 方法，该方法将会返回一个 guard 实例：

```
use Illuminate\Support\Facades\Auth;

protected function guard()
{
    return Auth::guard('guard-name');
}
```

自定义验证/存储

要修改新用户注册所必需的表单字段，或者自定义新用户字段如何存储到数据库，你可以修改 `RegisterController` 类。该类负责为应用验证输入参数和创建新用户。

`RegisterController` 的 `validator` 方法包含了新用户注册的验证规则，你可以按需要自定义该方法。

`RegisterController` 的 `create` 方法负责使用 Eloquent ORM 在数据库中创建新的 `App\User` 记录。

当然，你也可以基于自己的需要自定义该方法。

获取认证用户

你可以通过 `Auth` 面访问认证用户：

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

此外，用户通过认证后，你还可以通过 `Illuminate\Http\Request` 实例访问认证用户：

```
<?php

namespace App\Http\Controllers;

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

class ProfileController extends Controller{
    /**
     * 更新用户属性.

```

```
* @param Request $request
* @return Response
*/
public function update(Request $request)
{
    // $request->user() 返回认证用户实例...
}
}
```

判断当前用户是否通过认证

要判断某个用户是否登录到应用，可以使用 `Auth` 门面的 `check` 方法，如果用户通过认证则返回 `true`：

```
use Illuminate\Support\Facades\Auth;

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

注：尽管我们可以使用 `check` 方法判断用户是否通过认证，你还可以在用户访问特定路由/控制器之前使用中间件来验证用户是否通过认证，想要了解更多，可以查看下面的路由保护。

路由保护

路由中间件 可用于只允许通过认证的用户访问给定路由。Laravel 通过定义在 `app\Http\Middleware\Authenticate.php` 的 `auth` 中间件来处理这一操作。由于该中间件已经在

HTTP kernel 中定义，你所要做的仅仅是将该中间件加到相应的路由定义中：

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

当然，如果你正在使用[控制器类](#)，也可以在控制器的构造方法中调用 `middleware` 方法而不是在路由器中直接定义：

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

指定一个 Guard

添加 `auth` 中间件到路由后，还需要指定使用哪个 guard 来实现认证，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');
}
```

登录失败次数限制

如果你使用了 Laravel 内置的 `LoginController` 类，可以使用 `Illuminate\Foundation\Auth\ThrottlesLogins` trait 来限制用户登录失败次数。默认情况下，用户在几次登录失败后将在一分钟内不能登录，这种限制基于用户的用户名/邮箱地址+IP 地址。

3、手动认证用户

当然，你也可以不使用 Laravel 自带的认证控制器。如果你选择移除这些控制器，你需要直接使用 Laravel 认证类来管理用户认证。别担心，这很简单！

我们将会通过 `Auth` 门面来访问认证服务，因此我们需要确保在类的顶部导入了 `Auth` 门面，让我们看看 `attempt` 方法：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Auth;

class AuthController 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` 方法接收键值数组对作为第一个参数，数组中的值被用于从数据表中查找用户，因此，在上面的例子中，用户将会通过 `email` 的值获取，如果用户被找到，经哈希运算后存储在数据中的密码将会和传递过来的经哈希运算处理的密码值进行比较。如果两个经哈希运算的密码相匹配那么将会为这个用户开启一个认证 Session。

如果认证成功的话 `attempt` 方法将会返回 `true`。否则，返回 `false`。

重定向器上的 `intended` 方法将用户重定向到登录之前用户想要访问的 URL，在目标 URL 无效的情况下回退 URI 将会传递给该方法。

指定额外条件

如果需要的话，除了用户邮件和密码之外还可以在认证查询时添加额外的条件，例如，我们可以验证被标记为有效的用户：

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

注：在这些例子中，并不仅仅限于使用 `email` 进行登录认证，这里只是作为演示示例，你可以将其修改为数据库中任何其他可用作“username”的字段。

访问指定 Guard 实例

你可以使用 `Auth` 门面的 `guard` 方法指定想要使用的 `guard` 实例，这种机制允许你在同一个应用中对不同的认证模型或用户表实现完全独立的用户认证。

传递给 `guard` 方法的 `guard` 名称对应配置文件 `auth.php` 中 `guards` 配置的某个键：

```
if (Auth::guard('admin')->attempt($credentials)) {  
    //  
}
```

退出

要退出应用，可以使用 `Auth` 门面的 `logout` 方法，这将会清除用户 Session 中的认证信息：

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

记住用户

如果你想要在应用中提供“记住我”的功能，可以传递一个布尔值作为第二个参数到 `attempt` 方法，这样用户登录认证状态就会一直保持直到他们手动退出。当然，你的 `users` 表必须包含 `remember_token` 字段，该字段用于存储“记住我”令牌。

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

注：如果你在使用内置的 `LoginController` 控制器，相应的记住用户逻辑已经通过控制器使用的 trait 实现了。

如果你要“记住”用户，可以使用 `viaRemember` 方法来判断用户是否使用“记住我”cookie 进行认证：

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

其它认证方法

认证一个用户实例

如果你需要将一个已存在的用户实例登录到应用中 ,可以调用 `Auth` 门面的 `login` 方法并传入用户实例 ,传入实例必须是 `Illuminate\Contracts\Auth\Authenticatable` 契约的实现 ,当然 ,Laravel 自带的 `App\User` 模型已经实现了该接口 :

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

// Login and "remember" the given user...
Auth::login($user, true);
```

当然 ,你可以指定想要使用的 guard 实例 :

```
Auth::guard('admin')->login($user);
```

通过 ID 认证用户

要通过用户 ID 登录到应用 ,可以使用 `loginUsingId` 方法 ,该方法接收你想要认证用户的主键作为参数 :

```
Auth::loginUsingId(1);

// Login and "remember" the given user...
Auth::loginUsingId(1, true);
```

一次性认证用户

你可以使用 `once` 方法只在单个请求中将用户登录到应用，而不存储任何 Session 和 Cookie，这在构建无状态的 API 时很有用：

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

4、基于 HTTP 的基本认证

[HTTP 基本认证](#)能够帮助用户快速实现登录认证而不用设置专门的登录页面，首先要再路由中加上 `auth.basic` 中间件。该中间件是 Laravel 自带的，所以不需要自己定义：

```
Route::get('profile', function() {  
    // Only authenticated users may enter...  
})->middleware('auth.basic');
```

中间件加到路由中后，当在浏览器中访问该路由时，会自动提示需要认证信息，默认情况下，`auth.basic` 中间件使用用户记录上的 `email` 字段作为“用户名”。

FastCGI 上注意点

如果你使用 PHP FastCGI，HTTP 基本认证将不能正常工作，需要在 `.htaccess` 文件加入如下内容：

```
RewriteCond %{HTTP:Authorization} ^(.+)$  
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
```

无状态的 HTTP 基本认证

使用 HTTP 基本认证也不需要在 Session 中设置用户标识 Cookie , 这在 API 认证中非常有用。

要实现这个，需要定义一个调用 `onceBasic` 方法的中间件。如果该方法没有返回任何响应，那么请求会继续走下去：

```
<?php

namespace Illuminate\Auth\Middleware;

use 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);
    }
}
```

```
}
```

接下来，注册路由中间件并将其添加到路由中：

```
Route::get('api/user', function() {
    // Only authenticated users may enter...
})->middleware('auth.basic.once');
```

5、添加自定义的 Guard

你可以通过 `Auth` 门面的 `extend` 方法定义自己的认证 guard，该功能需要在某个服务提供者 `boot` 方法中实现，由于 Laravel 已经自带了一个 `AuthServiceProvider`，所以我们将代码放到这个服务提供者中：

```
<?php

namespace App\Providers;

use App\Services\Auth\JwtGuard;
use Illuminate\Support\Facades\Auth;
use Illuminate\Support\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']));
    });
}
```

正如你在上述例子中所看到的，传递给 `extend` 方法的闭包回调需要返回 `Illuminate\Contracts\Auth\Guard` 的实现实例，该接口包含了自定义认证 guard 需要的一些方法。定义好自己的认证 guard 之后，可以在配置文件 `auth.php` 的 `guards` 配置中使用这个 guard：

```
'guards' => [
    'api' => [
        'driver' => 'jwt',
        'provider' => 'users',
    ],
],
```

6、添加自定义用户提供者

如果你没有使用传统的关系型数据库存储用户信息，则需要使用自己的认证用户提供者来扩展 Laravel。我们使用 Auth 门面上的 `provider` 方法定义自定义该提供者：

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Auth;
use App\Extensions\RiakUserProvider;
use Illuminate\Support\ServiceProvider;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * 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，定义一个使用新驱动的 provider：

```
'providers' => [
    'users' => [
        'driver' => 'riak',
    ],
],
```

然后，可以在你的 guards 配置中使用这个提供者：

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],
],
```

User Provider 契约

`Illuminate\Contracts\Auth\UserProvider` 实现只负责从持久化存储系统中获取

`Illuminate\Contracts\Auth\Authenticatable` 实现，例如 MySQL、Riak 等等。这两个接口允许

Laravel 认证机制继续起作用而不管用户数据如何存储或者何种类来展现。

让我们先看看 `Illuminate\Contracts\Auth\UserProvider` 契约：

```
<?php

namespace Illuminate\Contracts\Auth;

interface UserProvider {

    public function retrieveById($identifier);
    public function retrieveByToken($identifier, $token);
    public function updateRememberToken(Authenticatable $user, $token);
    public function retrieveByCredentials(array $credentials);
    public function validateCredentials(Authenticatable $user, array
```

```
y $credentials);  
}  
}
```

`retrieveById` 方法通常获取一个代表用户的键，例如 MySQL 数据中的自增 ID。该方法获取并返回匹配该 ID 的 `Authenticatable` 实现。

`retrieveByToken` 函数通过唯一标识和存储在 `remember_token` 字段中的“记住我”令牌获取用户。

和上一个方法一样，该方法也返回 `Authenticatable` 实现。

`updateRememberToken` 方法使用新的 `$token` 更新 `$user` 的 `remember_token` 字段，新令牌可以是新生成的令牌（在登录时选择“记住我”被成功赋值）或者 null（用户退出）。

`retrieveByCredentials` 方法在尝试登录系统时获取传递给 `Auth::attempt` 方法的认证信息数组。该方法接下来去底层持久化存储系统查询与认证信息匹配的用户，通常，该方法运行一个带“where”条件（`$credentials['username']`）的查询。然后该方法返回 `UserInterface` 的实现。这个方法不做任何密码校验和认证。

`validateCredentials` 方法比较给定 `$user` 和 `$credentials` 来认证用户。例如，这个方法比较 `$user->getAuthPassword()` 字符串和经 `Hash::make` 处理的 `$credentials['password']`。这个方法只验证用户认证信息并返回布尔值。

Authenticatable 契约

既然我们已经探索了 `UserProvider` 上的每一个方法，接下来让我们看看 `Authenticatable`。该提供者应该从 `retrieveById` 和 `retrieveByCredentials` 方法中返回接口实现：

```
<?php
```

```
namespace Illuminate\Contracts\Auth;

interface Authenticatable {
    public function getAuthIdentifierName();
    public function getAuthIdentifier();
    public function getAuthPassword();
    public function getRememberToken();
    public function setRememberToken($value);
    public function getRememberTokenName();
}
```

这个接口很简单，`getAuthIdentifierName` 方法会返回用户的主键字段名称，`getAuthIdentifier` 方法返回用户“主键”，在后端 MySQL 中这将是自增 ID，`getAuthPassword` 返回经哈希处理的用户密码，这个接口允许认证系统处理任何用户类，不管是你使用的是 ORM 还是存储抽象层。默认情况下，Laravel `app` 目录下的 `User` 类实现了这个接口，所以你可以将这个类作为实现例子。

7、事件

Laravel 支持在认证过程中触发多种事件，你可以在自己的 `EventServiceProvider` 中监听这些事件：

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Auth\Events\Attempting' => [

```

```
'App\Listeners\LogAuthenticationAttempt',
],
['Illuminate\Auth\Events\Authenticated' => [
    'App\Listeners\LogAuthenticated',
],
],
['Illuminate\Auth\Events\Login' => [
    'App\Listeners\LogSuccessfulLogin',
],
],
['Illuminate\Auth\Events\Logout' => [
    'App\Listeners\LogSuccessfulLogout',
],
],
['Illuminate\Auth\Events\Lockout' => [
    'App\Listeners\LogLockout',
],
];
];
```

8.2 用户授权

1、简介

除了提供开箱即用的认证服务之外 ,Laravel 还提供了一个简单的方式来管理授权逻辑以便控制对

资源的访问权限。和认证一样 ,在 Laravel 中实现授权很简单 ,主要有两种方式 :gates 和 policies。

可以将 gates 和 policies 分别看作路由和控制器 , gates 提供了简单的基于闭包的方式进行授权 ,

而 policies 和控制器一样 ,对特定模型或资源上的复杂授权逻辑进行分组 ,本着由简入繁的思路 ,

我们首先来看 gates , 然后再看 policies。

不要将 gates 和 policies 看作互斥的东西 , 实际上 , 在大多数应用中我们会混合使用它们 , 这很

有必要 ,因为 gates 通常用于定于与模型或资源无关的权限 比如访问管理后台 与之相反 ,policies

则用于对指定模型或资源的动作进行授权。

2、Gates

编写 Gates

Gates 是一个用于判断[用户](#)是否有权进行某项操作的闭包，通常使用 [Gate](#) 门面定义在 [App\Providers\AuthServiceProvider](#) 类中。Gates 总是接收用户实例作为第一个参数，还可以接收相关的 Eloquent 模型实例作为额外参数：

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Gate::define('update-post', function ($user, $post) {
        return $user->id == $post->user_id;
    });
}
```

授权动作

要使用 gates 授权某个动作，可以使用 [allows](#) 方法，需要注意的是你可以不传用户实例到 [allows](#) 方法，Laravel 会自动将用户实例传递到 gate 闭包：

```
if (Gate::allows('update-post', $post)) {  
    // The current user can update the post...  
}
```

如果你想要判断指定用户(非当前用户)是否有权进行某项操作 ,可以使用 `Gate` 门面上的 `forUser`

方法 :

```
if (Gate::forUser($user)->allows('update-post', $post)) {  
    // The user can update the post...  
}
```

3、创建策略类

生成 Policies

Policies (策略) 是用于组织基于特定模型或资源的授权逻辑的类 , 例如 , 如果你开发的是一个博客应用 , 可以有一个 `Post` 模型和与之对应的 `PostPolicy` 来授权用户创建或更新博客的动作。

我们使用 Artisan 命令 `make:policy` 来生成一个 policy (策略), 生成的 policy 位于 `app/Policies` 目录下 , 如果这个目录之前不存在 , Laravel 会为我们生成 :

```
php artisan make:policy PostPolicy
```

`make:policy` 命令会生成一个空的 policy 类 , 如果你想要生成一个包含基本 CRUD 策略方法的 policy 类 , 在执行该命令的时候可以通过 `--model` 指定相应模型 :

```
php artisan make:policy PostPolicy --model=Post
```

注：所有策略类都通过服务容器进行解析，以便在策略类的构造函数中通过类型提示进行依赖注入。

注册 Policies

Policies 创建之后，需要进行注册。Laravel 自带的 `AuthServiceProvider` 包含了一个 `policies` 属性来映射 Eloquent 模型及与之对应的策略类。注册策略将会告知 Laravel 在授权给定模型动作时使用哪一个策略类：

```
<?php

namespace App\Providers;

use App\Post;
use App\Policies\PostPolicy;
use Illuminate\Support\Facades\Gate;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        Post::class => PostPolicy::class,
    ];

    /**
     * Register any application authentication / authorization serv
```

```
ices.  
*  
* @return void  
*/  
public function boot()  
{  
    $this->registerPolicies();  
  
    //  
}  
}
```

4、编写策略类

Policies 方法

策略类被注册后，还要为每个授权动作添加方法，例如，我们为用户更新 Post 实例这一动作在

PostPolicy 中定义一个 update 方法。

update 方法会接收一个 User 实例和一个 Post 实例作为参数，并且返回 true 或 false 以表明该

用户是否有权限对给定 Post 进行更新。因此，在这个例子中，我们验证用户的 id 和文章对应的

user_id 是否匹配：

```
<?php  
  
namespace App\Policies;  
  
use App\User;  
use App\Post;  
  
class PostPolicy
```

```
{  
    /**  
     * Determine if the given post can be updated by the user.  
     *  
     * @param \App\User $user  
     * @param \App\Post $post  
     * @return bool  
    */  
    public function update(User $user, Post $post)  
    {  
        return $user->id === $post->user_id;  
    }  
}
```

你可以继续在策略类中为授权的权限定义更多需要的方法 ,例如 ,你可以定义 `view` 或者 `delete` 等方法来授权多个 `Post` 动作 ,方法名不限。

注 :如果你在使用 Artisan 命令生成策略类的时候使用了 `--model` 选项 ,那么策略类中已经包含了 `view`、`create`、`update` 和 `delete` 动作。

不带模型的方法

有些策略方法只接收当前认证的用户 ,并不接收授权的模型实例作为参数 ,这种用法在授权 `create` 动作的时候很常见。例如 ,创建一篇博客的时候 ,你可能想要检查当前用户是否有权创建新博客。

当定义不接收模型实例的策略方法时 ,例如 `create` 方法 ,可以这么做 :

```
/**  
 * Determine if the given user can create posts.  
 *  
 * @param \App\User $user  
 * @return bool  
 */  
  
public function create(User $user)  
{  
    //  
}
```

策略过滤器

对特定用户，你可能想要在一个策略方法中对其授权所有权限，比如后台管理员。要实现这个功能，需要在策略类中定义一个 `before` 方法，`before` 方法会在策略类的所有其他方法执行前执行，从而确保在其他策略方法调用前执行其中的逻辑：

```
public function before($user, $ability)  
{  
    if ($user->isSuperAdmin()) {  
        return true;  
    }  
}
```

5、使用策略授权动作

通过 User 模型

Laravel 自带的 `User` 模型提供了两个方法用于授权动作：`can` 和 `cant`。`can` 方法接收你想要授权的动作和对应的模型作为参数。例如，下面的例子我们判断用户是否被授权更新给定的 `Post` 模型：

```
if ($user->can('update', $post)) {  
    //  
}
```

如果给定模型对应的策略已经注册，则 `can` 方法会自动调用相应的策略并返回布尔结果。如果给定模型没有任何策略被注册，`can` 方法将会尝试调用与动作名称相匹配的 Gate 闭包。

不依赖于模型的动作

有些动作比如 `create` 并不需要依赖给定模型实例，在这些场景中，可以传递一个类名到 `can` 方法，这个类名会在进行授权的时候用于判断使用哪一个策略：

```
use App\Post;  
  
if ($user->can('create', Post::class)) {  
    // Executes the "create" method on the relevant policy...  
}
```

通过中间件

Laravel 提供了一个可以在请求到达路由或控制器之前进行授权的中间件 — `Illuminate\Auth\Middleware\Authorize`，默认情况下，这个中间件在 `App\Http\Kernel` 类中被分配了一个 `can` 别名，下面我们来探究如何使用 `can` 中间件授权用户更新博客文章动作：

```
use App\Post;

Route::put('/post/{post}', function (Post $post) {
    // The current user may update the post...
})->middleware('can:update,post');
```

在这个例子中，我们传递了两个参数给 `can` 中间件，第一个是我们想要授权的动作名称，第二个是我们想要传递给策略方法的路由参数。在这个例子中，由于我们使用了隐式模型绑定，`Post` 模型将会被传递给策略方法，如果没有对用户进行给定动作的授权，中间件将会生成并返回一个状态码为 `403` 的 HTTP 响应。

不依赖于模型的动作

同样，对那些不需要传入模型实例的动作如 `create`，需要传递类名到中间件，类名将会在授权动作的时候用于判断使用哪个策略：

```
Route::post('/post', function () {
    // The current user may create posts...
})->middleware('can:create,App\Post');
```

通过控制器辅助函数

除了提供给 `User` 模型的辅助函数， Laravel 还为继承自 `App\Http\Controllers\Controller` 基类的所有控制器提供了 `authorize` 方法，和 `can` 方法类似，该方法接收你想要授权的动作名称以及相应模型实例作为参数，如果动作没有被授权，`authorize` 方法将会抛出 `Illuminate\Auth\Access\AuthorizationException`， Laravel 默认异常处理器将会将其转化为状态码为 `403` 的 HTTP 响应：

```
<?php

namespace App\Http\Controllers;

use App\Post;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PostController extends Controller
{
    /**
     * Update the given blog post.
     *
     * @param Request $request
     * @param Post $post
     * @return Response
     */
    public function update(Request $request, Post $post)
```

```
{  
    $this->authorize('update', $post);  
  
    // The current user can update the blog post...  
}  
}
```

不依赖模型的动作

和之前讨论的一样，类似 `create` 这样的动作不需要传入模型实例参数，在这些场景中，可以传递类名给 `authorize` 方法，该类名将会在授权动作时判断使用哪个策略：

```
/**  
 * Create a new blog post.  
 *  
 * @param Request $request  
 * @return Response  
 */  
  
public function create(Request $request)  
{  
    $this->authorize('create', Post::class);  
  
    // The current user can create blog posts...  
}
```

通过 Blade 模板

编写 Blade 模板的时候，你可能想要在用户被授权特定动作的情况下才显示对应的视图模板部分，

例如，你可能想要在用户被授权更新权限的情况下才显示更新表单。在这种情况下，你可以使

用 `@can` 和 `@cannot` 指令：

```
@can('update', $post)
    <!-- The Current User Can Update The Post -->
@endcan

@cannot('update', $post)
    <!-- The Current User Can't Update The Post -->
@endcannot
```

这种写法可看作是 `@if` 和 `@unless` 语句的缩写，上面的 `@can` 和 `@cannot` 语句与下面的语句等价：

```
@if (Auth::user()->can('update', $post))
    <!-- The Current User Can Update The Post -->
@endif

@unless (Auth::user()->can('update', $post))
    <!-- The Current User Can't Update The Post -->
@endunless
```

不依赖模型的动作

和其它授权方法一样，如果授权动作不需要传入模型实例的情况下可以传递类名

给 `@can` 和 `@cannot` 指令：

```
@can('create', Post::class)
    <!-- The Current User Can Create Posts -->
```

```
@endcan

@cannot('create', Post::class)
    <!-- The Current User Can't Create Posts -->
@endcannot
```

8.3 密码重置

1、简介

想要快速实现该功能？只需要在新安装的 [Laravel](#) 应用下运行 `php artisan make:auth`，然后在浏览器中访问 `http://your-app.dev/register` 或者其他分配给应用的 URL，该命令会生成[用户登录注册](#)所需要的所有东西，包括[密码重置](#)！

大多数 web 应用提供了用户重置密码的功能，Laravel 提供了便利方法用于发送密码重置链接及实现密码重置逻辑，而不需要你在每个应用中重新实现。

注：在使用 Laravel 提供的密码重置功能之前，[User](#) 模型必须使用了 [Illuminate\Notifications\Notifiable](#) trait。

2、数据库考量

开始之前，先验证 [App\User](#) 模型实现了 [Illuminate\Contracts\Auth\CanResetPassword](#) 契约。

当然，Laravel 自带的 [App\User](#) 模型已经实现了该接口，并使用 [Illuminate\Auth\Passwords\CanResetPassword](#) trait 来包含实现该接口需要的方法。

生成重置令牌表迁移

接下来，用来存储密码重置令牌的表必须被创建，Laravel 已经自带了这张表的迁移，就存放

在 `database/migrations` 目录。所有，你所要做的仅仅是运行迁移：

```
php artisan migrate
```

3、路由

Laravel 自带了 `Auth\ForgotPasswordController` 和 `Auth\ResetPasswordController`，分别用于发送密码重置链接[邮件](#)和重置用户密码功能。重置密码所需的路由都已经通过 `make:auth` 命令自动生成了：

```
php artisan make:auth
```

4、视图

和路由一样，重置密码所需的视图文件也通过 `make:auth` 命令一并生成了，这些视图文件位于 `resources/views/auth/passwords` 目录下，你可以按照所需对生成的文件进行相应修改。

5、重置密码后

定义好重置用户密码路由和视图后，只需要在浏览器中通过 `/password/reset` 访问这个入口路由。框架自带的 `PasswordController` 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。

密码被重置后，用户将会自动登录到应用并重定向到 `/home`。你可以通过定义 `ResetPasswordController` 的 `redirectTo` 属性来自定义密码重置成功后的跳转链接：

```
protected $redirectTo = '/dashboard';
```

注：默认情况下，密码重置令牌一小时内有效，你可以通过修改 `config/auth.php` 文件中的选项

`expire` 来改变有效时间。

6、自定义

自定义认证 Guard

在配置文件 `auth.php` 中，可以配置多个“guards”，以便用于实现基于多用户表的独立认证，你可以通过重写内置的 `ResetPasswordController` 控制器上的 `guard` 方法来使用你所选择的 `guard`，该方法将会返回一个 `guard` 实例：

```
use Illuminate\Support\Facades\Auth;

protected function guard()
{
    return Auth::guard('guard-name');
}
```

自定义密码 broker

在配置文件 `auth.php` 中，可以配置多个密码，以便用于重置多个用户表的密码 `broker`，同样，你可以通过重写自带的 `ForgotPasswordController` 和 `ResetPasswordController` 控制器中的 `broker` 方法来使用你所选择的 `broker`：

```
use Illuminate\Support\Facades\Password;

/**
```

```
* Get the broker to be used during password reset.  
*  
* @return PasswordBroker  
*/  
  
protected function broker()  
{  
    return Password::broker('name');  
}
```

自定义密码重置邮件

你可以很方便地编辑发送密码重置链接给用户的通知类实现自定义密码重置邮件，要实现这一功能，需要重写 `User` 模型上的 `sendPasswordResetNotification` 方法，在这个方法中，可以使用任何你所喜欢的通知类发送通知，该方法接收的第一个参数是密码重置 `$token`：

```
/**  
 * Send the password reset notification.  
 *  
 * @param string $token  
 * @return void  
 */  
public function sendPasswordResetNotification($token)  
{  
    $this->notify(new ResetPasswordNotification($token));  
}
```

8.4 API 认证 (Passport)

1、简介

Laravel 通过传统的登录表单已经让用户 认证 变得很简单，但是 API 怎么办？API 通常使用 token 进行认证并且在请求之间不维护 session 状态。Laravel 使用 Laravel Passport 让 API 认证变得轻

而易举，Passport 基于 Alex Bilbie 维护的 [League OAuth2 server](#)，可以在数分钟内为 Laravel 应用提供完整的 [OAuth2 服务器实现](#)。

注：本[文档](#)假设你已经很熟悉 OAuth2，如果你对 OAuth2 一无所知，那么在开始学习文文档之前，先要去熟悉 OAuth2 的一些术语和特性（参考阮一峰博客：[理解 OAuth 2.0](#)）。

2、安装

使用 Composer 包管理器安装 Passport：

```
composer require laravel/passport
```

接下来，在配置文件 `config/app.php` 的 `providers` 数组中注册 Passport 服务提供者：

```
Laravel\Passport\PassportServiceProvider::class,
```

Passport 服务提供着为框架注册了自己的数据库迁移目录，所以在注册之后需要迁移数据库，

Passport 迁移将会为应用生成用于存放客户端和访问令牌的数据表：

```
php artisan migrate
```

接下来，需要运行 `passport:install` 命令，该命令将会创建生成安全访问令牌（token）所需的加密键，此外，该命令还会创建“personal access”和“password grant”客户端用于生成访问令牌：

```
php artisan passport:install
```

运行完这个命令后，添加 `Laravel\Passport\HasApiTokens` trait 到 `App\User` 模型，该 trait 将会为模型类提供一些辅助函数用于检查认证用户的 token 和 scope：

```
<?php

namespace App;

use Laravel\Passport\HasApiTokens;
use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;
}
```

接下来，你需要在 `AuthServiceProvider` 的 `boot` 方法中调用 `Passport::routes` 方法，该方法将会注册发布/撤销访问令牌、客户端以及私人访问令牌所必需的[路由](#)：

```
<?php

namespace App\Providers;

use Laravel\Passport\Passport;
use Illuminate\Support\Facades\Gate;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider
```

```
er;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        'App\Model' => 'App\Policies\ModelPolicy',
    ];

    /**
     * Register any authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        Passport::routes();
    }
}
```

最后，在配置文件 `config/auth.php` 中，需要设置 `api` 认证 guard 的 `driver` 选项为 `passport`。

这将告知应用在认证输入的 API 请求时使用 Passport 的 `TokenGuard`：

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],
    'api' => [
        'driver' => 'passport',
        'provider' => 'users',
    ],
],
```

前端快速上手

注：为了使用 Passport 的 Vue 组件，前端 [JavaScript](#) 必须使用 Vue 框架，这些组件同时也使用了 Bootstrap CSS 框架。不过，即使你不使用这些工具，这些组件同样可以为你实现自己的前端组件提供有价值的参考。

Passport 附带了 [JSON](#) API 以便用户创建客户端和私人访问令牌（access token）。不过，考虑到编写前端代码与这些 API 交互是一件很花费时间的事，Passport 还预置了 Vue 组件作为示例以供使用（或者作为自己实现的参考）。

要发布 Passport Vue 组件，可以使用 `vendor:publish` 命令：

```
php artisan vendor:publish --tag=passport-components
```

发布后的组件位于 `resources/assets/js/components` 目录下，组件发布之后，还需要将它们注册到 `resources/assets/js/app.js` 文件：

```
Vue.component(
  'passport-clients',
  require('./components/passport/Clients.vue')
);

Vue.component(
  'passport-authorized-clients',
  require('./components/passport/AuthorizedClients.vue')
);

Vue.component(
  'passport-personal-access-tokens',
  require('./components/passport/PersonalAccessTokens.vue')
);
```

注册完成后，就可以将它们放到应用的某个模板中以便创建客户端和私人访问令牌：

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

3、配置

令牌生命周期

默认情况下，Passport 颁发的访问令牌（access token）是长期有效的，如果你想要配置更短的令牌生命周期，可以使用 `tokensExpireIn` 和 `refreshTokensExpireIn` 方法，这些方法需要在 `AuthServiceProvider` 的 `boot` 方法中调用：

```
/**
 * 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));
}
```

精简撤销令牌

默认情况下，Passport 不会从数据库删除撤销的访问令牌，随着时间的推移，在数据库中会累积

大量的撤销令牌，如果你希望 Passport 可以自动删除撤销的令牌，可以在 `AuthServiceProvider` 的 `boot` 方法中调用 `pruneRevokedTokens` 方法：

```
use Laravel\Passport\Passport;  
  
Passport::pruneRevokedTokens();
```

该方法不会立即删除所有的撤销令牌，而是当用户请求一个新的访问令牌或者刷新已存在令牌的时候才会删除撤销的令牌。

4、颁发访问令牌

通过[授权码](#)使用 OAuth2 是大多数开发者属性的方式。使用授权码的时候，客户端应用会将用户重定向到你的服务器，服务器将会通过或拒绝颁发访问令牌到客户端的请求。

管理客户端

首先，开发构建和你的应用 API 交互的应用时，需要通过创建一个“客户端”以便将他们的应用注册到你的应用。通常，这一过程包括提供应用的名称以及用户授权请求通过后重定向到的 URL。

`passport:client` 命令

创建客户端最简单的方式就是使用 Artisan 命令 `passport:client`，该命令可用于创建你自己的客户端以方便测试 OAuth2 功能。当你运行 `client` 命令时，Passport 会提示你关于客户端的更多信息，并且为你提供 client ID 和 secret：

```
php artisan passport:client
```

JSON API

由于用户不能使用 client 命令，Passport 提供了一个 JSON API 用于创建客户端，这省去了你手动编写控制器用于创建、更新以及删除客户端的麻烦。

不过，你需要配对 Passport 的 JSON API 和自己的前端以便为用户提供一个可以管理他们自己客户端的后台，下面，我们来回顾下所有用于管理客户端的 API，为了方便，我们将会使用 Vue 来演示发送 HTTP 请求到 API：

注：如果你不想要自己实现整个客户端管理前端，可以使用[前端快速上手教程](#)在数分钟内拥有完整功能的前端。

`GET /oauth/clients`

这个路由为认证用户返回所有客户端，这在展示用户客户端列表时很有用，可以让用户很容易编辑或删除客户端：

```
this.$http.get('/oauth/clients')
  .then(response => {
    console.log(response.data);
  });
}
```

`POST /oauth/clients`

这个路由用于创建新的客户端，要求传入两个数据：客户端的 `name` 和 `redirect` URL，`redirect` URL 是用户授权请求通过或拒绝后重定向到的位置。

当客户端被创建后，会附带一个 client ID 和 secret，这两个值会在请求访问令牌时用到。客户端创建路由会返回新的客户端实例：

```
const data = {
    name: 'Client Name',
    redirect: 'http://example.com/callback'
};

this.$http.post('/oauth/clients', data)
    .then(response => {
        console.log(response.data);
    })
    .catch (response => {
        // List errors on response...
    });
});
```

`PUT /oauth/clients/{client-id}`

这个路由用于更新客户端，要求传入两个参数：客户端的 `name` 和 `redirect` URL。`redirect` URL 是用户授权请求通过或拒绝后重定向到的位置。该路由将会返回更新后的客户端实例：

```
const data = {
    name: 'New Client Name',
    redirect: 'http://example.com/callback'
};

this.$http.put('/oauth/clients/' + clientId, data)
    .then(response => {
        console.log(response.data);
    })
});
```

```
.catch (response => {
    // List errors on response...
});
```

`DELETE /oauth/clients/{client-id}`

这个路由用于删除客户端：

```
this.$http.delete('/oauth/clients/' + clientId)
.then(response => {
    //
});
```

请求令牌

授权重定向

客户端被创建后，开发者就可以使用相应的 client ID 和 secret 从应用请求授权码和访问令牌。首

先，消费者应用要生成一个重定向请求到应用的 `/oauth/authorize` 路由：

```
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'code',
        'scope' => '',
    ]);

    return redirect('http://your-app.com/oauth/authorize?'.$query);
```

```
});
```

注：记住，`/oauth/authorize` 路由已经通过 `Passport::routes` 方法定义了，不需要手动定义这个路由。

通过请求

接收授权请求的时候，Passport 会自动显示一个视图模板给用户从而允许他们通过或拒绝授权请求，如果用户通过请求，就会被重定向回消费者应用指定的 `redirect_uri`，这个 `redirect_uri` 必须和客户端创建时指定的 `redirect` URL 一致。

如果你想要自定义授权通过界面，可以使用 Artisan 命令 `vendor:publish` 发布 Passport 的视图模板，发布后的视图位于 `resources/views/vendor/passport`：

```
php artisan vendor:publish --tag=passport-views
```

将授权码转化为访问令牌

如果用户通过了授权请求，会被重定向回消费者应用。消费者接下来会发送一个 POST 请求到应用来请求访问令牌。这个请求应该包含用户通过授权请求时指定的授权码。在这个例子中，我们会使用 Guzzle HTTP 库来生成 POST 请求：

```
Route::get('/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' => env('OAUTH_CLIENT_ID'),
            'client_secret' => env('OAUTH_CLIENT_SECRET'),
            'code' => $request->query('code')
        ]
    ]);
    $data = json_decode($response->getBody(), true);
    dd($data);
});
```

```
'client_id' => 'client-id',
'client_secret' => 'client-secret',
'redirect_uri' => 'http://example.com/callback',
'code' => $request->code,
],
]);

return json_decode((string) $response->getBody(), true);
});
```

`/oauth/token` 路由会返回一个包含 `access_token`、`refresh_token` 和 `expires_in` 属性的 JSON 响应。`expires_in` 属性包含访问令牌的过期时间 (s)。

注：和 `/oauth/authorize` 路由一样，`/oauth/token` 路由已经通过 `Passport::routes` 方法定义过了，不需要手动定义这个路由。

刷新令牌

如果应用颁发的是短期有效的访问令牌，那么用户需要通过访问令牌颁发时提供的 `refresh_token` 刷新访问令牌，在这个例子中，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);
```

`/oauth/token` 路由会返回一个包含 `access_token`、`refresh_token` 和 `expires_in` 属性的 JSON 响应，同样，`expires_in` 属性包含访问令牌过期时间 (s)。

5、密码发放令牌

OAuth2 密码发放允许你的其他第一方客户端，例如移动应用，使用邮箱地址/用户名+密码获取访问令牌。这使得你可以安全地颁发访问令牌给第一方客户端而不必要求你的用户走整个 OAuth2 授权码重定向流程。

创建一个密码发放客户端

在应用可以通过密码发放颁发令牌之前，需要创建一个密码发放客户端，你可以通过使用带 `--password` 选项的 `passport:client` 命令来实现。如果你已经运行了 `passport:install` 命令，则不必再运行这个命令：

```
php artisan passport:client --password
```

请求令牌

创建完密码发放客户端后，可以通过发送 POST 请求到 `/oauth/token` 路由 (带上用户邮箱地址和

密码参数 获取访问令牌。这个路由已经通过 `Passport::routes` 方法注册过了 不需要手动定义。

如果请求成功，就可以从服务器返回的 JSON 响应中获取 `access_token` 和 `refresh_token`：

```
$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);
```

注：记住，访问令牌默认长期有效，不过，如果需要的话你也可以配置访问令牌的最长生命周期。

请求所有域

使用密码发放的时候，你可能想要授权应用所支持的所有域的令牌，这可以通过请求`*域`来实现。

如果你请求的是`*域` 则令牌实例上的 `can` 方法总是返回 `true` 这个域只会分配给使用 `password` 发放的令牌：

```
$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'password',
        'client_id' => 'client-id',
        'username' => 'taylor@laravel.com',
        'password' => 'my-password',
        'scope' => '*',
    ],
]);
```

6、私人访问令牌

有时候，你的用户可能想要颁发访问令牌给自己而不走典型的授权码重定向流程。允许用户通过应用的 UI 颁发令牌给自己在用户体验你的 API 或者作为更简单的颁发访问令牌方式时会很有用。

注：私人访问令牌总是一直有效的，它们的生命周期在使用 `tokensExpireIn` 或 `refreshTokensExpireIn` 方法时不会修改。

创建一个私人访问客户端

在你的应用可以颁发私人访问令牌之前，需要创建一个私人的访问客户端。你可以通过带 `--personal` 选项的 `passport:client` 命令来实现，如果你已经运行过了 `passport:install` 命令，则不必再运行此命令：

```
php artisan passport:client --personal
```

管理私人访问令牌

创建好私人访问客户端之后，就可以使用 `User` 模型实例上的 `createToken` 方法为给定用户颁发令牌。`createToken` 方法接收令牌名称作为第一个参数，以及一个可选的域数组作为第二个参数：

```
$user = App\User::find(1);

// Creating a token without scopes...
$token = $user->createToken('Token Name')->accessToken;

// Creating a token with scopes...
$token = $user->createToken('My Token', ['place-orders'])->accessToken;
```

JSON API

Passport 还提供了一个 JSON API 用于管理私人访问令牌，你可以将其与自己的前端配对以便为用户提供管理私人访问令牌的后台。下面，我们来回顾用于管理私人访问令牌的所有 API。为了方便起见，我们使用 Vue 来演示发送 HTTP 请求到 API。

注：如果你不想要实现自己的私人访问令牌前端，可以使用[前端快速上手教程](#)在数分钟内打造拥有完整功能的前端。

`GET /oauth/scopes`

这个路由会返回应用所定义的所有域。你可以使用这个路由来列出用户可以分配给私人访问令牌的所有域：

```
this.$http.get('/oauth/scopes')
  .then(response => {
```

```
        console.log(response.data);
    });
}
```

GET /oauth/personal-access-tokens

这个路由会返回该认证用户所创建的所有私人访问令牌，这在列出用户的所有令牌以便编辑或删除时很有用：

```
this.$http.get('/oauth/personal-access-tokens')
.then(response => {
    console.log(response.data);
});
```

POST /oauth/personal-access-tokens

这个路由会创建一个新的私人访问令牌，该路由要求传入两个参数：令牌的 `name` 和需要分配到这个令牌的 `scopes`：

```
const data = {
    name: 'Token Name',
    scopes: []
};

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}`

这个路由可以用于删除私人访问令牌：

```
this.$http.delete('/oauth/personal-access-tokens/' + tokenId);
```

7、路由保护

通过中间件

Passport 提供了一个认证 guard 用于验证输入请求的访问令牌，当你使用 `passport` 驱动配置

好 `api` guard 后，只需要在所有路由上指定需要传入有效访问令牌的 `auth:api` 中间件即可：

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

传递访问令牌

调用被 Passport 保护的路由时，应用的 API 消费者需要在请求的 `Authorization` 头中指定它们的

访问令牌作为 Bearer 令牌。例如：

```
$response = $client->request('GET', '/api/user', [
    'headers' => [
        'Accept' => 'application/json',
        'Authorization' => 'Bearer ' . $accessToken,
```

```
],  
]);
```

8、令牌作用域

定义作用域

作用域 (Scope) 允许 API 客户端在请求账户授权的时候请求特定的权限集合。例如，如果你在构建一个电子商务应用，不是所有的 API 消费者都需要下订单的能力，取而代之地，你可以让这些消费者只请求访问订单物流状态的权限，换句话说，作用域允许你的应用用户限制第三方应用自身可以执行的操作。

你可以在 `AuthServiceProvider` 的 `boot` 方法中使用 `Passport::tokensCan` 方法定义 API 的作用域。 `tokensCan` 方法接收作用域名称数组和作用域描述，作用域描述可以是任何你想要在授权通过页面展示给用户的东西：

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

分配作用域到令牌

请求授权码时

当使用授权码请求访问令牌时，消费者应该指定他们期望的作用域作为 `scope` 查询字符串参数，`scope` 参数是通过空格分隔的作用域列表：

```
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'code',
        'scope' => 'place-orders check-status',
    ]);

    return redirect('http://your-app.com/oauth/authorize?'.$query);
});
```

颁发私人访问令牌时

如果你使用 `User` 模型的 `createToken` 方法颁发私人访问令牌，可以传递期望的作用域数组作为该方法的第二个参数：

```
$token = $user->createToken('My Token', ['place-orders'])->accessToken;
```

检查作用域

Passport 提供两个可用于验证输入请求是否经过已发放作用域的令牌认证的中间件。开始之前，添加下面的中间件到 `app/Http/Kernel.php` 文件的 `$routeMiddleware` 属性：

```
'scopes' => \Laravel\Passport\Http\Middleware\CheckScopes::class,  
'scope' => \Laravel\Passport\Http\Middleware\CheckForAnyScope::class,
```

检查所有作用域

`scopes` 中间件会分配给一个用于验证输入请求的访问令牌拥有所有列出作用域的路由：

```
Route::get('/orders', function () {  
    // Access token has both "check-status" and "place-orders" scopes...  
})->middleware('scopes:check-status,place-orders');
```

检查任意作用域

`scope` 中间件会分配给一个用于验证输入请求的访问令牌拥有至少一个列出作用域的路由：

```
Route::get('/orders', function () {  
    // Access token has either "check-status" or "place-orders" scope...  
})->middleware('scope:check-status,place-orders');
```

检查令牌实例上的作用域

当一个访问令牌认证过的请求进入应用后，你仍然可以使用经过认证的 `User` 实例上的 `tokenCan` 方法来检查这个令牌是否拥有给定作用域：

```
use Illuminate\Http\Request;  
  
Route::get('/orders', function (Request $request) {  
    if ($request->user()->tokenCan('place-orders')) {  
        //  
    }  
});
```

```
    }  
});
```

9、使用 JavaScript 消费 API

构建 API 时，能够从你的 JavaScript 应用消费你自己的 API 非常有用。这种 API 开发方式允许你自己的应用消费你和其他人分享的同一个 API，这个 API 可以被你的 Web 应用消费，也可以被你的移动应用分配，还可以被第三方应用消费，以及任何你可能发布在多个包管理器上的 SDK 消费。

通常，如果你想要从你的 JavaScript 应用消费自己的 API，需要手动发送访问令牌到应用并在应用的每一个请求中传递它。不过，Passport 提供了一个中间件用于处理这一操作。你所需要做的只是添加这个中间件 `CreateFreshApiToken` 到 `web` 中间件组：

```
'web' => [  
    // Other middleware...  
    \Laravel\Passport\Http\Middleware\CreateFreshApiToken::class,  
],
```

这个 Passport 中间件将会附加 `laravel_token` cookie 到输出响应，这个 cookie 包含加密过的 JWT，Passport 将使用这个 JWT 来认证来自 JavaScript 应用的 API 请求，现在，你可以发送请求到应用的 API，而不必显示传递访问令牌：

```
this.$http.get('/user')  
.then(response => {
```

```
    console.log(response.data);  
});
```

使用认证的这个方法时，你需要在每个请求中通过 `X-CSRF-TOKEN` 头发送 CSRF 令牌，如果你使用的是框架提供的默认 Vue 配置的话，Laravel 会自动发送这个头：

```
Vue.http.interceptors.push((request, next) => {  
    request.headers['X-CSRF-TOKEN'] = Laravel.csrfToken;  
  
    next();  
});
```

注：如果你使用的是其它 JavaScript 框架，需要确保每个请求都被配置为发送这个请求头。

8.5 加密

1、简介

Laravel 的加密器使用 [OpenSSL](#) 来提供 AES-256 和 AES-128 加密。强烈建议使用 Laravel 自带的加密设置，不要尝试推出自己“土生土长”的加密算法。所有 Laravel 加密过的值都使用消息授权码（MAC）进行签名以便底层值一经加密就不能修改。

2、配置

在使用 Laravel 的加密器之前，必须在配置文件 `config/app.php` 中设置 `key` 选项为 32 位随机字符串。可以使用 `php artisan key:generate` 命令来生成这个 key，该 Artisan 命令会使用 PHP 的

安全随机字节生成器来构建 key 的值。如果这个值没有被设置，所有 Laravel 加密过的值都是不安全的。

3、使用加密器

加密

你可以使用辅助函数 `encrypt` 对数据进行加密，所有加密值都使用 OpenSSL 和 `AES-256-CBC` 密码 (cipher) 进行加密。此外，所有加密值都通过一个消息认证码 (MAC) 来进行签名以防止对加密字符串的任何修改。

```
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller
{
    /**
     * Store a secret message for the user.
     *
     * @param Request $request
     * @param int $id
}
```

```
* @return Response
*/
public function storeSecret(Request $request, $id)
{
    $user = User::findOrFail($id);

    $user->fill([
        'secret' => encrypt($request->secret)
    ])->save();
}

}
```

注：加密值在加密期间都会经过 `serialize`，从而允许对对象和数组的加密。因此，非 PHP 客户端接收的加密数据需要进行 `unserialize`。

解密

你可以使用辅助函数 `decrypt` 对加密数据进行解密。如果该值不能被解密，例如 MAC 无效，将会抛出一个 `Illuminate\Contracts\Encryption\DecryptException` 异常：

```
use Illuminate\Contracts\Encryption\DecryptException;

try {
    $decrypted = decrypt($encryptedValue);
} catch (DecryptException $e) {
    //
}
```

8.6 哈希 (Hashing)

1、简介

Laravel 的 Hash 为存储用户密码提供了安全的 Bcrypt 哈希算法。如果你正在使用 Laravel 应用自带的 `LoginController` 和 `RegisterController` 控制器，它们将会自动在注册和认证时使用该 Bcrypt。

注：Bcrypt 是散列密码的绝佳选择，因为其“工作因子”是可调整的，这意味着随着硬件功能的提升，生成哈希所花费的时间也会增加。

2、基本使用

可以调用 Hash 门面上的 make 方法对存储密码进行哈希：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use App\Http\Controllers\Controller;

class UpdatePasswordController extends Controller
{
    /**
     * Update the password for the user.
     *
     * @param Request $request
     * @return Response
    */
}
```

```
public function update(Request $request)
{
    // Validate the new password length...

    $request->user()->fill([
        'password' => Hash::make($request->newPassword)
    ])->save();
}

}
```

通过哈希验证密码

`check` 方法允许你验证给定原生字符串和给定哈希是否相等，不过，如果你在使用 Laravel 自带的 `LoginController` (详见[用户认证](#)一节)，就不需要再直接使用该方法，因为这个控制器自动调用了该方法：

```
if (Hash::check('plain-text', $hashedPassword)) {
    // 密码匹配...
}
```

检查密码是否需要被重新哈希

`needsRehash` 方法允许你判断哈希计算器使用的工作因子在上次密码被哈希后是否发生改变：

```
if (Hash::needsRehash($hashed)) {
    $hashed = Hash::make('plain-text');
}
```

9. 综合话题

9.1 事件广播

1、简介

在很多现代 Web 应用中，Web 套接字 ([WebSocket](#)s) 被用于实现实时更新的用户接口。当一些数据在服务器上被更新，通常一条消息通过 websocket 连接被发送给客户端处理。这为我们提供了一个更强大的、更有效的选择来持续拉取应用的更新。

为帮助你构建这样的应用，[Laravel](#) 让通过 websocket 连接[广播事件](#)变得简单。广播 Laravel 事件允许你在服务端和客户端 JavaScript 框架之间共享同一事件名。

注：在深入了解事件广播之前，确保你已经阅读并理解 Laravel 事件与监听器相关[文档](#)。

配置

应用的所有事件广播配置选项都存放在 `config/broadcasting.php` 配置文件中。Laravel 开箱支持多种广播驱动：[Pusher](#)、[Redis](#) 以及一个服务于本地开发和调试的日志驱动。此外，还提供了一个 `null` 驱动用于完全禁止事件广播。每一个驱动在 `config/broadcasting.php` 配置文件中都有一个配置示例。

广播服务提供者

在广播任意事件之前，首先需要注册 `App\Providers\BroadcastServiceProvider`。在新安装的 Laravel 应用中，你只需要取消 `config/app.php` 配置文件中 `providers` 数组内对应服务提供者之前的注释即可。该提供者允许你注册广播认证路由和回调。

CSRF 令牌

[Laravel Echo](#) 需要访问当前 session 的 CSRF 令牌(token) , 如果有效的话 , [Echo](#) 将会从 JavaScript 变量 `Laravel.csrfToken` 中获取令牌。如果你运行过 Artisan 命令 `make:auth` 的话 , 该对象定义在 `resources/views/layouts/app.blade.php` 布局文件中。如果你没有使用这个布局 , 你可以在应用的 HTML 元素 `head` 中定义一个 `meta` 标签 :

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

驱动预备知识

[Pusher](#)

如果你准备通过 [Pusher](#) 广播事件 , 需要使用 Composer 包管理器安装对应的 Pusher PHP SDK :

```
composer require pusher/pusher-php-server
```

接下来 , 你需要在 `config/broadcasting.php` 配置文件中配置你的 Pusher 证书。一个配置好的 Pusher 示例已经包含在这个文件中 , 你可以按照这个模板进行修改 , 指定自己的 Pusher key、secret 和应用 ID 即可。

使用 Pusher 和 [Laravel Echo](#) 的时候 , 需要在安装某个 Echo 实例的时候指定 `pusher` 作为期望的广播 :

```
import Echo from "laravel-echo"

window.Echo = new Echo({
```

```
broadcaster: 'pusher',
key: 'your-pusher-key'
});
```

Redis

如果你使用 Redis 广播，需要安装 Predis 库：

```
composer require predis/predis
```

Redis 广播使用 Redis 的 pub/sub 功能进行广播；不过，你需要将其和能够接受 Redis 消息的 Websocket 服务器进行配对以便将消息广播到 Websocket 频道。

当 Redis 广播发布事件时，事件将会被发布到指定的频道上，传递的数据是一个 JSON 格式的字符串，其中包含了事件名称、数据明细 data、以及生成事件 socket ID 的用户。

Socket.IO

如果你想配对 Redis 广播和 Socket.IO 服务器，则需要在应用的 HTML 元素 head 中引入 Socket.IO JavaScript 库：

```
<script src="https://cdn.socket.io/socket.io-1.4.5.js"></script>
```

接下来，你需要使用 socket.io 连接器和 host 来实例化 Echo。例如，如果你的应用和 socket 服务器运行在了 app.dev 上，那么需要注意实例化 Echo：

```
import Echo from "laravel-echo"
window.Echo = new Echo({
```

```
broadcaster: 'socket.io',
host: 'http://app.dev:6001'
});
```

最后，需要运行一个与之兼容的 Socket.IO 服务器。Laravel 并未内置一个 Socket.IO 服务器实现，

不过，这里有一个第三方实现的 Socket.IO 驱动：[tlaverdure/laravel-echo-server](https://github.com/tlaverdure/laravel-echo-server)。

队列预备知识

在开始介绍广播事件之前，还需要配置并运行一个队列监听器。所有事件广播都通过队列任务来完成以便应用的响应时间不受影响。

2、概念概览

Laravel 的事件广播允许你使用基于驱动的 WebSockets 将服务器端端事件广播到客户端 JavaScript 应用。目前 Laravel 使用 Pusher 和 Redis 驱动，这些事件可以通过 JavaScript 包 Laravel Echo 在客户端被轻松消费。

事件通过“频道”进行广播，这些频道可以是公共的，也可以是私有的，应用的任何访问者都可以不经认证和[授权](#)注册到一个共有的频道，不过，想要注册到私有频道，用户必须经过认证和授权才能监听该频道。

使用示例应用

在深入了解每个事件广播组件之前，让我们先通过一个电商网站作为示例对整体有个大致的了解。这里我们不会讨论 Pusher 和 Laravel Echo 的配置细节，这些将会放在文档的其它章节进行讨论。

在我们的应用中，假设我们有一个页面允许用户查看订单的物流状态，我们还假设当应用进行订

单状态更新处理时会触发一个 `ShippingStatusUpdated` 事件：

```
event(new ShippingStatusUpdated($update));
```

ShouldBroadcast 接口

当用户查看某个订单时，我们不希望他们必须刷新页面来查看更新状态。取而代之地，我们希望在创建时将更新广播到应用。因此，我们需要标记 `ShippingStatusUpdated` 事件实现

`ShouldBroadcast` 接口，这样，Laravel 就会在触发事件时广播该事件：

```
<?php

namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class ShippingStatusUpdated implements ShouldBroadcast
{
    //
}
```

`ShouldBroadcast` 接口要求事件类定义一个 `broadcastOn` 方法，该方法会返回事件将要广播的频道。事件类生成时一个空的方法存根已经定义，我们所要做的只是填充其细节。我们只想要订单的创建者能够察看状态更新，所以我们将这个事件广播在一个与订单绑定的私有频道上：

```
/**  
 * Get the channels the event should broadcast on.  
 *  
 * @return array  
 */  
public function broadcastOn()  
{  
    return new PrivateChannel('order.'.$this->update->order_id);  
}
```

授权频道

记住，用户必须经过授权之后才能监听私有频道。我们可以在 `BroadcastServiceProvider` 的 `boot` 方法中定义频道授权规则。在本例中，我们需要验证任意试图监听 `order.1` 频道的用户确实是订单的创建者：

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

`channel` 方法接收两个参数：频道的名称以及返回 `true` 或 `false` 以表明用户是否被授权可以监听频道的回调。

所有授权回调都接收当前认证用户作为第一个参数以及任意额外通配符参数作为随后参数，在本例中，我们使用 `*` 字符标识频道名称的 ID 部分是一个通配符。

监听事件广播

接下来要做的就是在 JavaScript 中监听事件。我们可以使用 Laravel Echo 来完成这一工作。首先，我们使用 `private` 方法注册私有频道。然后，我们使用 `listen` 方法监听 `ShippingStatusUpdated` 事件。默认情况下，所有事件的公共属性都要包含在广播事件中：

```
Echo.private('order.' + orderId)
  .listen('ShippingStatusUpdated', (e) => {
    console.log(e.update);
});
```

3、定义广播事件

要告诉 Laravel 给定事件应该被广播，需要在事件类上实现 `Illuminate\Contracts\Broadcasting\ShouldBroadcast` 接口，这个接口已经在 Laravel 框架生成的事件类中导入了，你只需要将其添加到事件即可。

`ShouldBroadcast` 接口要求你实现一个方法：`broadcastOn`。该方法应该返回一个事件广播频道或频道数组。这些频道必须是 `Channel`、`PrivateChannel` 或 `PresenceChannel` 的实例，`Channel` 频道表示任意用户可以注册的公共频道，而 `PrivateChannels` 或 `PresenceChannels` 则代表需要进行频道授权的私有频道：

```
<?php

namespace App\Events;

use Illuminate\Broadcasting\Channel;
```

```
use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class ServerCreated implements ShouldBroadcast
{
    use SerializesModels;

    public $user;

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

    /**
     * Get the channels the event should broadcast on.
     *
     * @return Channel|array
     */
    public function broadcastOn()

```

```
{  
    return new PrivateChannel('user.'.$this->user->id);  
}  
}
```

然后，你只需要正常触发这个事件即可。一旦事件被触发，队列任务会自动通过指定广播驱动广播该事件。

广播数据

如果某个事件被广播，其所有的 `public` 属性都会按照事件负载（payload）自动序列化和广播，从而允许你从 JavaScript 中访问所有 `public` 数据，因此，举个例子，如果你的事件有一个单独的包含 Eloquent 模型的 `$user` 属性，广播负载定义如下：

```
{  
    "user": {  
        "id": 1,  
        "name": "Patrick Stewart"  
        ...  
    }  
}
```

不过，如果你希望对广播负载有更加细粒度的控制，可以添加 `broadcastWith` 方法到事件，该方法会返回你想要通过事件广播的数据数组：

```
/**
```

```
* 获取广播数据
*
* @return array
*/
public function broadcastWith(){
    return ['user' => $this->user->id];
}
```

广播队列

默认情况下，每个广播事件都放置在配置文件 `queue.php` 中指定的默认队列连接中，你可以通过在事件类上定义一个 `broadcastQueue` 属性来自定义广播使用的队列。该属性需要指定广播时你想要使用的队列名称：

```
/**
 * The name of the queue on which to place the event.
 *
 * @var string
 */
public $broadcastQueue = 'your-queue-name';
```

4、授权频道

私有频道要求你授权当前认证用户可以监听的频道。这可以通过向 Laravel 发送包含频道名称的 HTTP 请求然后让应用判断该用户是否可以监听频道来实现。使用 Laravel Echo 的时候，授权注册到私有频道的 HTTP 请求会自动发送，不过，你也需要定义相应路由来响应这些请求。

定义授权路由

庆幸的是， Laravel 让定义响应频道授权请求的路由变得简单，在 Laravel 自带的 `BroadcastServiceProvider` 中，你可以看到 `Broadcast::routes` 方法的调用，该方法会注册 `/broadcasting/auth` 路由来处理授权请求：

```
Broadcast::routes();
```

`Broadcast::routes` 方法将会自动将路由放置到 `web` 中间件组，不过，你也可以传递一个路由属性数组到这个方法以便自定义分配的属性：

```
Broadcast::routes($attributes);
```

定义授权回调

接下来，我们需要定义执行频道授权的逻辑，和定义授权路由一样，这也是通过 `BroadcastServiceProvider` 中的 `boot` 方法来完成。在这个方法中，你可以使用 `Broadcast::channel` 方法来注册频道授权回调：

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

`channel` 方法接收两个参数：频道名称和返回 `true` 或 `false` 以标识用户是否授权可以监听该频道的回调。

所有授权回调都接收当前认证用户作为第一个参数以及任意额外通配符参数作为其他参数。在本例中，我们使用*字符来标识频道名称的 ID 部分是一个通配符。

5、广播事件

定义好事件并标记其实现 `ShouldBroadcast` 接口后，你所要做的就是使用 `event` 方法触发该事件。

事件分发器将会注意事件是否实现了 `ShouldBroadcast` 接口，如果是的话就将其放置到广播队列中：

```
event(new ShippingStatusUpdated($update));
```

只广播给其他人

构建使用事件广播的应用时，你还可以使用 `broadcast` 函数替代 `event` 函数，和 `event` 函数一样，`broadcast` 函数将事件分发到服务器端监听器：

```
broadcast(new ShippingStatusUpdated($update));
```

不过，`broadcast` 函数还暴露了 `toOthers` 方法以便允许你从广播接收者中排除当前用户：

```
broadcast(new ShippingStatusUpdated($update))->toOthers();
```

为了更好地理解 `toOthers` 方法，我们先假设有一个任务列表应用，在这个应用中，用户可以通过输入任务名称创建一个新的任务，而要创建一个任务，应用需要发送请求到 `/task`，在这里，会广播任务创建并返回一个 JSON 格式的新任务。当你的 JavaScript 应用从服务端接收到响应后，会

直接将这个新任务插入到任务列表：

```
this.$http.post('/task', task)
  .then((response) => {
    this.tasks.push(response.data);
  });
}
```

不过，还记得吗？我们还广播了任务创建，如果你的 JavaScript 应用正在监听这个事件以便添加任务到任务列表，就会在列表中出现重复任务：一个来自服务端，一个来自广播。

你可以通过使用 `toOthers` 方法来解决这个问题，该方法告知广播不要讲事件广播给当前用户。

配置

当你初始化 Laravel Echo 实例的时候，需要给连接分配一个 socket ID。如果你使用的是 Vue 和 Vue 资源，这个 socket ID 会以 `X-Socket-ID` 头的方式自动添加到每个输出请求，Laravel 会从请求头中解析这个 socket ID 并告知广播不要广播到带有这个 socket ID 的连接。

如果你没有使用 Vue 和 Vue 资源，则需要手动配置 JavaScript 应用发送 `X-Socket-ID` 请求头。你可以使用 `Echo.socketId` 方法获取这个 socket ID：

```
var socketId = Echo.socketId();
```

6、接收广播

安装 Laravel Echo

Laravel Echo 是一个 JavaScript 库，有了它之后，注册到频道监听 Laravel 广播的事件将变得轻而

易举。你可以通过 NPM 包管理器安装 Echo，在本例中，我们还会安装 `pusher-js` 包，因为我们将会使用 Pusher 进行广播：

```
npm install --save laravel-echo pusher-js
```

创建好 Echo 之后，就可以在应用的 JavaScript 中创建一个新的 Echo 实例，做这件事的最佳位置

当然是在 Laravel 自带的 `resources/assets/js/bootstrap.js` 文件的底部：

```
import Echo from "laravel-echo"

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: 'your-pusher-key'
});
```

创建一个使用 `pusher` 连接器的 Echo 实例时，还可以指定一个 `cluster` 以及连接是否需要加密：

```
window.Echo = new Echo({
    broadcaster: 'pusher',
    key: 'your-pusher-key',
    cluster: 'eu',
    encrypted: true
});
```

监听事件

安装并初始化 Echo 之后，就可以开始监听事件广播了。首先，使用 `channel` 方法获取一个频道实例，然后调用 `listen` 方法监听指定事件：

```
Echo.channel('orders')
    .listen('OrderShipped', (e) => {
        console.log(e.order.name);
    });
}
```

如果你想要监听一个私有频道上的事件，可以使用 `private` 方法，你仍然可以在其后调用 `listen` 方法在单个频道监听多个事件：

```
Echo.private('orders')
    .listen(...)
    .listen(...)
    .listen(...);
```

命名空间

你可能已经注意到在上述例子中我们并没有指定事件类的完整命名空间，这是因为 Echo 会默认事件都位于 `App\Events` 命名空间下。不过，你可以在实例化 Echo 的时候通过传递配置选项 `namespace` 来配置根命名空间：

```
window.Echo = new Echo({
```

```
broadcaster: 'pusher',
key: 'your-pusher-key',
namespace: 'App.Other.Namespace'
});
```

另外，使用 Echo 注册事件的时候你可以在事件类前面加上 `.前缀`，这样你就可以指定完整的类名了：

```
Echo.channel('orders')
.listen('.Namespace.Event.Class', (e) => {
    //
});
```

7、存在频道（Presence Channel）

存在频道构建于私有频道之上，并且暴露了额外功能：获知谁注册到频道。基于这一点，我们可以构建强大的、协作的应用功能，例如当其他用户访问同一个页面时 [通知](#) 当前用户。

授权存在频道

所有存在频道同时也是私有频道，因此，用户必须被授权访问权限。不过，当定义存在频道的授权回调时，如果用户被授权加入频道不要返回 `true`，取而代之地，你应该返回关于该用户数据的数组。

授权回调返回的数据在 JavaScript 应用的存在频道事件监听器中使用，如果用户没有被授权加入存在频道，应该返回 `false` 或 `null`：

```
Broadcast::channel('chat.*', function ($user, $roomId) {
    if ($user->canJoinRoom($roomId)) {
        return ['id' => $user->id, 'name' => $user->name];
    }
});
```

加入授权频道

要加入存在频道，可以使用 Echo 的 `join` 方法，`join` 方法会返回一个 `PresenceChannel` 实现，并暴露 `listen` 方法，从而允许你注册到 `here`、`joining` 和 `leaving` 事件：

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

`here` 回调会在频道加入成功后立即执行，并接收一个包含所有其他注册到该频道的用户信息数组。`joining` 方法会在新用户加入频道时执行，`leaving` 方法则在用户离开频道时执行。

广播到存在频道

存在频道可以像公共或私有频道一样接收事件，以聊天室为例，我们可能想要广播 `NewMessage` 事件到房间的存在频道，要实现这个，需要从事件的 `broadcastOn` 方法返回 `PresenceChannel` 实例：

```
/*
 * Get the channels the event should broadcast on.
 *
 * @return Channel|array
 */
public function broadcastOn()
{
    return new PresenceChannel('room.' . $this->message->room_id);
}
```

和公共或私有频道一样，存在频道事件可以使用 `broadcast` 函数进行广播。和其他事件一样，你可以使用 `toOthers` 方法从所有接收广播的用户中排除当前用户：

```
broadcast(new NewMessage($message));
broadcast(new NewMessage($message))->toOthers();
```

你可以通过 Echo 的 `listen` 方法监听加入事件：

```
Echo::join('chat.' + roomId)
    .here(...)
    .joining(...)
```

```
.leaving(...)

.listen('NewMessage', (e) => {
    //
});
```

8、通知

通过配对事件广播和通知，JavaScript 应用可以在事件发生时无需刷新当前页面接收新的通知。

首先，确保你已经通读广播通知频道文档。

配置好使用广播频道的通知后，可以通过使用 Echo 的 `notification` 方法监听广播事件，记住，频道名称应该和接收通知的类名保持一致：

```
Echo.private('App.User.' + userId)

.notification((notification) => {
    console.log(notification.type);
});
```

在这个例子中，所有通过 `broadcast` 频道发送给 `App\User` 实例的通知都会被这个回调接收。

Laravel 框架内置的 `BroadcastServiceProvider` 中包含了一个针对 `App.User.*` 频道的频道授权回调。

9.2 缓存

1、配置

Laravel 为不同的缓存系统提供了统一的 API。缓存配置位于 `config/cache.php`。在该文件中你可以指定在应用中默认使用哪个缓存驱动。 Laravel 目前支持主流的缓存后端如 [Memcached](#) 和 [Redis](#) 等。

缓存配置文件还包含其他[文档](#)化的选项，确保仔细阅读这些选项。默认情况下，Laravel 被配置成使用文件缓存，这会将序列化数据和缓存对象存储到文件系统。对大型应用，建议使用内存缓存如 [Memcached](#) 或 APC，你甚至可以为同一驱动配置多个缓存配置。

驱动预备知识

数据库

使用 `database` 缓存驱动时，你需要设置一张表包含缓存缓存项。下面是该表的 `Schema` 声明：

```
Schema::create('cache', function($table) {  
    $table->string('key')->unique();  
    $table->text('value');  
    $table->integer('expiration');  
});
```

注：你还可以使用 Artisan 命令 `php artisan cache:table` 通过相应的 schema 生成迁移。

Memcached

使用 Memcached 缓存要求安装了 [Memcached PECL 包](#)，即 PHP Memcached 扩展。你可以在配

置文件 `config/cache.php` 中列出所有 Memcached 服务器：

```
'memcached' => [
    [
        'host' => '127.0.0.1',
        'port' => 11211,
        'weight' => 100
    ],
],
```

你还可以设置 `host` 选项为 UNIX socket 路径，如果你这样做，`port` 选项应该置为 `0`：

```
'memcached' => [
    [
        'host' => '/var/run/memcached/memcached.sock',
        'port' => 0,
        'weight' => 100
    ],
],
```

Redis

使用 Laravel 的 Redis 缓存之前，你需要通过 Composer 安装 `predis/predis` 包 (~1.0)。

想要了解更多关于 Redis 的配置，查看 Laravel 的 Redis 文档。

2、缓存使用

获取缓存实例

`Illuminate\Contracts\Cache\Factory` 和 `Illuminate\Contracts\Cache\Repository` 契约提供

了访问 Laravel 的缓存服务的方法。`Factory` 契约提供了所有访问应用定义的缓存驱动的方法。

`Repository` 契约通常是应用中 `cache` 配置文件中指定的默认缓存驱动的一个实现。

不过，你还可以使用 `Cache` 门面，这也是我们在整个文档中使用的方式，`Cache` 门面提供了简单方便的方式对底层 Laravel 缓存契约实现进行访问。

例如，让我们在控制器中导入 `Cache` 门面：

```
<?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');

        //
    }
}
```

访问多个缓存存储

使用 `Cache` 门面，你可以使用 `store` 方法访问不同的缓存存储器，传入 `store` 方法的键就是 `cache`

配置文件中 `stores` 配置数组里列出的相应的存储器：

```
$value = Cache::store('file')->get('foo');
Cache::store('redis')->put('bar', 'baz', 10);
```

从缓存中获取数据

`Cache` 门面的 `get` 方法用于从缓存中获取缓存项，如果缓存项不存在，返回 `null`。如果需要的话

你可以传递第二个参数到 `get` 方法指定缓存项不存在时返回的自定义默认值：

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

你甚至可以传递一个闭包作为默认值，如果缓存项不存在的话闭包的结果将会被返回。传递闭包

允许你可以从数据库或其它外部服务获取默认值：

```
$value = Cache::get('key', function() {
    return DB::table(...)->get();
});
```

检查缓存项是否存在

`has` 方法用于判断缓存项是否存在：

```
if (Cache::has('key')) {
    //
}
```

数值增加/减少

`increment` 和 `decrement` 方法可用于调整缓存中的整型数值。这两个方法都可以接收第二个参数

来指明缓存项数值增加和减少的数目：

```
Cache::increment('key');
Cache::increment('key', $amount);

Cache::decrement('key');
```

```
Cache::decrement('key', $amount);
```

获取&存储

有时候你可能想要获取缓存项，但如果请求的缓存项不存在时给它存储一个默认值。例如，你可能想要从缓存中获取所有用户，或者如果它们不存在的话，从数据库获取它们并将其添加到缓存中，你可以通过使用 `Cache::remember` 方法实现：

```
$value = Cache::remember('users', $minutes, function() {
    return DB::table('users')->get();
});
```

如果缓存项不存在，传递给 `remember` 方法的闭包被执行并且将结果存放到缓存中。

你还可以联合 `remember` 和 `forever` 方法：

```
$value = Cache::rememberForever('users', function() {
    return DB::table('users')->get();
});
```

获取&删除

如果你需要从缓存中获取缓存项然后删除，你可以使用 `pull` 方法，和 `get` 方法一样，如果缓存项不存在的话返回 `null`：

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

在缓存中存储数据

你可以使用 `Cache` 门面上的 `put` 方法在缓存中存储数据。当你在缓存中存储数据的时候，需要指定数据被缓存的时间（分钟数）：

```
Cache::put('key', 'value', $minutes);
```

除了传递缓存项失效时间，你还可以传递一个代表缓存项有效时间的 PHP `Datetime` 实例：

```
$expiresAt = Carbon::now()->addMinutes(10);
Cache::put('key', 'value', $expiresAt);
```

缓存不存在的情况下存储数据

`add` 方法只会在缓存项不存在的情况下添加数据到缓存，如果数据被成功添加到缓存返回 `true`，

否则，返回 `false`：

```
Cache::add('key', 'value', $minutes);
```

永久存储数据

`forever` 方法用于持久化存储数据到缓存，这些值必须通过 `forget` 方法手动从缓存中移除：

```
Cache::forever('key', 'value');
```

注：如果你使用的是 Memcached 驱动，当缓存数据达到上限后永久存储的数据就会被移除。

从缓存中移除数据

你可以使用 `Cache` 门面上的 `forget` 方法从缓存中移除缓存项数据：

```
Cache::forget('key');
```

还可以使用 `flush` 方法清除所有缓存：

```
Cache::flush();
```

注：清除缓存并不管什么缓存键前缀，而是从缓存系统中移除所有数据，所以在使用这个方法时

如果其他应用与本应用有共享缓存时需要格外注意。

3、缓存标签

注：缓存标签目前不支持 `file` 或 `database` 缓存驱动，此外，当使用多标签的缓存被设置为永久存储时，使用 `memcached` 驱动的缓存有着最佳性能表现，因为 Memcached 会自动清除陈旧记录。

存储被打上标签的缓存项

缓存标签允许你给相关缓存项打上同一个标签以便于后续清除这些缓存值，被打上标签的缓存可以通过传递一个被排序的标签数组来访问。例如，我们可以通过以下方式在添加缓存的时候设置标签：

```
Cache::tags(['people', 'artists'])->put('John', $john, $minutes);
Cache::tags(['people', 'authors'])->put('Anne', $anne, $minutes);
```

你可以给多个缓存项打上相同标签，这是没有数目限制的。

访问被打上标签的缓存项

要获取被打上标签的缓存项，传递同样的有序标签数组到 `tags` 方法然后使用你想要获取的 key 来调用 `get` 方法：

```
$john = Cache::tags(['people', 'artists'])->get('John');
$anne = Cache::tags(['people', 'authors'])->get('Anne');
```

移除被打上标签的数据项

你可以同时清除被打上同一标签/标签列表的所有缓存项，例如，以下语句会移除被打上 `people` 或 `authors` 标签的所有缓存：

```
Cache::tags(['people', 'authors'])->flush();
```

这样，上面设置的 `Anne` 和 `John` 缓存项都会从缓存中移除。

相反，以下语句只移除被打上 `authors` 标签的语句，所以只有 `Anne` 会被移除而 `John` 不会：

```
Cache::tags('authors')->flush();
```

4、添加自定义缓存驱动

编写驱动

要创建自定义的缓存驱动，首先需要实现 `Illuminate\Contracts\Cache\Store` 契约，所以，我们的 `MongoDB` 缓存实现看起来会像这样子：

```
<?php

namespace App\Extensions;

use Illuminate\Contracts\Cache\Store;

class MongoStore implements Store
{
    public function get($key) {}

    public function many(array $keys);
```

```
public function put($key, $value, $minutes) {}

public function putMany(array $values, $minutes);

public function increment($key, $value = 1) {}

public function decrement($key, $value = 1) {}

public function forever($key, $value) {}

public function forget($key) {}

public function flush() {}

public function getPrefix() {}

}
```

我们只需要使用一个 MongoDB 连接来实现其中的每一个方法，想要看如何实现每个方法的示例，可以参考 Laravel 底层源码 `Illuminate\Cache\MemcachedStore`，实现完成后，我们就可以完成自定义驱动注册：

```
Cache::extend('mongo', function($app) {
    return Cache::repository(new MongoStore());
});
```

注：如果你在担心将自定义缓存驱动代码放到哪，考虑将其放到 Packgist！或者，你可以在 `app` 目录下创建一个 `Extensions` 命名空间。然而，记住 Laravel 并没有一个严格的应用目录结构，你可以基于你的需要自由的组织目录结构。

注册驱动

要通过 Laravel 注册自定义的缓存驱动，可以使用 `Cache` 门面上的 `extend` 方法。对 `Cache::extend` 的调用可以在 Laravel 自带的 `App\Providers\AppServiceProvider` 提供的 `boot` 方法中完成，或

者，你也可以创建自己的服务提供者来收容扩展——只是别忘了在配置文件 `config/app.php` 中

注册服务提供者到 `providers` 数组：

```
<?php

namespace App\Providers;

use App\Extensions\MongoStore;
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\ServiceProvider;

class CacheServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Cache::extend('mongo', function($app) {
            return Cache::repository(new MongoStore());
        });
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

传递给 `extend` 方法的第一个参数是驱动名称。该值对应配置文件 `config/cache.php` 中

的 `driver` 选项。第二个参数是返回 `Illuminate\Cache\Repository` 实例的闭包。该闭包中被传入

一个 `$app` 实例，也就是 服务容器 的一个实例。

扩展被注册后，只需简单更新配置文件 `config/cache.php` 的 `driver` 选项为自定义扩展名称即可。

5、缓存事件

要在每次缓存操作时执行代码，你可以监听缓存触发的事件，通常，你可以将这些缓存处理器代码放到 `EventServiceProvider` 中：

```
/*
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Cache\Events\CacheHit' => [
        'App\Listeners\LogCacheHit',
    ],
    'Illuminate\Cache\Events\CacheMissed' => [
        'App\Listeners\LogCacheMissed',
    ],
    'Illuminate\Cache\Events\KeyForgotten' => [
        'App\Listeners\LogKeyForgotten',
    ],
    'Illuminate\Cache\Events\KeyWritten' => [
        'App\Listeners\LogKeyWritten',
    ],
];
```

9.3 事件

1、简介

Laravel 事件提供了简单的观察者模式实现，允许你订阅和监听应用中的事件。事件类通常存放
在 `app/Events` 目录，监听器存放在 `app/Listeners`。如果你在应用中没有看到这些目录，不要担
心，因为它们会在你使用 Artisan 命令生成事件和监听器的时候创建。

事件为应用功能模块解耦提供了行之有效的办法，因为单个事件可以有多个监听器而这些监听器
之间并不相互依赖。例如，你可能想要在每次订单发送时给用户发送一个 Slack 通知，有了事件
之后，你大可不必将订单处理代码和 Slack 通知代码耦合在一起，而只需要简单触发一个可以被
监听器接收并处理为 Slack 通知的 `OrderShipped` 事件即可。

2、注册事件/监听器

Laravel 自带的 `EventServiceProvider` 为事件监听器注册提供了方便之所。其中的 `listen` 属性
包含了事件（键）和对应监听器（值）数组。如果应用需要，你可以添加多个事件到该数组。例
如，让我们添加一个 `OrderShipped` 事件：

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'App\Events\OrderShipped' => [
        'App\Listeners\SendShipmentNotification',
    ],
];
```

生成事件/监听器类

当然，手动为每个事件和监听器创建文件是很笨重的，取而代之地，我们可见简单添加监听器和事件到 `EventServiceProvider` 然后使用 `event:generate` 命令。该命令将会生成罗列在 `EventServiceProvider` 中的所有事件和监听器。当然，已存在的事件和监听器不会被重复创建：

```
php artisan event:generate
```

手动注册事件

通常，我们需要通过 `EventServiceProvider` 的 `$listen` 数组注册事件，此外，你还可以在 `EventServiceProvider` 的 `boot` 方法中手动注册基于闭包的事件：

```
/**  
 * Register any other events for your application.  
 *  
 * @return void  
 */  
  
public function boot()  
{  
    parent::boot();  
  
    Event::listen('event.name', function ($foo, $bar) {  
        //  
    });  
}
```

通配符事件监听器

你甚至还可以使用通配符 * 来注册监听器，这样就可以通过同一个监听器捕获多个事件。通配符监听器接收整个事件数据数组作为参数：

```
$events->listen('event.*', function (array $data) {
    //
});
```

3、定义事件

事件类是一个处理与事件相关的简单数据容器，例如，假设我们生成的 OrderShipped 事件接收一个 Eloquent ORM 对象：

```
<?php

namespace App\Events;

use App\Order;
use App\Events\Event;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Event
{
    use SerializesModels;

    public $order;

    /**
     *

```

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

正如你所看到的，该事件类不包含任何特定逻辑，只是一个存放被购买的 `Order` 对象的容器，如果事件对象被序列化的话，事件使用的 `SerializesModels` trait 将会使用 PHP 的 `serialize` 函数序列化所有 Eloquent 模型。

4、定义监听器

接下来，让我们看看我们的示例事件的监听器，事件监听器在 `handle` 方法中接收事件实例，`event:generate` 命令将会自动在 `handle` 方法中导入合适的事件类和类型提示事件。在 `handle` 方法内，你可以执行任何需要的逻辑以响应事件：

```
<?php  
  
namespace App\Listeners;  
  
use App\Events\OrderShipped;
```

```
class SendShipmentNotification

{
    /**
     * Create the event listener.
     *
     * @return void
     */

    public function __construct()
    {
        //
    }

    /**
     * Handle the event.
     *
     * @param OrderShipped $event
     * @return void
     */
    public function handle(OrderShipped $event)
    {
        // Access the order using $event->order...
    }
}
```

注：你的事件监听器还可以在构造器中类型提示任何需要的依赖，所有事件监听器通过[服务容器](#)

解析，所以依赖会自动注入。

停止事件继续往下传播

有时候，你希望停止事件被传播到其它监听器，你可以通过从监听器的 `handle` 方法中返回 `false` 来实现。

5、事件监听器队列

如果监听器将要执行耗时任务比如发送邮件或者发送 HTTP 请求，那么将监听器放到队列是一个不错的选择。在队列化监听器之前，确保已经配置好队列并且在服务器或本地环境启动一个队列监听器。

要指定某个监听器需要放到队列，只需要让监听器类实现 `ShouldQueue` 接口即可，通过 Artisan 命令 `event:generate` 生成的监听器类已经将接口导入当前命名空间，所有你可以直接拿来使用：

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    //

}
```

就是这么简单！当这个监听器被调用的时候，将会使用 Laravel 的队列系统通过事件分发器自动推送到队列。如果通过队列执行监听器的时候没有抛出任何异常，队列任务会在执行完成后被自

动删除。

手动访问队列

如果你需要手动访问底层队列任务的 `delete` 和 `release` 方法，在生成的监听器中默认导入的

`Illuminate\Queue\InteractsWithQueue` trait 为此提供了访问这两个方法的权限：

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    public function handle(OrderShipped $event)
    {
        if (true) {
            $this->release(30);
        }
    }
}
```

6、触发事件

要触发一个事件，可以传递事件实例到辅助函数 `event`，这个辅助函数会分发事件到所有注册的监听器。由于辅助函数 `event` 全局有效，所以可以在应用的任何地方调用它：

```
<?php

namespace App\Http\Controllers;

use App\Order;
use App\Events\OrderShipped;
use App\Http\Controllers\Controller;

class OrderController extends Controller

{
    /**
     * Ship the given order.
     *
     * @param int $orderId
     * @return Response
     */
    public function ship($orderId)
    {
        $order = Order::findOrFail($orderId);

        // Order shipment logic...
    }
}
```

```
    event(new OrderShipped($order));  
}  
}
```

注：测试的时候，只需要断言特定事件被触发，无需真正触发监听器，Laravel 内置的测试函数让这一实现不在话下。

7、事件订阅者

编写事件订阅者

事件订阅者是指那些在类本身中订阅多个事件的类，通过事件订阅者你可以在单个类中定义多个事件处理器。订阅者需要定义一个 `subscribe` 方法，该方法中传入一个事件分发器实例。你可以在给定的分发器中调用 `listen` 方法注册事件监听器：

```
<?php  
  
namespace App\Listeners;  
  
class UserEventSubscriber  
{  
    /**  
     * Handle user login events.  
     */  
    public function onUserLogin($event) {}  
  
    /**
```

```
* Handle user logout events.  
*/  
  
public function onUserLogout($event) {}  
  
/**  
 * Register the listeners for the subscriber.  
 *  
 * @param Illuminate\Events\Dispatcher $events  
 */  
  
public function subscribe($events)  
{  
  
    $events->listen(  
        'Illuminate\Auth\Events\Login',  
        'App\Listeners\UserEventSubscriber@onUserLogin'  
    );  
  
    $events->listen(  
        'Illuminate\Auth\Events\Logout',  
        'App\Listeners\UserEventSubscriber@onUserLogout'  
    );  
}  
  
}
```

注册事件订阅者

编写好订阅者之后，就可以通过事件分发器对订阅者注册，你可以使用 `EventServiceProvider` 提供的 `$subscribe` 属性来注册订阅者。例如，让我们添加 `UserEventSubscriber`：

```
<?php

namespace App\Providers;

use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvider;

class EventServiceProvider extends ServiceProvider
{
    /**
     * The event listener mappings for the application.
     *
     * @var array
     */
    protected $listen = [
        //
    ];

    /**
     * The subscriber classes to register.
     *
     * @var array
     */
    protected $subscribe = [
        'App\Listeners\UserEventSubscriber',
    ];
}
```

9.4 文件存储

1、简介

Laravel 基于 Frank de Jonge 开发的 PHP 包 [Flysystem](#) 提供了强大的[文件系统](#)抽象。Laravel 文件系统集成对使用驱动处理本地文件系统进行了简化，这些驱动包括 [Amazon S3](#)，以及 [Rackspace 云存储](#)。此外在这些[存储](#)选项间切换非常简单，因为对不同系统而言，API 是一致的。

2、配置

文件系统配置文件位于 `config/filesystems.php`。在该文件中可以配置所有“磁盘”，每个磁盘描述了特定的存储驱动和存储位置。该配置文件为每种支持的驱动提供了示例配置，所以，简单编辑该配置来应用你的存储参数和认证信息。

当然，你想配置磁盘多少就配置多少，多个磁盘也可以共用同一个驱动。

公共磁盘

`public` 磁盘用于存储可以被公开访问的文件，默认情况下，`public` 磁盘使用 `local` 驱动并将文件存储在 `storage/app/public`，要让这些文件可以通过 web 访问到，需要创建一个软链 `public/storage` 指向 `storage/app/public`，这种方式可以将公开访问的文件保存在一个很容易被不同部署环境共享的目录，在使用零停机时间部署系统如 [Envoyer](#) 的时候尤其方便。

要创建这个软链，可以使用 Artisan 命令 `storage:link`：

```
php artisan storage:link
```

文件被存储并且软链已经被创建的情况下，就可以使用辅助函数 `asset` 创建一个指向该文件的 URL：

```
echo asset('storage/file.txt');
```

本地驱动

使用 `local` 驱动的时候，所有的文件操作都相对于定义在配置文件中的 `root` 目录，默认情况下，

该值设置为 `storage/app` 目录，因此，下面的方法将会存储文件到 `storage/app/file.txt`：

```
Storage::disk('local')->put('file.txt', 'Contents');
```

驱动预备知识

Composer 包

在使用 Amazon S3 或 Rackspace 驱动之前，需要通过 Composer 安装相应的包：

- Amazon S3: `league/flysystem-aws-s3-v3 ~1.0`
- Rackspace: `league/flysystem-rackspace ~1.0`

S3 驱动配置

S3 驱动配置信息位于配置文件 `config/filesystems.php`，该文件包含 S3 驱动的示例配置数组。

你可以使用自己的 S3 配置和认证信息自由编辑该数组。

FTP 驱动配置

Laravel 的文件系统集成了 FTP 操作，不过，框架默认的配置文件 `filesystems.php` 并没有提供示例配置。如果你需要配置一个 FTP 文件系统，可以使用以下示例配置：

```
'ftp' => [
    'driver'    => 'ftp',
    'host'      => 'ftp.example.com',
    'username'  => 'your-username',
    'password'  => 'your-password',

    // Optional FTP Settings...
    // 'port'      => 21,
    // 'root'      => '',
];
```

```
// 'passive' => true,  
// 'ssl'      => true,  
// 'timeout' => 30,  
],
```

Rackspace 驱动配置

Laravel 的 [Flysystem](#) 还集成了 Rackspace，同样，默认配置文件 [filesystems.php](#) 也没有提供对应的示例配置，如果你需要配置 Rackspace 文件系统，可以使用以下示例配置：

```
'rackspace' => [  
    'driver'      => 'rackspace',  
    'username'   => 'your-username',  
    'key'        => 'your-key',  
    'container'  => 'your-container',  
    'endpoint'   => 'https://identity.api.rackspacecloud.com/v2.0/',  
    'region'     => 'IAD',  
    'url_type'   => 'publicURL',  
,
```

3、获取硬盘实例

[Storage 门面](#)用于和你配置的任意磁盘进行交互，例如，你可以使用该门面上的 [put](#) 方法来存储头像到默认磁盘，如果你调用 [Storage](#) 门面上的方法而没有调用 [disk](#) 方法，则调用的方法会自动被传递到默认磁盘：

```
use Illuminate\Support\Facades\Storage;
```

```
Storage::put('avatars/1', $fileContents);
```

与多个磁盘进行交互时，可以使用 `Storage` 门面上的 `disk` 方法访问特定磁盘：

```
Storage::disk('s3')->put('avatars/1', $fileContents);
```

4、获取文件

`get` 方法用于获取给定文件的内容，该方法将会返回该文件的原生字符串内容。需要注意的是，

所有文件路径都是相对于配置文件中指定的磁盘默认根目录：

```
$contents = Storage::get('file.jpg');
```

`exists` 方法用于判断给定文件是否存在与磁盘上：

```
$exists = Storage::disk('s3')->exists('file.jpg');
```

文件 URL

使用 `local` 或 `s3` 驱动时，可以使用 `url` 方法获取给定文件的 URI。如果你使用的是 `local` 驱动，

通常会在给定路径前加上 `/storage`，并返回该文件的相对 URL；如果使用的是 `s3` 驱动，则会返回完整的远程 URL：

```
use Illuminate\Support\Facades\Storage;  
$url = Storage::url('file1.jpg');
```

注：记住，如果你在使用 `local` 驱动，所有需要公开访问的文件都应该存放在 `storage/app/public`

目录下，此外，你还需要为 `storage/app/public` 创建一个软链接。

文件元信息

除了读写文件之外， Laravel 还可以提供文件本身的信息。例如，`size` 方法可用于以字节方式返回文件大小：

```
use Illuminate\Support\Facades\Storage;  
$size = Storage::size('file1.jpg');
```

`lastModified` 方法以 UNIX 时间戳格式返回文件最后一次修改时间：

```
$time = Storage::lastModified('file1.jpg');
```

5、存储文件

`put` 方法可用于存储原生文件内容到磁盘。此外，还可以传递一个 PHP 资源到 `put` 方法，该方法将会使用 Flysystem 底层的流支持。在处理大文件的时候推荐使用文件流：

```
use Illuminate\Support\Facades\Storage;  
  
Storage::put('file.jpg', $contents);  
Storage::put('file.jpg', $resource);
```

自动文件流

如果你想要 Laravel 自动将给定文件流输出到对应存储路径，可以使用 `putFile` 或 `putFileAs` 方法，该方法接收 `Illuminate\Http\File` 或 `Illuminate\Http\UploadedFile` 实例，然后自动将文

件流保存到期望的路径：

```
use Illuminate\Http\File;

// Automatically calculate MD5 hash for file name...
Storage::putFile('photos', new File('/path/to/photo'));

// Manually specify a file name...
Storage::putFileAs('photos', new File('/path/to/photo'), 'photo.jpg');
```

这里有一些关于 `putFile` 方法的重要注意点，注意到我们只指定了目录名，默认情况下，`putFile`

方法会基于文件内容自动生成文件名。实现原理是对文件内容进行 MD5 哈希运算。`putFile` 方法

会返回文件路径，包括文件名，以便于在数据库中进行存储。

`putFile` 和 `putFileAs` 方法还接收一个用于指定存储文件“能见度”的参数，这在你将文件存储到云存储（如 S3）平台并期望文件可以被公开访问时很有用：

```
Storage::putFile('photos', new File('/path/to/photo'), 'public');
```

添加内容到文件开头/结尾

`prepend` 和 `append` 方法允许你轻松插入内容到文件开头/结尾：

```
Storage::prepend('file.log', 'Prepended Text');
Storage::append('file.log', 'Appended Text');
```

拷贝 & 移动文件

`copy` 方法将磁盘中已存在的文件从一个地方拷贝到另一个地方，而 `move` 方法将磁盘中已存在的文件从一定地方移到到另一个地方：

```
Storage::copy('old/file1.jpg', 'new/file1.jpg');
Storage::move('old/file1.jpg', 'new/file1.jpg');
```

文件上传

在 web 应用中，最常见的存储文件案例就是存储用户上传的文件，如用户头像、照片和文档等。

Laravel 通过使用上传文件实例上的 `store` 方法让存储上传文件变得简单。你只需要传入上传文件保存的路径并调用 `store` 方法即可：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserAvatarController extends Controller
{
    /**
     * Update the avatar for the user.
     *
     * @param Request $request
     * @return Response
}
```

```
/*
public function update(Request $request)
{
    $path = $request->file('avatar')->store('avatars');

    return $path;
}
```

这里有一些需要注意的重要事项，在这里我们只指定了目录名，而不是文件名。默认情况下，`store` 方法会基于文件内容自动生成文件名，这通过对文件内容进行 MD5 实现。`store` 方法会返回文件路径以便在数据库中保存文件路径和文件名。

你还可以调用 `Storage` 门面上的 `putFile` 方法来执行与上例同样的文件操作：

```
$path = Storage::putFile('avatars', $request->file('avatar'));
```

注：如果你接收的上传文件尺寸很大，你可能需要手动指定文件名，因为计算大文件的 MD5 哈希值很耗内存。

指定文件名

如果你不想要自动生成文件名，可以使用 `storeAs` 方法，该方法接收路径、文件名以及磁盘（可选）作为参数：

```
$path = $request->file('avatar')->storeAs(
    'avatars', $request->user()->id
);
```

当然，你还可以使用 `Storage` 门面上的 `putFileAs` 方法，该方法与上面的方法实现同样的操作：

```
$path = Storage::putFileAs(
    'avatars', $request->file('avatar'), $request->user()->id
);
```

指定磁盘

默认情况下，`store` 方法会使用默认的磁盘，如果你想要指定其它磁盘，传递磁盘名称作为 `store` 方法的第二个参数即可：

```
$path = $request->file('avatar')->store(
    'avatars' . $request->user()->id, 's3'
);
```

文件能见度

在 Laravel 的 Flysystem 集成中，“能见度”是对不同平台上文件权限的抽象，文件可以被声明成 `public` 或 `private`，当文件被声明为 `public`，意味着文件可以被其他人访问。例如，使用 S3 时，可以获取 `public` 文件的 URL。

使用 `put` 方法设置文件的时候可以顺便设置能见度：

```
use Illuminate\Support\Facades\Storage;
Storage::put('file.jpg', $contents, 'public');
```

如果文件已经被存储，能见度可以通过 `getVisibility` 和 `setVisibility` 方法获取和设置：

```
$visibility = Storage::getVisibility('file.jpg');  
Storage::setVisibility('file.jpg', 'public');
```

6、删除文件

`delete` 方法接收单个文件名或多个文件数组并将其从磁盘移除：

```
use Illuminate\Support\Facades\Storage;  
  
Storage::delete('file.jpg');  
Storage::delete(['file1.jpg', 'file2.jpg']);
```

7、目录

获取一个目录下的所有文件

`files` 方法返回给定目录下的所有文件数组，如果你想要获取给定目录下包含子目录的所有文件

列表，可以使用 `allFiles` 方法：

```
use Illuminate\Support\Facades\Storage;  
  
$files = Storage::files($directory);  
$files = Storage::allFiles($directory);
```

获取一个目录下的所有子目录

`directories` 方法返回给定目录下所有目录数组，此外，可以使用 `allDirectories` 方法获取嵌套

的所有子目录数组：

```
$directories = Storage::directories($directory);
// 递归...
$directories = Storage::allDirectories($directory);
```

创建目录

`makeDirectory` 方法将会创建给定目录，包含子目录（递归）：

```
Storage::makeDirectory($directory);
```

删除目录

最后，`deleteDirectory` 方法用于移除目录，包括该目录下的所有文件：

```
Storage::deleteDirectory($directory);
```

8、自定义文件系统

Laravel 的 Flysystem 集成支持自定义驱动，为了设置自定义的文件系统，你需要创建一个[服务提供者](#)如 `DropboxServiceProvider`。在该提供者的 `boot` 方法中，你可以使用 `Storage` 门面的 `extend` 方法定义自定义驱动：

```
<?php

namespace App\Providers;

use Storage;
use League\Flysystem\Filesystem;
use Dropbox\Client as DropboxClient;
use Illuminate\Support\ServiceProvider;
use League\Flysystem\Dropbox\DropboxAdapter;

class DropboxServiceProvider extends ServiceProvider{
    /**
     * Perform post-registration booting of services.
}
```

```

/*
 * @return void
 */
public function boot()
{
    Storage::extend('dropbox', function($app, $config) {
        $client = new DropboxClient(
            $config['accessToken'], $config['clientIdentifier']
        );

        return new Filesystem(new DropboxAdapter($client));
    });
}

/**
 * Register bindings in the container.
 *
 * @return void
 */
public function register()
{
    //
}
}

```

`extend` 方法的第一个参数是驱动名称，第二个参数是获取`$app` 和`$config` 变量的闭包。该解析器闭包必须返回一个 `League\Flysystem\Filesystem` 实例。`$config` 变量包含了定义在配置文件 `config/filesystems.php` 中为特定磁盘定义的选项。

创建好注册扩展的服务提供者后，就可以使用配置文件 `config/filesystem.php` 中的 `dropbox` 驱动了。

9.5 邮件

1、简介

Laravel 基于 [SwiftMailer](#) 库提供了一套干净、清爽的[邮件](#) API。 Laravel 为 [SMTP](#)、[Mailgun](#)、[SparkPost](#)、[Amazon SES](#)、PHP 的 `mail` 函数，以及 `sendmail` 提供了驱动，从而允许你快速通过本地或云服务发送邮件。

邮件驱动预备知识

基于驱动的 API 如 [Mailgun](#) 和 SparkPost 通常比 SMTP 服务器更简单、更快，所以如果可以的话，尽可能使用这些服务。所有的 API 驱动要求应用已经安装 Guzzle HTTP 库，你可以通过 Composer 包管理器来安装它：

```
composer require guzzlehttp/guzzle
```

Mailgun 驱动

要使用 Mailgun 驱动 (Mailgun 前 10000 封邮件免费，后续收费)，首先安装 Guzzle，然后在配置文件 `config/mail.php` 中设置 `driver` 选项为 `mailgun`。接下来，验证配置文件 `config/services.php` 包含如下选项：

```
'mailgun' => [
    'domain' => 'your-mailgun-domain',
    'secret' => 'your-mailgun-key',
],
```

SparkPost 驱动

要使用 SparkPost 驱动，首先安装 Guzzle，然后在配置文件 `config/mail.php` 中设置 `driver` 选

项值为 sparkpost。接下来，验证配置文件 `config/services.php` 包含如下选项：

```
'sparkpost' => [
    'secret' => 'your-sparkpost-key',
],
```

SES 驱动

要使用 Amazon SES 驱动（收费），先安装 Amazon AWS 的 PHP SDK，你可以通过添加如下行到 `composer.json` 文件的 `require` 部分然后运行 `composer update` 命令来安装该库：

```
"aws/aws-sdk-php": "~3.0"
```

接下来，设置配置文件 `config/mail.php` 中的 `driver` 选项为 `ses`。然后，验证配置文件 `config/services.php` 包含如下选项：

```
'ses' => [
    'key' => 'your-ses-key',
    'secret' => 'your-ses-secret',
    'region' => 'ses-region', // e.g. us-east-1
],
```

2、生成可邮寄类

在 Laravel 中，应用发送的每一封邮件都可以表示为“可邮寄”类，这些类都存放在 `app/Mail` 目录。

如果没看到这个目录，别担心，它将会在你使用 `make:mail` 命令创建第一个可邮寄类时生成：

```
php artisan make:mail OrderShipped
```

3、编写可邮寄类

所有的可邮寄类配置都在 `build` 方法中完成，在这个方法中，你可以调用多个方法，例如 `from`、`subject`、`view` 和 `attach` 来配置邮件的内容和发送。

配置发件人

使用 `from` 方法

我们来看一下邮件发件人的配置，或者，换句话说，邮件的来自于谁。有两者方式来配置发送者，第一种方式是在可邮寄类的 `build` 方法方法中调用 `from` 方法：

```
/**  
 * Build the message.  
 *  
 * @return $this  
 */  
  
public function build()  
{  
    return $this->from('example@example.com')  
        ->view('emails.orders.shipped');  
}
```

使用全局的 `from` 地址

不过，如果你的应用在所有邮件中都使用相同的发送地址，在每个生成的可邮寄类中都调用 `from`

方法就显得很累赘。取而代之地，你可以在配置文件 `config/mail.php` 中指定一个全局的发送地址，该地址可用于在所有可邮寄类中没有指定其它发送地址的场景下：

```
'from' => ['address' => 'example@example.com', 'name' => 'App Name'],
```

配置视图

你可以在可邮寄类的 `build` 方法中使用 `view` 方法来指定渲染邮件内容时使用哪个模板，由于每封邮件基本都使用 [Blade 模板](#) 来渲染内容，所以你可以在构建邮件 HTML 时充分使用 Blade 模板引擎：

```
/**  
 * Build the message.  
 *  
 * @return $this  
 */  
  
public function build()  
{  
    return $this->view('emails.orders.shipped');  
}
```

注：你可以创建一个 `resources/views/emails` 目录来存放所有邮件模板，当然，你也可以将邮件模板放到 `resources/views` 目录下任意其它位置。

纯文本邮件

如果你想要定义一个纯文本版本的邮件，可以使用 `text` 方法。和 `view` 方法一样，`text` 方法接收

一个用于渲染邮件内容的模板名，你既可以定义纯文本消息也可以定义 HTML 消息：

```
/**  
 * 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，我们需要传递一些数据到渲染邮件的 HTML 视图以便被使用。有两种方式将数据传递到视图，第一种是可邮寄类的公共 (public) 属性在视图中自动生效，举个例子，我们将数据传递给可邮寄类的构造器并将数据设置给该类的公共属性：

```
<?php  
  
namespace App\Mail;  
  
use App\Order;  
use Illuminate\Bus\Queueable;  
use Illuminate\Mail\Mailable;  
use Illuminate\Queue\SerializesModels;  
  
class OrderShipped extends Mailable
```

```
{  
use Queueable, SerializesModels;  
  
/**  
 * The order instance.  
 *  
 * @var Order  
 */  
public $order;  
  
/**  
 * Create a new message instance.  
 *  
 * @return void  
 */  
public function __construct(Order $order)  
{  
    $this->order = $order;  
}  
  
/**  
 * Build the message.  
 *  
 * @return $this  
 */  
public function build()  
{  
    return $this->view('emails.orders.shipped');  
}
```

```
    }  
}
```

数据被设置给公共属性后，将会在视图中自动生效，所以你可以像在 Blade 模板中访问其它数据一样访问它们：

```
<div>  
    Price: {{ $order->price }}  
</div>
```

通过 with 方法

如果你想要在数据发送到模板之前自定义邮件数据的格式，可以通过 `with` 方法手动传递数据到视图。一般情况下，你还是需要通过可邮寄类的构造器传递数据，不过，这次你需要设置数据为 `protected` 或 `private` 属性，这样，这些数据就不会在视图中自动生效。然后，当调用 `with` 方法时，传递数据数组到该方法以便数据在视图模板中生效：

```
<?php  
  
namespace App\Mail;  
  
use App\Order;  
use Illuminate\Bus\Queueable;  
use Illuminate\Mail\Mailable;  
use Illuminate\Queue\SerializesModels;  
  
class OrderShipped extends Mailable
```

```
{  
use Queueable, SerializesModels;  
  
/**  
 * The order instance.  
 *  
 * @var Order  
 */  
protected $order;  
  
/**  
 * Create a new message instance.  
 *  
 * @return void  
 */  
public function __construct(Order $order)  
{  
    $this->order = $order;  
}  
  
/**  
 * Build the message.  
 *  
 * @return $this  
 */  
public function build()  
{  
    return $this->view('emails.orders.shipped')  
}
```

```
->with([
    'orderName' => $this->order->name,
    'orderPrice' => $this->order->price,
])
}
```

数据通过 `with` 方法传递到视图后，将会在视图中自动生效，因此你也可以像在 Blade 模板访问其它数据一样访问传递过来的数据：

```
<div>
    Price: {{ $orderPrice }}
</div>
```

附件

要添加附件到邮件，可以在可邮寄类的 `build` 方法中使用 `attach` 方法。`attach` 方法接收完整的文件路径作为第一个参数：

```
/**
 * 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，还可以通过传递一个数组作为 `attach` 方法第二个参数来指定文件显示名或 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',  
        ]);  
}
```

原生数据附件

`attachData` 方法可用于添加原生的字节字符串作为附件。如果你想在内存中生成 PDF，并且在不保存到磁盘的情况下将其添加到邮件作为附件，则可以使用该方法。`attachData` 方法接收数据字节作为第一个参数，文件名作为第二个参数，可选数组作为第三个参数：

```
/**
```

```
* Build the message.  
*  
* @return $this  
*/  
  
public function build()  
{  
  
    return $this->view('emails.orders.shipped')  
        ->attachData($this->pdf, 'name.pdf', [  
            'mime' => 'application/pdf',  
        ]);  
}
```

内联附件

嵌套内联图片到邮件中通常是很笨重的，为此，Laravel 提供了便捷的方式附加图片到邮件并获取相应的 CID，要嵌入内联图片，在邮件视图中使用 `$message` 变量上的 `embed` 方法即可。Laravel 在所有邮件视图中注入 `$message` 变量并使其自动有效，所以你不用关心如何将这个变量手动传入视图的问题：

```
<body>  
  
    Here is an image:  
  
      
  
</body>
```

嵌入原生数据附件

如果你已经有一个想要嵌入邮件模板的原生数据字符串，可以使用 `$message` 变量上的 `embedData`

方法：

```
<body>
    Here is an image from raw data:

    
</body>
```

4、发送邮件

要发送一条信息，可以使用 [Mail 面面](#)上的 `to` 方法。`to` 方法接收邮箱地址、用户实例或用户集合作为参数。如果传递的是对象或对象集合，在设置邮件收件人的时候邮件会自动使用它们的 `email` 和 `name` 属性，所以事先要确保这些属性在相应类上有效。指定好收件人以后，传递一个可邮寄类的实例到 `send` 方法：

```
<?php

namespace App\Http\Controllers;

use App\Order;
use App\Mail\OrderShipped;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Mail;
use App\Http\Controllers\Controller;
```

```
class OrderController extends Controller
{
    /**
     * 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));
    }
}
```

发送邮件消息时并不仅限于指定收件人，你可以在单个方法链调用中设置“to”、“cc”以及“bcc”：

```
Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->send(new OrderShipped($order));
```

邮件队列

邮件消息队列

由于发送邮件消息可能会大幅度延长应用的响应时间，许多开发者选择将邮件发送放到队列中在后台发送，Laravel 中可以使用内置的统一队列 API 来实现这一功能。要将邮件消息放到队列，可以在指定消息的接收者后使用 `Mail` 门面上的 `queue` 方法：

```
Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->queue(new OrderShipped($order));
```

该方法自动将邮件任务推送到队列以便在后台发送。当然，你需要在使用该特性前配置队列。

延迟消息队列

如果你想要延迟已经放到队列中邮件的发送，可以使用 `later` 方法。作为第一个参数，`later` 方法接收一个 `DateTime` 实例来表示消息发送时间：

```
$when = Carbon\Carbon::now()->addMinutes(10);

Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->later($when, new OrderShipped($order));
```

推入到指定队列

由于通过 `make:mail` 命令生成的所有可邮寄类都使用了 `Illuminate\Bus\Queueable` trait，所以你可以调用可邮寄类实例上的 `onQueue` 和 `onConnection` 方法，以便为消息指定连接和队列名称：

```
$message = (new OrderShipped($order))
    ->onConnection('sqS')
    ->onQueue('emails');

Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->queue($message);
```

5、邮件&本地开发

本地环境开发发送邮件的应用时，你可能不想要真的发送邮件到有效的电子邮件地址，而只是想要做下测试。为此，Laravel 提供了几种方式“禁止”邮件的实际发送。

日志驱动

一种解决方案是在本地开发时使用 `log` 邮件驱动。该驱动将所有邮件信息写到日志文件中以备查看，想要了解更多关于每个环境的应用配置信息，查看[配置文档](#)。

通用配置

Laravel 提供的另一种解决方案是为框架发送的所有邮件设置通用收件人，这样的话，所有应用生成的邮件将会被发送到指定地址，而不是实际发送邮件指定的地址。这可以通过在配置文件 `config/mail.php` 中设置 `to` 选项来实现：

```
'to' => [
    'address' => 'example@example.com',
    'name' => 'Example'
],
```

Mailtrap

最后，你可以使用 [Mailtrap](#) 服务和 [smtp](#) 驱动发送邮件信息到“虚拟”邮箱，这种方法允许你在 Mailtrap 的消息查看器中查看最终的邮件。

6、事件

Laravel 会在发送邮件消息前触发一个事件，需要记住的是这个事件是在邮件被发送时触发，而不是推送到队列时，你可以在 [EventServiceProvider](#) 中注册对应的事件监听器：

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Mail\Events\MessageSending' => [
        'App\Listeners\LogSentMessage',
    ],
];
```

9.6 通知

1、简介

除了支持[发送邮件](#)之外，[Laravel](#) 还支持通过多种传输通道发送[通知](#)，这些通道包括[邮件](#)、短信

(通过 [Nexmo](#)) 以及等 [Slack](#) 等。通知可以存储在[数据库](#)以便后续在 web 界面中显示。

通常，通知都是很短的、用于告知用户应用中所发生[事件](#)的消息。例如，如果你在开发一个计费应用，则需要通过邮件或短信等渠道给用户发送“账单支付”通知。

2、创建通知

在 Laravel 中，每个通知都以单独类的形式存在（通常存放在 `app/Notifications` 目录），如果在应用中没看到这个目录，别担心，它将会在你运行 Artisan 命令 `make:notification` 的时候自动创建：

```
php artisan make:notification InvoicePaid
```

该命令会在 `app/Notifications` 目录下生成一个新的通知类，每个通知类都包含一个 `via` 方法以及多个消息构建方法（如 `toMail` 或 `toDatabase`），这些消息构建方法用于将通知转化成为特定渠道优化的消息。

3、发送通知

使用 Notifiable Trait

通知可以通过两种方式发送：使用 `Notifiable` trait 提供的 `notify` 方法或者使用 [Notification](#) 门面。首先，我们来检验 `Notifiable` trait。该 trait 被默认的 `App\User` 模型使用并提供一个可用于发送通知的方法：`notify`。`notify` 方法接收一个通知实例：

```
use App\Notifications\InvoicePaid;
```

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

注：记住，你可以在任何模型中使用 `Illuminate\Notifications\Notifiable` trait，不限于只在 `User` 模型中使用。

使用 Notification 门面

作为替代方案，还可以通过 `Notification` 门面发送通知。这主要在你需要发送通知给多个用户的时候很有用，要使用这个门面发送通知，需要将所有被通知用户和通知实例传递给 `send` 方法：

```
Notification::send($users, new InvoicePaid($invoice));
```

指定传输通道

每个通知类都有一个 `via` 方法用于决定通知通过何种通道传输，Laravel 开箱支持 `mail`、`database`、`broadcast`、`nexmo` 以及 `slack` 通道。

注：如果你想要使用其他传输通道，比如 Telegram 或 Pusher，参考社区提供的驱动：[Laravel 通知通道网站](#)。

`via` 方法接收一个 `$notifiable` 实例，用于指定通知被发送到的类实例。你可以使用 `$notifiable` 来判断通知通过何种通道传输：

```
/**  
 * Get the notification's delivery channels.  
 */
```

```
* @param mixed $notifiable
*
* @return array
*/
public function via($notifiable)
{
    return $notifiable->prefers_sms ? ['nexmo'] : ['mail', 'database'];
}
```

通知队列

注：使用通知队列前需要配置队列并开启一个队列任务。

发送同时可能是耗时的，尤其是通道需要调用额外的 API 来传输通知。为了加速应用的响应时间，可以让通知类实现 `ShouldQueue` 接口并使用 `Queueable` trait。如果通知类是通过 `make:notification` 命令生成的，那么该接口和 trait 已经默认导入，你可以快速将它们添加到通知类：

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Notifications\Notification;
use Illuminate\Contracts\Queue\ShouldQueue;

class InvoicePaid extends Notification implements ShouldQueue
{
```

```
use Queueable;  
  
// ...  
}
```

`ShouldQueue` 接口被添加到通知类以后，你可以像之前一样正常发送通知，Laravel 会自动检测到 `ShouldQueue` 接口然后将通知传输推送到队列：

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

如果你想要延迟通知的传输，可以在加上 `delay` 方法：

```
$when = Carbon::now()->addMinutes(10);  
  
$user->notify((new InvoicePaid($invoice))->delay($when));
```

4、邮件通知

格式化邮件消息

如果通知支持以邮件方式发送，你需要在通知类上定义一个 `toMail` 方法。该方法会接收一个 `$notifiable` 实体并返回 `Illuminate\Notifications\Messages\MailMessage` 实例。邮件消息可以包含多行文本以及对动作的调用，让我们来看一个 `toMail` 方法的示例：

```
/**  
 * Get the mail representation of the notification.  
 */
```

```
* @param mixed $notifiable
* @return \Illuminate\Notifications\Messages\MailMessage
*/
public function toMail($notifiable)
{
    $url = url('/invoice/'.$this->invoice->id);

    return (new MailMessage)
        ->greeting('Hello!')
        ->line('One of your invoices has been paid!')
        ->action('View Invoice', $url)
        ->line('Thank you for using our application!');
}
```

注：注意到我们在 `message` 方法中使用了 `$this->invoice->id`，你可以传递任何通知生成消息所需要的的数据到通知的构造器。

在这个例子中，我们注册了一条问候、一行文本、对动作的调用以及另一行文本。`MailMessage` 对象提供的这些方法让格式化短小的交易邮件变得简单快捷。`mail` 通道会将消息组件转化为漂亮的、响应式的、带有纯文本副本的 HTML 邮件模板。下面是一个通过 `mail` 通道生成的邮件示例：

Laravel

Hello!

One of your invoices has been paid!

[View Invoice](#)

Thank you for using our application!

Regards,
Laravel

If you're having trouble clicking the "View Invoice" button, copy and paste the URL below into your web browser:

<https://example.com/invoice/1>

© 2016 [Laravel](#). All rights reserved.

注：发送邮件通知时，确保在配置文件 `config/app.php` 中设置了 `name` 的值，该值将会用在邮件通知消息的头部和尾部。

自定义收件人

通过 `mail` 通道发送通知时，通知系统会自动在被通知实体上查找 `email` 属性，你可以通过在该

实体上定义一个 `routeNotificationForMail` 自定义使用哪个邮箱地址发送通知：

```
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Route notifications for the mail channel.
     *
     * @return string
     */
    public function routeNotificationForMail()
    {
        return $this->email_address;
    }
}
```

自定义主题

默认情况下，邮件的主题就是格式为“标题化”的通知类名，因此如果通知类被命名为 `InvoicePaid`，

邮件的主题就是 `Invoice Paid`，如果你想要为消息指定明确的主题，可以在构建消息的时候调用

`subject` 方法：

```
/**  
 * Get the mail representation of the notification.  
  
 * @param mixed $notifiable  
 * @return \Illuminate\Notifications\Messages\MailMessage  
 */  
  
public function toMail($notifiable)  
{  
  
    return (new MailMessage)  
        ->subject('Notification Subject')  
        ->line('...');  
  
}
```

自定义模板

你可以通过发布通知扩展包的资源来修改邮件通知所使用的 HTML 和纯文本模板。运行完下面这个命令之后，邮件通知模板将会存放到 `resources/views/vendor/notifications` 目录：

```
php artisan vendor:publish --tag=laravel-notifications
```

错误消息

一些通知会告知用户错误信息，例如一次失败的单据支付，你可以在构建消息的时候调用 `error`

方法标识邮件消息是一个错误消息。当在一个邮件消息上使用 `error` 方法时，动作按钮的颜色将会由蓝色变成红色：

```
/**  
 * Get the mail representation of the notification.  
 *  
 * @param mixed $notifiable  
 * @return \Illuminate\Notifications\Message  
 */  
  
public function toMail($notifiable)  
{  
  
    return (new MailMessage)  
        ->error()  
        ->subject('Notification Subject')  
        ->line('...');  
}
```

5、数据库通知

预备知识

`database` 通知通道会在数据表中存储通知信息，该表包含诸如通知类型以及用于描述通知的自定义 JSON 数据之类的信息。

你可以在用户界面中查询这个数据表来展示通知，不过，在此之前，需要创建数据表来保存信息，

你可以使用 `notifications:table` 命令来生成迁移然后生成数据表：

```
php artisan notifications:table
```

```
php artisan migrate
```

格式化数据库通知

如果一个通知支持存放在数据表，则需要在通知类中定义 `toDatabase` 或 `toArray` 方法，该方法接收一个 `$notifiable` 实体并返回原生的 PHP 数组。返回的数组会被编码为 JSON 格式然后存放到 `notifications` 表的 `data` 字段。让我们来看一个 `toArray` 方法的例子：

```
/**  
 * Get the array representation of the notification.  
 *  
 * @param mixed $notifiable  
 * @return array  
 */  
  
public function toArray($notifiable)  
{  
    return [  
        'invoice_id' => $this->invoice->id,  
        'amount' => $this->invoice->amount,  
    ];  
}
```

toDatabase Vs. toArray

`toArray` 方法还被 `broadcast` 通道用来判断广播什么数据到 JavaScript 客户端，如果你想要为

`database` 和 `broadcast` 通道提供两种不同的数组表示，则需要定义一个 `toDatabase` 方法来取代 `toArray` 方法。

访问通知

通知被存放到数据表之后，需要在被通知实体中有一个便捷的方式来访问它们。Laravel 默认提供的是 `App\User` 模型引入的 `Illuminate\Notifications\Notifiable` trait 包含了返回实体对应通知的 Eloquent 关联关系方法 `notifications`，要获取这些通知，可以像访问其它 Eloquent 关联关系一样访问该关联方法，默认情况下，通知按照 `created_at` 时间戳排序：

```
$user = App\User::find(1);

foreach ($user->notifications as $notification) {
    echo $notification->type;
}
```

如果你只想获取未读消息，可使用关联关系 `unreadNotifications`，同样，这些通知也按照 `created_at` 时间戳排序：

```
$user = App\User::find(1);

foreach ($user->unreadNotifications as $notification) {
    echo $notification->type;
}
```

注 :要想从 JavaScript 客户端访问通知 ,需要在应用中定义一个通知控制器为指定被通知实体(比如当前用户) 返回通知 ,然后从 JavaScript 客户端发送一个 HTTP 请求到控制器对应 URI。

标记通知为已读

一 般 情 况 下 , 我 们 会 将 用 户 浏 览 过 的 通 知 标 记 为 已 读 ,

`Illuminate\Notifications\Notifiable` trait 提供了一个 `markAsRead` 方法 , 用于更新对应通知数据库纪录上的 `read_at` 字段 :

```
$user = App\User::find(1);

foreach ($user->unreadNotifications as $notification) {
    $notification->markAsRead();
}
```

如果觉得循环便利每个通知太麻烦 , 可以直接在通知集合上调用 `markAsRead` 方法 :

```
$user->unreadNotifications->markAsRead();
```

还可以使用批量更新方式标记通知为已读 , 无需先从数据库获取通知 :

```
$user = App\User::find(1);

$user->unreadNotifications()->update(['read_at' => Carbon::now()]);
```

当然 , 你也可以通过 `delete` 方法从数据库中移除这些通知 :

```
$user->notifications()->delete();
```

6、广播通知

预备知识

在进行广播通知之前，需要配置并了解[事件广播](#)，事件广播为 JavaScript 客户端响应服务端事件触发铺平了道路。

格式化广播通知

`broadcast` 通道广播通知使用了 Laravel 的事件广播服务，从而允许 JavaScript 客户端实时捕获通知。如果通知支持广播，则需要在通知类上定义 `toBroadcast` 或 `toArray` 方法，该方法接收一个 `$notifiable` 实体并返回原生的 PHP 数组，返回的数组会编码成 JSON 格式然后广播到 JavaScript 客户端。让我们来看一个 `toArray` 方法的示例：

```
/**  
 * Get the array representation of the notification.  
 *  
 * @param  mixed  $notifiable  
 * @return array  
 */  
  
public function toArray($notifiable)  
{  
    return [  
}
```

```
'invoice_id' => $this->invoice->id,  
'amount' => $this->invoice->amount,  
];  
}
```

注：除了指定的数据，广播通知还包含了一个 `type` 字段，用于表示通知类名。

toBroadcast Vs. toArray

`toArray` 方法还可以用于 `database` 通道以判断在数据表中存储哪些数据。如果你想要为 `database` 和 `broadcast` 通道提供两种不同的数组表示方式，需要定义一个 `toBroadcast` 方法来取代 `toArray` 方法。

监听通知

通知将会以格式化为 `{notifiable}.{id}` 的形式在私人频道上广播，因此，如果你要发送通知到 ID 为 `1` 的 `App\User` 实例，那么该通知将会在私人频道 `App.User.1` 上进行广播。如果使用了 [Laravel Echo](#)，可以使用辅助函数 `notification` 轻松在某个频道上监听通知：

```
Echo.private('App.User.' + userId)  
.notification((notification) => {  
    console.log(notification.type);  
});
```

7、短信（SMS）通知

预备知识

Laravel 基于 [Nexmo](#) 发送短信通知，在使用 Nexmo 发送通知前，需要安装 Composer

包 `nexmo/client` 并在配置文件 `config/services.php` 中进行少许配置。你可以从拷贝以下示例配置开始：

```
'nexmo' => [
    'key' => env('NEXMO_KEY'),
    'secret' => env('NEXMO_SECRET'),
    'sms_from' => '15556666666',
],
```

`sms_from` 配置项就是你用于发送短信消息的手机号码，你需要在 Nexmo 控制面板中为应用生成一个手机号码。

格式化短信通知

如果通知支持以短信方式发送，需要在通知类上定义一个 `toNexmo` 方法。该方法接收一个 `$notifiable` 实体并返回 `Illuminate\Notifications\Messages\NexmoMessage` 实例：

```
/**
 * Get the Nexmo / SMS representation of the notification.
 *
 * @param mixed $notifiable
 * @return NexmoMessage
 */
public function toNexmo($notifiable)
{
    return (new NexmoMessage)
```

```
->content('Your SMS message content');

}
```

自定义来源号码

如果你要通过与配置文件 `config/services.php` 中指定的手机号不同的其他号码发送通知，可以使用 `NexmoMessage` 实例上的 `from` 方法：

```
/**  
 * Get the Nexmo / SMS representation of the notification.  
 *  
 * @param mixed $notifiable  
 * @return NexmoMessage  
 */  
  
public function toNexmo($notifiable)  
{  
    return (new NexmoMessage)  
        ->content('Your SMS message content')  
        ->from('15554443333');  
}
```

短信通知路由

使用 `nexmo` 通道发送通知的时候，通知系统会自动在被通知实体上查找 `phone_number` 属性。如果你想想要自定义通知被发送到的手机号码，可以在该实体上定义一个 `routeNotificationForNexmo` 方法：

```
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Route notifications for the Nexmo channel.
     *
     * @return string
     */
    public function routeNotificationForNexmo()
    {
        return $this->phone;
    }
}
```

8、Slack 通知

预备知识

在通过 Slack 发送通知前，必须通过 Composer 安装 Guzzle HTTP 库：

```
composer require guzzlehttp/guzzle
```

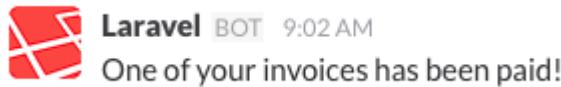
此外，你还要为 Slack 组配置一个“Incoming Webhook”集成。该集成会在你进行 Slack 通知路由的时候提供一个 URL。

格式化 Slack 通知

如果通知支持通过 Slack 消息发送，则需要在通知类上定义一个 `toSlack` 方法，该方法接收一个 `$notifiable` 实体并返回 `Illuminate\Notifications\Messages\SlackMessage` 实例，该实例包含文本内容以及格式化额外文本或数组字段的“附件”。让我们来看一个基本的 `toSlack` 使用示例：

```
/**  
 * Get the Slack representation of the notification.  
 *  
 * @param mixed $notifiable  
 * @return SlackMessage  
 */  
public function toSlack($notifiable)  
{  
    return (new SlackMessage)  
        ->content('One of your invoices has been paid!');  
}
```

在这个例子中，我们只发送一行简单的文本到 Slack，最终创建的消息如下：

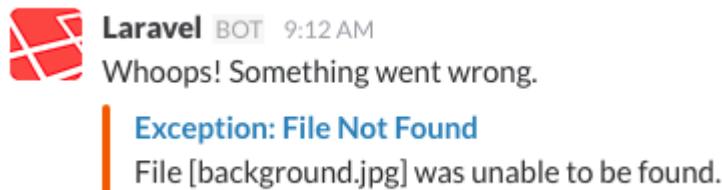


Slack 附件

你还可以添加“附件”到 Slack 消息。相对简单文本消息，附件可以提供更加丰富的格式选择。在这个例子中，我们会发送一个在应用程序中出现的异常错误通知，包含链接到更多异常细节的链接：

```
/**  
 * Get the Slack representation of the notification.  
 *  
 * @param mixed $notifiable  
 * @return SlackMessage  
 */  
  
public function toSlack($notifiable)  
{  
  
    $url = url('/exceptions/'.$this->exception->id);  
  
    return (new SlackMessage)  
        ->error()  
        ->content('Whoops! Something went wrong.')  
        ->attachment(function ($attachment) use ($url) {  
            $attachment->title('Exception: File Not Found', $url)  
            ->content('File [background.jpg] was not found.');?>;  
});  
}
```

上述代码会生成如下 Slack 消息：

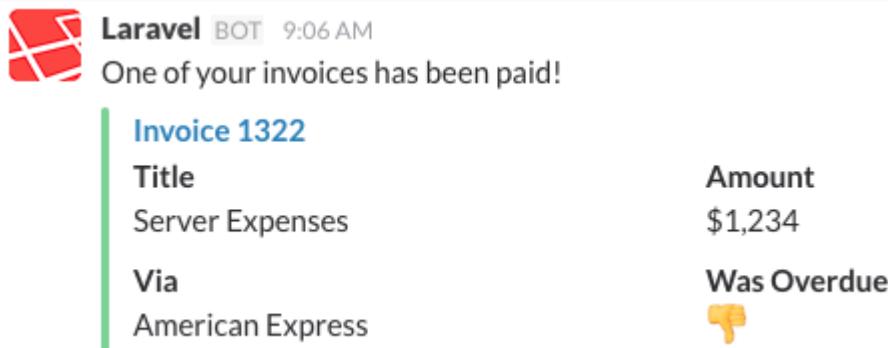


附件还允许你指定要呈献给用户的数组数据。为了提高可读性，给定的数组会以表格形式展示：

```
/**  
 * Get the Slack representation of the notification.  
  
 * @param mixed $notifiable  
 * @return SlackMessage  
 */  
  
public function toSlack($notifiable)  
{  
  
    $url = url('/invoices/'.$this->invoice->id);  
  
    return (new SlackMessage)  
        ->success()  
        ->content('One of your invoices has been paid!')  
        ->attachment(function ($attachment) use ($url) {  
            $attachment->title('Invoice 1322', $url)  
            ->fields([  
                'Title' => 'Server Expenses',  
                'Amount' => '$1,234',  
                'Via' => 'American Express',  
                'Was Overdue' => ':1:',  
            ]);  
        });  
}
```

```
});  
}
```

上述代码会生成如下 Slack 消息：



自定义发送者 & 接收者

你可以使用 `from` 和 `to` 方法自定义发送者和接收者，`from` 方法接收用户名和 emoji 标识符，

而 `to` 方法接收一个频道或用户名：

```
/**  
 * Get the Slack representation of the notification.  
 *  
 * @param mixed $notifiable  
 * @return SlackMessage  
 */  
  
public function toSlack($notifiable)  
{  
    return (new SlackMessage)  
        ->from('Ghost', ':ghost:')  
        ->to('#other');  
        ->content('This will be sent to #other')  
}
```

```
}
```

Slack 通知路由

要路由 Slack 通知到适当的位置，需要在被通知的实体上定义一个 `routeNotificationForSlack` 方法，这将会返回通知被发送到的 webhook URL。webhook URL 可通过在 Slack 组上添加一个“Incoming Webhook”服务来生成：

```
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Route notifications for the Slack channel.
     *
     * @return string
     */
    public function routeNotificationForSlack()
    {
        return $this->slack_webhook_url;
    }
}
```

```
    }  
}
```

9、通知事件

当通知被发送后，通知系统会触发 `Illuminate\Notifications\Events\NotificationSent` 事件，该事件实例包含被通知的实体（如用户）和通知实例本身。你可以在 `EventServiceProvider` 中为该事件注册监听器：

```
/**  
 * The event listener mappings for the application.  
 *  
 * @var array  
 */  
  
protected $listen = [  
  
    'Illuminate\Notifications\Events\NotificationSent' => [  
        'App\Listeners\LogNotification',  
    ],  
];
```

注：在 `EventServiceProvider` 中注册监听器之后，使用 Artisan 命令 `event:generate` 快速生成监听器类。

在事件监听器中，可以访问事件的 `notifiable`、`notification` 和 `channel` 属性了解通知接收者和通知本身的更多信息：

```
/**  
 * Handle the event.  
  
 * @param NotificationSent $event  
 * @return void  
 */  
  
public function handle(NotificationSent $event)  
{  
    // $event->channel  
    // $event->notifiable  
    // $event->notification  
}
```

10、自定义频道

通过上面的介绍，可见 Laravel 为我们提供了一大把通知通道，但是如果你想要编写自己的驱动以便通过其他通道发送通知，也很简单。首先定义一个包含 `send` 方法的类，该方法接收两个参数：`$notifiable` 和 `$notification`：

```
<?php  
  
namespace App\Channels;  
  
use Illuminate\Notifications\Notification;  
  
class VoiceChannel  
{
```

```
/*
 * Send the given notification.
 *
 * @param mixed $notifiable
 * @param \Illuminate\Notifications\Notification $notification
 * @return void
 */
public function send($notifiable, Notification $notification)
{
    $message = $notification->toVoice($notifiable);

    // Send notification to the $notifiable instance...
}
```

通知类被定义后，就可以在应用中通过 `via` 方法返回类名：

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use App\Channels\VoiceChannel;
use App\Channels\Messages\VoiceMessage;
use Illuminate\Notifications\Notification;
use Illuminate\Contracts\Queue\ShouldQueue;

class InvoicePaid extends Notification
{
    use Queueable;

    /**
     * Get the notification channels.
     *
     * @param mixed $notifiable

```

```
* @return array|string
*/
public function via($notifiable)
{
    return [VoiceChannel::class];
}

/**
 * Get the voice representation of the notification.
 *
 * @param mixed $notifiable
 * @return VoiceMessage
 */
public function toVoice($notifiable)
{
    // ...
}
```

9.7 队列

1、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。

连接和队列的关系

在开始使用 Laravel 队列以前，了解“连接”和“队列”的关系非常重要。在配置文件 `config/queue.php` 有关于“连接(connections)”的配置项。该配置项定义了后台队列服务类型，如 Amazon SQS, Beanstalk, 或 Redis. 显而易见，每种连接都可以有很多队列，可以想象在银行办理现金业务的各个窗口队列。

请注意每个连接都有一个“queue”配置项。当新的队列任务被添加到指定的连接时，该配置项的值就是默认监听的队列（名称）。换种说法，如果你没有指派特别的队列名称，那么“queue”的值，也是该任务默认添加到的队列（名称）。

```
// 以下的任务将被委派到默认队列...
dispatch(new Job);

// 以下任务将被委派到 "emails" 队列...
dispatch((new Job)->onQueue('emails'));
```

多数应用并不需要将任务分配成多到队列，单个队列已经非常适用。但是，应用的任务有优先级差异或者类别差异的时候，多队列将是更好地选择，尤其 Laravel 的队列进程已经支持的情况下。举个例子，你可以将高优先级的任务委派到“high”(高优先级)队列，从而让它优先执行。

```
php artisan queue:work --queue=high,default
```

驱动预备知识

数据库

如果使用数据库来驱动队列，你需要数据表保存任务信息。可以很容易的通过 Artisan 命令建立

这些表，相关知识需要参考 migration，代码示例如下：

```
php artisan queue:table //生成数据库队列的 migration  
php artisan migrate      //创建该数据库队列表
```

其他驱动预备知识

如果使用以下几种队列驱动，需要相应的依赖：

- Amazon SQS: `aws/aws-sdk-php ~3.0`
- [Beanstalkd](#): `pda/pheanstalk ~3.0`
- Redis: `predis/predis ~1.0`

2、创建任务

生成任务类

通常，所有的任务类都保存在 `app/Jobs` 目录。如果 `app/Jobs` 不存在，在运行 Artisan 命令 `make:job` 的时候，它将会自动创建。你可以通过 Artisan CLI 来生成队列任务类：

```
php artisan make:job SendReminderEmail
```

生成的类都实现了 `Illuminate\Contracts\Queue\ShouldQueue` 接口，告诉 Laravel 将该任务分配到队列，而不是立即运行。

任务类结构

任务类非常简单，通常只包含处理该任务的“handle”方法，让我们看一个任务类的例子。我们模

拟管理播客发布服务，并在发布以前上传相应的播客文件：

```
<?php

namespace App\Jobs;

use App\Podcast;
use App\AudioProcessor;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class ProcessPodcast implements ShouldQueue
{
    use InteractsWithQueue, Queueable, SerializesModels;

    protected $podcast;

    /**
     * 创建任务实例
     *
     * @param Podcast $podcast
     * @return void
     */
    public function __construct(Podcast $podcast)
    {
        $this->podcast = $podcast;
```

```

}

/**
 * 执行任务
 *
 * @param AudioProcessor $processor
 * @return void
 */
public function handle(AudioProcessor $processor)
{
    // 执行播客的上传...
}
}

```

在本示例中，我们将 Eloquent 模型作为参数直接传递到构造函数。因为该任务使用了 `SerializesModels` trait，Eloquent 模型将会在任务被执行时优雅地序列化和反序列化。如果你的队列任务在构造函数中接收 Eloquent 模型，只有模型的主键会被序列化到队列，当任务真正被执行的时候，队列系统会自动从数据库中获取整个模型实例。这对应用而言是完全透明的，从而避免序列化整个 Eloquent 模型实例引起的问题。

`handle` 方法在任务被处理的时候调用，注意我们可以在任务的 `handle` 方法中进行依赖注入。

Laravel 服务容器会自动注入这些依赖。

注：二进制数据，如原生图片内容，在传递给队列任务之前先经过 `base64_encode` 方法处理，此外，该任务被推送到队列时将不会被序列化为 JSON 格式。

3、委派任务

一旦你创建了任务类，就可以通过辅助函数 `dispatch` 委派它到队列。辅助函数 `dispatch` 需要的唯一参数就是该任务的实例：

```
<?php

namespace App\Http\Controllers;

use App\Jobs\ProcessPodcast;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PodcastController extends Controller

{
    /**
     * 保存新的播客.
     *
     * @param Request $request
     * @return Response
     */

    public function store(Request $request)
    {
        // 创建新的播客

        dispatch(new ProcessPodcast($podcast));
    }
}
```

```
}
```

辅助函数 `dispatch` 是全局且易用的函数，而且非常容易测试。请通过 Laravel 的测试[文档](#)可以了解更多细节。

延时任务

有时候你可能想要延迟队列任务的执行，可以通过在任务实例使用 `delay` 方法。该 `delay` 方法由 `Illuminate\Bus\Queueable` trait 提供，已经自动添加在通过命令行生成的任务类中。例如你希望将某个任务在创建 10 分钟以后才执行：

```
<?php

namespace App\Http\Controllers;

use Carbon\Carbon;
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...

    $job = (new ProcessPodcast($pocast))
        ->delay(Carbon::now()->addMinutes(10));

    dispatch($job);
}

}
```

Amazon SQS 的队列服务最长延时 15 分钟。

自定义队列和连接

委派到指定的队列

由于任务可推送到的不同队列，你应该将队列任务进行“分类”，甚至根据优先级来分配每个队列的进程数。请注意，这并不意味着使用了配置项中那些不同的连接来管理队列，实际上只有单一连接会被用到。要指定队列，请在任务实例使用 `onQueue` 方法：

```
<?php

namespace App\Http\Controllers;

use App\Jobs\ProcessPodcast;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;
```

```
class PodcastController extends Controller
{
    /**
     * 保存播客
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        // 创建播客

        $job = (new ProcessPodcast($podcast))->onQueue('processing');

        dispatch($job);
    }
}
```

委派到指定的连接

如果你的项目使用多个连接来管理队列，那么你会用到委派任务到指定的连接。请在任务实例使

用 `onConnection` 方法来指定连接:

```
<?php

namespace App\Http\Controllers;
```

```
use App\Jobs\ProcessPodcast;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PodcastController extends Controller
{
    /**
     * 保存播客
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        // 创建播客

        $job = (new ProcessPodcast($podcast))->onConnection('sq
        dispatch($job);
    }
}
```

当然，你可以同时使用 `onConnection` 和 `onQueue` 方法来切换不同的连接和队列：

```
$job = (new ProcessPodcast($podcast))
    ->onConnection('sq
    ->onQueue('processing');
```

处理错误

如果任务在处理的时候有异常抛出，则该任务将会被自动释放回队列以便再次尝试执行。任务会持续被释放直到尝试次数达到应用允许的最大次数。最大尝试次数通过 Artisan 任务 `queue:listen` 或 `queue:work` 上的 `--tries` 开关来定义。关于运行队列监听器的更多信息可以在下面看到。

4、运行队列进程

Laravel 包含了一个 Artisan 命令用来运行被推送到队列的新任务。你可以使用 `queue:work` 命令运行队列进程。请注意，队列进程开始运行后，会持续监听队列，直至你手动停止或 php 周期结束停止。

```
php artisan queue:work
```

注：为了保持队列进程 `queue:work` 持续在后台运行，你需要使用进程守护程序，比如 [Supervisor](#) 来确保队列进程持续运行。

请记住，队列进程是长生命周期的进程，会在启动后驻留内存。若应用有任何改动将不会影响到已经启动的进程。所以请在发布程序后，重启队列进程。

指定连接和队列

队列进程同样可以自定义连接和队列。队列进程使用到的名称必须在 `config/queue.php` 中已经配置：

```
php artisan queue:work redis
```

你可以自定义将某个队列进程指定某个连接来管理。举例来说，如果所有的邮件任务都是通过 `redis` 连接管理，那么可以用以下示例代码来启动单一进程仅仅出来单一队列：

```
php artisan queue:work redis --queue=emails
```

队列优先级

有时候你需要区分任务的优先级。比如，在配置文件 `config/queue.php` 中，你可能定义连接 `redis` 的默认队列为 `low`。可是，偶尔需要将任务委派到高优先级 `high`，可以这么做：

```
dispatch((new Job)->onQueue('high'));
```

如果期望所有 `high` 高优先级的队列都将先于 `low` 低优先级的任务执行，可以像这样启动队列进程

```
php artisan queue:work --queue=high,low
```

队列进程 & 部署

前文已经提到队列进程是长生命周期的进程，在重启以前，所有源码的修改并不会对其产生影响。所以，最简单的方法是在每次发布新版本后重新启动队列进程。你可以通过 `Aritisan` 命令 `queue:restart` 来优雅的重启队列进程：

```
php artisan queue:restart
```

该命令将在队列进程完成正在进行的任务后，结束该进程，避免队列任务的丢失或错误。你仍然需要通过 Supervisor 进程守护程序来确保队列进程的运行。

任务过期和超时

任务执行时间

在配置文件 `config/queue.php` 中，每个连接都定义了 `retry_after` 项。该项的目的是定义任务在执行以后多少秒后释放回队列。如果 `retry_after` 设定的值为 90，任务在运行 90 秒后还未完成，那么将被释放回队列而不是删除掉。毫无疑问，你需要把 `retry_after` 的值设定为任务执行时间的最大可能值。

注：只有亚马逊 SQS 配置信息不包含 `retry_after` 项。亚马逊 SQS 的任务执行时间基于 [Default Visibility Timeout](#)，该项在亚马逊 AWS 控制台配置。

队列进程超时

队列进程 `queue:work` 可以设定超时 `--timeout` 项。该 `--timeout` 控制队列进程执行每个任务的最长时间，如果超时，该进程将被关闭。各种错误都可能导致某个任务处于“冻结”状态，比如 HTTP 无响应等。队列进程超时就是为了将这些“冻结”的进程关闭。

```
php artisan queue:work --timeout=60
```

配置项 `retry_after` 和 Arisan 参数项 `--timeout` 不同，但目的都是为了确保任务的安全，并且只被成功的执行一次。

注：参数项 `--timeout` 的值应该是中小于配置项 `retry_after` 的值，这是为了确保队列进程总在

任务重试以前关闭。如果 `--timeout` 比 `retry_after` 大，则你的任务可能被执行两次。

5、守护进程 Supervisor 的配置

安装 Supervisor

Supervisor 是 Linux 系统中常用的进程守护程序。如果队列进程 `queue:work` 意外关闭，它会自动重启启动队列进程。在 Ubuntu 安装 Supervisor 非常简单：

```
sudo apt-get install supervisor
```

注：如果自己配置 Supervisor 有困难，可以考虑使用 [Laravel Forge](#)，它会为 Laravel 项目自动安装并配置 Supervisor。

配置 Supervisor

Supervisor 配置文件通常存放在 `/etc/supervisor/conf.d` 目录，在该目录中，可以创建多个配置文件指示 Supervisor 如何监视进程，例如，让我们创建一个开启并监视 `queue:work` 进程的 `laravel-worker.conf` 文件：

```
[program:laravel-worker]
process_name=%(program_name)s_%(process_num)02d
command=php /home/forge/app.com/artisan queue:work sqs --sleep=3 --tries=3
autostart=true
autorestart=true
user=forge
numprocs=8
```

```
redirect_stderr=true  
stdout_logfile=/home/forge/app.com/worker.log
```

在本例中，`numprocs` 指令让 Supervisor 运行 8 个 `queue:work` 进程并监视它们，如果失败的话自动重启。配置文件创建好了之后，可以使用如下命令更新 Supervisor 配置并开启进程：

启动 Supervisor

当你成功创建配置文件后，你需要刷新 Supervisor 的配置信息：

```
sudo supervisorctl reread  
sudo supervisorctl update  
sudo supervisorctl start laravel-worker:*
```

你可以通过 Supervisor 官方文档获的更多信息 [Supervisor 文档](#)。

6、处理失败的任务

不可避免会出现失败的任务。不必担心，Laravel 很容易设置任务允许的最大尝试次数，若是执行次数达到该限定，任务会被插入到 `failed_jobs` 表，失败任务的名字可以通过配置文件 `config/queue.php` 来配置。

要创建一个 `failed_jobs` 表的迁移，可以使用 `queue:failed-table` 命令

```
php artisan queue:failed-table  
php artisan migrate
```

通过 `--tries` 参数项来设置队列任务允许的最大尝试次数：

```
php artisan queue:work redis --tries=3
```

清理失败的任务

你可以在任务类中定义 `failed` 方法，从而允许你在失败发生时执行指定的动作，比如发送任务失败的通知，记录日志等。异常抛出的时候，就会传递到 `failed` 方法：

```
<?php

namespace App\Jobs;

use Exception;
use App\Podcast;
use App\AudioProcessor;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class ProcessPodcast implements ShouldQueue
{
    use InteractsWithQueue, Queueable, SerializesModels;

    protected $podcast;

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

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

/**
 * Execute the job.
 *
 * @param AudioProcessor $processor
 * @return void
*/
public function handle(AudioProcessor $processor)
{
    // Process uploaded podcast...
}

/**
 * The job failed to process.
 *
 * @param Exception $exception
 * @return void
*/
public function failed(Exception $e)
{
```

```
// 发送失败通知，etc...  
}  
}
```

任务失败事件

如果你期望在任务失败的时候触发某个事件，可以使用 `Queue::failing` 方法。该事件通过邮件或 HipChat 通知团队。举个例子，我么可以在 Laravel 自带的 `AppServiceProvider` 中附加一个回调到该事件：

```
<?php  
  
namespace App\Providers;  
  
use Illuminate\Support\Facades\Queue;  
use Illuminate\Queue\Events\JobFailed;  
use Illuminate\Support\ServiceProvider;  
  
class AppServiceProvider extends ServiceProvider  
{  
    /**  
     * Bootstrap any application services.  
     *  
     * @return void  
     */  
    public function boot()  
    {
```

```
Queue::failing(function (JobFailed $event) {
    // $event->connectionName
    // $event->job
    // $event->exception
});

/**
 * Register the service provider.
 *
 * @return void
 */
public function register()
{
    //
}

}
```

重试失败的任务

要查看已插入到 `failed_jobs` 数据表中的所有失败任务，可以使用 Artisan 命令 `queue:failed`：

```
php artisan queue:failed
```

该命令将会列出任务 ID，连接，对列和失败时间，任务 ID 可用于重试失败任务，例如，要重试一个 ID 为 5 的失败任务，要用到下面的命令

```
php artisan queue:retry 5
```

要重试所有失败任务，使用如下命令即可：

```
php artisan queue:retry all
```

如果你要删除一个失败任务，可以使用 `queue:forget` 命令：

```
php artisan queue:forget 5
```

要删除所有失败任务，可以使用 `queue:flush` 命令

```
php artisan queue:flush
```

7、任务事件

当你使用 `Queue` 门面的时候，可以使用 `before` 和 `after` 方法。你可以自定义在任务开始前或者结束后执行某个回调。这些回调可用来记录日志或者记录统计数据。通常，你可以在服务提供者中使用这些方法。比如，我们可能在 `AppServiceProvider` 这样用：

```
<?php  
  
namespace App\Providers;  
  
use Illuminate\Support\Facades\Queue;  
use Illuminate\Support\ServiceProvider;
```

```
use Illuminate\Queue\Events\JobProcessed;
use Illuminate\Queue\Events\JobProcessing;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Queue::before(function (JobProcessing $event) {
            // $event->connectionName
            // $event->job
            // $event->job->payload()
        });

        Queue::after(function (JobProcessed $event) {
            // $event->connectionName
            // $event->job
            // $event->job->payload()
        });
    }

    /**
     * Register the service provider.
     *
     */
}
```

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

声明：本文档由网友 AC1982 提供翻译支持，学院君进行校注。感谢 AC1982 的努力和付出！

10. 数据库

10.1 起步

1、简介

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

- MySQL
- Postgres
- SQLite
- SQL Server

配置

Laravel 让连接数据库和运行查询都变得非常简单。应用的数据库配置位于 [config/database.php](#)。

在该文件中你可以定义所有的数据库连接，并指定哪个连接是默认连接。该文件中提供了所有支持数据库系统的配置示例。

默认情况下，Laravel 示例[环境配置](#)已经为 [Laravel Homestead](#) 做好了设置，当然，你也可以按照需要为本地的数据库修改该配置。

SQLite 配置

使用 `touch database/database.sqlite` 命令创建好新的 SQLite 数据库之后，就可以使用数据库绝对路径配置环境变量指向这个新创建的数据库：

```
DB_CONNECTION=sqlite  
DB_DATABASE=/absolute/path/to/database.sqlite
```

SQL Server 配置

Laravel 开箱支持 SQL Server，不过，你需要为数据库添加连接配置到配置文件 `config/database.php`：

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

读写分离

有时候你希望使用一个数据库连接做查询，另一个数据库连接做插入、更新和删除，Laravel 使

得这种读写分离变得轻而易举，不管你用的是原生 SQL，还是查询构建器，还是 Eloquent ORM，合适的连接总是会被使用。

想要知道如何配置读/写连接，让我们看看下面这个例子：

```
'mysql' => [
    'read' => [
        'host' => '192.168.1.1',
    ],
    'write' => [
        'host' => '196.168.1.2'
    ],
    'driver' => 'mysql',
    'database' => 'database',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
    'collation' => 'utf8_unicode_ci',
    'prefix' => '',
],
```

注意我们在配置数组中新增了两个键：`read` 和 `write`，这两个键都对应一个包含单个键“host”的数组，读/写连接的其它数据库配置选项都共用 `mysql` 的主数组配置。

如果我们想要覆盖主数组中的配置，只需要将相应配置项放到 `read` 和 `write` 数组中即可。在本例中，`192.168.1.1` 将被用作“读”连接，而 `196.168.1.2` 将被用作“写”连接。两个数据库连接的凭证（用户名/密码）前缀、字符集以及其它配置将会共享 `mysql` 数组中的设置。

多个数据库连接

使用多个数据库连接的时候，可以通过 `DB` 门面上的 `connection` 方法访问每个连接。传递给 `connection` 方法的 `name` 对应配置文件 `config/database.php` 中列出的某个连接：

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

你还可以使用连接实例上的 `getPdo` 方法访问底层原生的 PDO 实例：

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

2、运行原生 SQL 查询

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

`select`, `update`, `insert`, `delete` 和 `statement`。

运行 Select 查询

运行一个最基本的查询，可以使用 `DB` 门面的 `select` 方法：

```
<?php

namespace App\Http\Controllers;

use 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::select('select * from users where active = ?', [1]);

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

传递给 `select` 方法的第一个参数是原生的 SQL 语句，第二个参数需要绑定到查询的参数绑定，通常，这些都是 `where` 字句约束中的值。参数绑定可以避免 SQL 注入攻击。

`select` 方法以数组的形式返回结果集，数组中的每一个结果都是一个 PHP `StdClass` 对象，从而允许你像下面这样访问结果值：

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

使用命名绑定

除了使用`?`占位符来代表参数绑定外，还可以使用命名绑定来执行查询：

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

运行插入语句

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

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

```
']);
```

运行更新语句

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

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

运行删除语句

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

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

运行一个通用语句

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

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

监听查询事件

如果你想要获取应用中每次 SQL 语句的执行，可以使用 `listen` 方法，该方法对查询日志和调试

非常有用，你可以在[服务提供者](#)中注册查询监听器：

```
<?php  
  
namespace App\Providers;  
  
use Illuminate\Support\Facades\DB;  
use Illuminate\Support\ServiceProvider;  
  
class AppServiceProvider extends ServiceProvider
```

```
{  
    /**  
     * Bootstrap any application services.  
     *  
     * @return void  
     */  
  
    public function boot()  
    {  
  
        DB::listen(function ($query) {  
            // $query->sql  
            // $query->bindings  
            // $query->time  
        });  
    }  
  
    /**  
     * Register the service provider.  
     *  
     * @return void  
     */  
  
    public function register()  
    {  
        //  
    }  
}
```

3、数据库事务

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

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

手动使用事务

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

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

你可以通过 `rollBack` 方法回滚事务：

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

最后，你可以通过 `commit` 方法提交事务：

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

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

10.2 查询构建器

1、简介

[数据库查询构建器](#)提供了一个方便的流接口用于创建和执行数据库[查询](#)。查询构建器可以用于执

行应用中大部分数据库操作，并且能够在支持的所有数据库系统上工作。

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

2、获取结果集

从一张表中取出所有行

我们可以从 DB 门面的 table 方法开始，table 方法为给定表返回一个流式查询构建器实例，该实例允许你在查询上链接多个约束条件并最终返回查询结果。在本例中，我们使用 get 方法获取表中所有记录：

```
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller{

    /**
     * 显示用户列表
     *
     * @return Response
     */

    public function index()
```

```
{  
    $users = DB::table('users')->get();  
  
    return view('user.index', ['users' => $users]);  
}  
}
```

`get` 方法返回包含结果集的 `Illuminate\Support\Collection`，其中每一个结果都是 PHP 的 `StdClass` 对象实例。你可以像访问对象的属性一样访问字段的值：

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

从一张表中获取一行/一列

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

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

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

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

获取数据列值列表

如果想要获取包含单个列值的数组，可以使用 `pluck` 方法，在本例中，我们获取角色标题数组：

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

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

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

```
$roles = DB::table('roles')->pluck('title', 'name');  
  
foreach ($roles as $name => $title) {  
    echo $title;  
}
```

组块结果集

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

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

你可以通过从闭包函数中返回 `false` 来终止组块的运行：

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

聚合函数

查询构建器还提供了多个聚合方法，如 `count`, `max`, `min`, `avg` 和 `sum`，你可以在构造查询之后调用这些方法：

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

当然，你可以联合其它查询子句和聚合函数来构建查询：

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

3、查询 (Select)

指定查询子句

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

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

`distinct` 方法允许你强制查询返回不重复的结果集：

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

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

用 `addSelect` 方法：

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

4、原生表达式

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

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

5、连接 (join)

内连接 (等值连接)

查询构建器还可以用于编写基本的 SQL“内连接”，你可以使用查询构建器实例上的 `join` 方法，传递给 `join` 方法的第一个参数是你需要连接到的表名 剩余的其它参数则是为连接指定的列约束，当然，正如你所看到的，你可以在单个查询中连接多张表：

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

左连接

如果你是想要执行“左连接”而不是“内连接”，可以使用 `leftJoin` 方法。该方法和 `join` 方法的用

法一样：

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

交叉连接

要执行“交叉连接”可以使用 `crossJoin` 方法，传递你想要交叉连接的表名到该方法即可。交叉连接在第一张表和被连接表之间生成一个笛卡尔积：

```
$users = DB::table('sizes')
    ->crossJoin('colours')
    ->get();
```

高级连接语句

你还可以指定更多的高级连接子句，传递一个闭包到 `join` 方法作为该方法的第二个参数，该闭包将会返回允许你指定 `join` 子句约束的 `JoinClause` 对象：

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

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

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

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

6、联合 (union)

查询构建器还提供了“联合”两个查询的快捷方式，比如，你可以先创建一个查询，然后使用 `union` 方法将其和第二个查询进行联合：

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

`unionAll` 方法也是有效的，并且和 `union` 有同样的使用方式。

7、Where 子句

简单 where 子句

使用查询构建器上的 `where` 方法可以添加 `where` 子句到查询中，调用 `where` 最基本的方法需要传递三个参数，第一个参数是列名，第二个参数是任意一个数据库系统支持的操作符，第三个参数是该列要比较的值。

例如，下面是一个验证“votes”列的值是否等于 100 的查询：

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

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

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

当然，你还可以使用其它操作符来编写 `where` 子句：

```
$users = DB::table('users')

->where('votes', '>=', 100)

->get();

$users = DB::table('users')

->where('votes', '<>', 100)

->get();

$users = DB::table('users')

->where('name', 'like', 'T%')

->get();
```

还可以传递条件数组到 `where` 函数：

```
$users = DB::table('users')->where([
    ['status', '=', '1'],
    ['subscribed', '<>', '1'],
])->get();
```

or 语句

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

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

更多 Where 子句

whereBetween

`whereBetween` 方法验证列值是否在给定值之间：

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

whereNotBetween

`whereNotBetween` 方法验证列值不在给定值之间：

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

whereIn/whereNotIn

`whereIn` 方法验证给定列的值是否在给定数组中：

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

`whereNotIn` 方法验证给定列的值不在给定数组中：

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

whereNull/whereNotNull

`whereNull` 方法验证给定列的值为 NULL：

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

whereNotNull 方法验证给定列的值不是 NULL：

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

whereDate / whereMonth / whereDay / whereYear

whereDate 方法用于比较字段值和日期：

```
$users = DB::table('users')
    ->whereDate('created_at', '2016-10-10')
    ->get();
```

whereMonth 方法用于比较字段值和一年中的指定月份：

```
$users = DB::table('users')
    ->whereMonth('created_at', '10')
    ->get();
```

whereDay 方法用于比较字段值和一月中的制定天：

```
$users = DB::table('users')
    ->whereDay('created_at', '10')
    ->get();
```

whereYear 方法用于比较字段值和指定年：

```
$users = DB::table('users')
```

```
->whereYear('created_at', '2016')  
->get();
```

whereColumn

`whereColumn` 方法用于验证两个字段是否相等：

```
$users = DB::table('users')  
->whereColumn('first_name', 'last_name')  
->get();
```

还可以传递一个比较运算符到该方法：

```
$users = DB::table('users')  
->whereColumn('updated_at', '>', 'created_at')  
->get();
```

还可以传递多条件数组到 `whereColumn` 方法，这些条件通过 `and` 操作符进行连接：

```
$users = DB::table('users')  
->whereColumn([  
    ['first_name', '=', 'last_name'],  
    ['updated_at', '>', 'created_at']  
)->get();
```

参数分组

有时候你需要创建更加高级的 `where` 子句，比如“where exists”或者嵌套的参数分组。Laravel 查

询构建器也可以处理这些。作为开始，让我们看一个在括号中进行分组约束的例子：

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

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

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

where exists 子句

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

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

上述查询等价于下面的 SQL 语句：

```
select * from users
```

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

JSON Where 子句

Laravel 还支持在提供 JSON 字段类型的数据库（目前是 MySQL 5.7 和 Postgres）上使用操作符 `->`

查询 JSON 字段类型：

```
$users = DB::table('users')
    ->where('options->language', 'en')
    ->get();

$users = DB::table('users')
    ->where('preferences->dining->meal', 'salad')
    ->get();
```

8、排序、分组、限定

orderBy

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

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

inRandomOrder

`inRandomOrder` 方法可用于对查询结果集进行随机排序，比如，你可以用该方法获取一个随机用户：

```
$randomUser = DB::table('users')
    ->inRandomOrder()
    ->first();
```

groupBy / having / havingRaw

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

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

`havingRaw` 方法可用于设置原生字符串作为 `having` 子句的值，例如，我们可以这样找到所有售价大于 \$2500 的部分：

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

skip / take

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

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

作为替代方法，还可以使用 `limit` 和 `offset` 方法：

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

9、条件子句

有时候你可能想要某些条件为 `true` 的时候才讲条件子句应用到查询。例如，你可能只想给定值

在请求中存在的情况下才应用 `where` 语句，这可以通过 `when` 方法实现：

```
$role = $request->input('role');  
  
$users = DB::table('users')  
    ->when($role, function ($query) use ($role) {  
        return $query->where('role_id', $role);  
    })  
    ->get();
```

`when` 方法只有在第一个参数为 `true` 的时候才执行给定闭包，如果第一个参数为 `false`，则闭包

不执行。

你可以传递另一个闭包作为 `when` 方法的第三个参数，该闭包会在第一个参数为 `false` 的情况下

执行。为了演示这个特性如何使用，我们来配置一个查询的默认排序：

```
$sortBy = null;  
  
$users = DB::table('users')
```

```
->when($sortBy, function ($query) use ($sortBy) {
    return $query->orderBy($sortBy);
}, function ($query) {
    return $query->orderBy('name');
})
->get();
```

10、插入 (Insert)

查询构建器还提供了 `insert` 方法用于插入记录到数据表。`insert` 方法接收数组形式的字段名和字段值进行插入操作：

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

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

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

自增 ID

如果数据表有自增 ID，使用 `insertGetId` 方法来插入记录并返回 ID 值：

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

```
);
```

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

11、更新 (Update)

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

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

更新 JSON 字段

更新 JSON 字段的时候，需要使用 `->` 语法访问 JSON 对象上相应的值，该操作只能用于支持 JSON 字段类型的数据库：

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

增加/减少

查询构建器还为增减给定字段名对应数值提供方便。相较于编写 `update` 语句，这是一条捷径，提供了更好的体验和测试接口。

这两个方法都至少接收一个参数 :需要修改的列。第二个参数是可选的 , 用于控制列值增加/减少的数目。

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

在操作过程中你还可以指定额外的列进行更新 :

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

12、删除 (Delete)

当然 , 查询构建器还可以通过 `delete` 方法从表中删除记录 :

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

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

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

13、悲观锁

查询构建器还提供了一些方法帮助你在 `select` 语句中实现“悲观锁”。可以在查询中使用

`sharedLock` 方法从而在运行语句时带一把“共享锁”。共享锁可以避免被选择的行被修改直到事务提交 :

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

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

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

10.3 分页

1、简介

在其他框架中，分页是件非常痛苦的事，Laravel的分页器集成了查询构建器和Eloquent ORM，并且开箱提供了方便的、易于使用的、基于数据库结果集的分页。分页器生成的 HTML 兼容 Bootstrap CSS 框架。

2、基本使用

基于查询构建器进行分页

有多种方式实现分页，最简单的方式就是使用查询构建器或 Eloquent 模型的 `paginate` 方法。该方法基于当前用户查看页自动设置合适的偏移（offset）和限制（limit）。默认情况下，当前页通过 HTTP 请求查询字符串参数`?page` 的值判断。当然，该值由 Laravel 自动检测，然后自动插入分页器生成的链接中。

让我们先来看看如何在查询上调用 `paginate` 方法。在本例中，传递给 `paginate` 的唯一参数就是你每页想要显示的数目，这里我们指定每页显示 15 个：

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

```
use Illuminate\Support\Facades\DB;  
use App\Http\Controllers\Controller;  
  
class UserController extends Controller{  
    /**  
     * 显示应用中的所有用户  
     *  
     * @return Response  
     */  
    public function index()  
    {  
        $users = DB::table('users')->paginate(15);  
        return view('user.index', ['users' => $users]);  
    }  
}
```

注：目前，使用 `groupBy` 的分页操作不能被 Laravel 有效执行，如果你需要在分页结果中使用 `groupBy`，推荐你手动查询数据库然后创建分页器。

简单分页

如果你只需要在分页视图中简单的显示“下一页”和“上一页”链接，可以使用 `simplePaginate` 方法来执行一个更加高效的查询。在渲染包含大数据集的视图且不需要显示每个页码时这一功能非常有用：

```
$users = DB::table('users')->simplePaginate(15);
```

基于 Eloquent 结果集进行分页

你还可以对 Eloquent 查询结果进行分页，在本例中，我们对 `User` 模型进行分页，每页显示 15 条记录。正如你所看到的，该语法和基于查询构建器的分页差不多：

```
$users = App\User::paginate(15);
```

当然，你可以在设置其它约束条件之后调用 `paginate`，比如 `where` 子句：

```
$users = User::where('votes', '>', 100)->paginate(15);
```

在对 Eloquent 模型进行分页时你也可以使用 `simplePaginate` 方法：

```
$users = User::where('votes', '>', 100)->simplePaginate(15);
```

手动创建分页器

有时候你可能想要通过传递数组数据来手动创建分页实例，你可以基于自己的需求通过创建 `Illuminate\Pagination\Paginator` 或 `Illuminate\Pagination\LengthAwarePaginator` 实例来实现。

`Paginator` 类不需要知道结果集中数据项的总数；不过，正因如此，该类也没有提供获取最后一页索引的方法。

`LengthAwarePaginator` 接收参数和 `Paginator` 几乎一样，只是，它要求传入结果集的总数。

换句话说，`Paginator` 对应 `simplePaginate` 方法，而 `LengthAwarePaginator` 对应 `paginate` 方法。

注：当手动创建分页器实例的时候，应该手动对传递到分页器的结果集进行“切片”，如果你不确定怎么做，查看 PHP 函数 [array_slice](#)。

3、显示分页结果

当调用 `paginate` 方法时，你将获取 `Illuminate\Pagination\LengthAwarePaginator` 实例，而调用方法 `simplePaginate` 时，将会获取 `Illuminate\Pagination\Paginator` 实例。这些对象提供相关方法描述这些结果集，除了这些辅助函数外，分页器实例本身就是迭代器，可以像数组一样对其进行循环调用。

所以，获取到结果后，可以按如下方式使用 `Blade` 显示这些结果并渲染页面链接：

```
<div class="container">  
    @foreach ($users as $user)  
        {{ $user->name }}  
    @endforeach  
    </div>  
  
    {{ $users->links() }}  
</pre>
```

`links` 方法将会将结果集中的其它页面链接渲染出来。每个链接已经包含了 `?page` 查询字符串变量。记住，`render` 方法生成的 HTML 兼容 [Bootstrap CSS 框架](#)。

自定义分页链接

`setPath` 方法允许你生成分页链接时自定义分页器使用的 URI，例如，如果你想要分页器生成形如 `http://example.com/custom/url?page=N` 的链接，应该传递 `custom/url` 到 `setPath` 方法：

```
Route::get('users', function () {
    $users = App\User::paginate(15);
    $users->setPath('custom/url');
    //
});
```

添加参数到分页链接

你可以使用 `appends` 方法添加查询参数到分页链接查询字符串。例如，要添加 `&sort=votes` 到每个分页链接，应该像如下方式调用 `appends`：

```
{{ $users->appends(['sort' => 'votes'])->links() }}
```

如果你想要添加“哈希片段”到分页链接，可以使用 `fragment` 方法。例如，要添加 `#foo` 到每个分页链接的末尾，像这样调用 `fragment` 方法：

```
{{ $users->fragment('foo')->links() }}
```

将结果转化为 JSON

Laravel 分页器结果类实现了 `Illuminate\Contracts\Support\JsonableInterface` 契约并提供了 `toJson` 方法，所以将分页结果转化为 JSON 非常简单。你还可以简单通过从路由或控制器动作返回分页器实例将转其化为 JSON：

```
Route::get('users', function () {
    return App\User::paginate();
});
```

从分页器转化来的 JSON 包含了元信息如 `total`、`current_page`、`last_page` 等等，实际的结果对象数据可以通过该 JSON 数组中的 `data` 键访问。下面是一个通过从路由返回的分页器实例创建的

JSON 例子：

```
{  
    "total": 50,  
    "per_page": 15,  
    "current_page": 1,  
    "last_page": 4,  
    "next_page_url": "http://laravel.app?page=2",  
    "prev_page_url": null,  
    "from": 1,  
    "to": 15,  
    "data": [  
        {  
            // Result Object  
        },  
        {  
            // Result Object  
        }  
    ]  
}
```

4、自定义分页视图

默认情况下，用于渲染分页链接的视图兼容于 Bootstrap CSS 框架，如果你没有使用 Bootstrap，可以自定义视图来渲染这些链接。当调用分页器实例上的 `links` 方法时，传递视图名称作为第一个参数：

```
 {{ $paginator->links('view.name') }}
```

不过，自定义分页视图最简单的方式是使用 `vendor:publish` 命令导出视图文件到 `resources/views/vendor` 目录：

```
php artisan vendor:publish --tag=laravel-pagination
```

该命令会将视图放到 `resources/views/vendor/pagination` 目录，该目录下的 `default.blade.php` 文件对应默认的视图文件，编辑该文件即可修改分页 HTML。

5、分页器实例方法

每个分页器实例都可以通过以下方法提供更多分页信息：

- `$results->count()`
- `$results->currentPage()`
- `$results->firstItem()`
- `$results->hasMorePages()`
- `$results->lastItem()`
- `$results->lastPage()` (使用 `simplePaginate` 时无效)
- `$results->nextPageUrl()`
- `$results->perPage()`
- `$results->previousPageUrl()`
- `$results->total()` (使用 `simplePaginate` 时无效)
- `$results->url($page)`

10.4 迁移

1、简介

迁移就像数据库的版本控制，允许团队简单轻松的编辑并共享应用的数据库表结构，迁移通常和 Laravel 的 schema 构建器结对从而可以很容易地构建应用的数据库表结构。如果你曾经告知小组成员需要手动添加列到本地数据库结构，那么这正是数据库迁移所致力于解决的问题。

Laravel 的 Schema 门面提供了与数据库系统无关的创建和操纵表的支持，在 Laravel 所支持的所有数据库系统中提供一致的、优雅的、平滑的 API。

2、生成迁移

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

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

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

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

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

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

3、迁移结构

迁移类包含了两个方法：`up` 和 `down`。`up` 方法用于新增表，列或者索引到数据库，而 `down` 方法就

是 `up` 方法的反操作，和 `up` 里的操作相反。

在这两个方法中你都要用到 Laravel 的 schema 构建器来创建和修改表，要了解更多 Schema 构建器提供的方法，参考其[文档](#)。下面让我们先看看创建 `flights` 表的简单示例：

```
<?php

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

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

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

4、运行迁移

要运行应用中所有未执行的迁移，可以使用 Artisan 命令提供的 `migrate` 方法：

```
php artisan migrate
```

注：如果你正在使用 [Homestead 虚拟机](#)，需要在虚拟机中运行上面这条命令。

在生产环境中强制运行迁移

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

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

回滚迁移

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

```
php artisan migrate:rollback
```

你也可以通过 `rollback` 命令上提供的 `step` 选项来回滚指定数目的迁移，例如，下面的命令将会回滚最后五条迁移：

```
php artisan migrate:rollback --step=5
```

`migrate:reset` 命令将会回滚所有的应用迁移：

```
php artisan migrate:reset
```

在单个命令中回滚/迁移

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

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

当然，你也可以回滚或重建指定数量的迁移，通过 `refresh` 命令提供的 `step` 选项，例如，下面的命令将会回滚或重建最后五条迁移：

```
php artisan migrate:refresh --step=5
```

5、数据表

创建表

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

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

当然，创建新表的时候，可以使用 `schema` 构建器中的任意列方法来定义数据表的列。

检查表/列是否存在

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

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

连接&存储引擎

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

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

要设置表的存储引擎，在 schema 构建器上设置 `engine` 属性：

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

重命名/删除表

要重命名一个已存在的数据表，使用 `rename` 方法：

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

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

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

通过外键重命名表

在重命名表之前，需要验证该表包含的外键在迁移文件中有明确的名字，而不是 Laravel 基于惯例分配的名字。否则，外键约束名将会指向旧的数据表。

6、列

创建列

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

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

可用的列类型

当然，schema 构建器包含一系列你可以用来构建表的列类型：

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

命令	描述
<code>\$table->boolean('confirmed');</code>	等同于数据库中的 BOOLEAN 类型
<code>\$table->char('name', 4);</code>	等同于数据库中的 CHAR 类型
<code>\$table->date('created_at');</code>	等同于数据库中的 DATE 类型
<code>\$table->dateTime('created_at');</code>	等同于数据库中的 DATETIME 类型
<code>\$table->dateTimeTz('created_at');</code>	等同于数据库中的 DATETIME 类型 (带时区)
<code>\$table->decimal('amount', 5, 2);</code>	等同于数据库中的 DECIMAL 类型 , 带一个精度和范围
<code>\$table->double('column', 15, 8);</code>	等同于数据库中的 DOUBLE 类型 , 带精度, 总共 15 位数字 , 小数后 8 位.
<code>\$table->enum('choices', ['foo', 'bar']);</code>	等同于数据库中的 ENUM 类型
<code>\$table->float('amount');</code>	等同于数据库中的 FLOAT 类型
<code>\$table->increments('id');</code>	数据库主键自增 ID
<code>\$table->integer('votes');</code>	等同于数据库中的 INTEGER 类型
<code>\$table->ipAddress('visitor');</code>	等同于数据库中的 IP 地址

命令	描述
<code>\$table->json('options');</code>	等同于数据库中的 JSON 类型
<code>\$table->jsonb('options');</code>	等同于数据库中的 JSONB 类型
<code>\$table->longText('description');</code>	等同于数据库中的 LONGTEXT 类型
<code>\$table->macAddress('device');</code>	等同于数据库中的 MAC 地址
<code>\$table->mediumIncrements('id');</code>	自增 ID，类型为无符号的 mediumint
<code>\$table->mediumInteger('numbers');</code>	等同于数据库中的 MEDIUMINT 类型
<code>\$table->mediumText('description');</code>	等同于数据库中的 MEDIUMTEXT 类型
<code>\$table->morphs('taggable');</code>	添加一个 INTEGER 类型的 <code>taggable_id</code> 列和一个 STRING 类型的 <code>taggable_type</code> 列
<code>\$table->nullableTimestamps();</code>	和 <code>timestamps()</code> 一样但允许 NULL 值.
<code>\$table->rememberToken();</code>	添加一个 <code>remember_token</code> 列： VARCHAR(100) NULL.
<code>\$table->smallIncrements('id');</code>	自增 ID，类型为无符号的 smallint
<code>\$table->smallInteger('votes');</code>	等同于数据库中的 SMALLINT 类型

命令	描述
<code>\$table->softDeletes();</code>	新增一个 <code>deleted_at</code> 列 用于软删除.
<code>\$table->string('email');</code>	等同于数据库中的 VARCHAR 列 .
<code>\$table->string('name', 100);</code>	等同于数据库中的 VARCHAR , 带一个长度
<code>\$table->text('description');</code>	等同于数据库中的 TEXT 类型
<code>\$table->time('sunrise');</code>	等同于数据库中的 TIME 类型
<code>\$table->timeTz('sunrise');</code>	等同于数据库中的 TIME 类型 (带时区)
<code>\$table->tinyInteger('numbers');</code>	等同于数据库中的 TINYINT 类型
<code>\$table->timestamp('added_on');</code>	等同于数据库中的 TIMESTAMP 类型
<code>\$table->timestampTz('added_on');</code>	等同于数据库中的 TIMESTAMP 类型 (带时区)
<code>\$table->timestamps();</code>	添加 <code>created_at</code> 和 <code>updated_at</code> 列
<code>\$table->timestampsTz();</code>	添加 <code>created_at</code> 和 <code>updated_at</code> 列 (带时区)
<code>\$table->unsignedBigInteger('votes');</code>	等同于数据库中无符号的 BIGINT 类型

命令	描述
<code>\$table->unsignedInteger('votes');</code>	等同于数据库中无符号的 INT 类型
<code>\$table->unsignedMediumInteger('votes');</code>	等同于数据库中无符号的 MEDIUMINT 类型
<code>\$table->unsignedSmallInteger('votes');</code>	等同于数据库中无符号的 SMALLINT 类型
<code>\$table->unsignedTinyInteger('votes');</code>	等同于数据库中无符号的 TINYINT 类型
<code>\$table->uuid('id');</code>	等同于数据库的 UUID

列修改器

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

认为 `null`，可以使用 `nullable` 方法：

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

下面是所有可用的列修改器列表，该列表不包含索引修改器：

修改器	描述
<code>->after('column')</code>	将该列置于另一个列之后 (仅适用于 MySQL)
<code>->comment('my comment')</code>	添加注释信息

修改器	描述
<code>->default(\$value)</code>	指定列的默认值
<code>->first()</code>	将该列置为表中第一个列 (仅适用于 MySQL)
<code>->nullable()</code>	允许该列的值为 NULL
<code>->storedAs(\$expression)</code>	创建一个存储生成列 (只支持 MySQL)
<code>->unsigned()</code>	设置 <code>integer</code> 列为 <code>UNSIGNED</code>
<code>->virtualAs(\$expression)</code>	创建一个虚拟生成列 (只支持 MySQL)

修改列

先决条件

在修改列之前，确保已经将 `doctrine/dbal` 依赖添加到 `composer.json` 文件，Doctrine DBAL 库

用于判断列的当前状态并创建对列进行指定调整所需的 SQL 语句：

```
composer require doctrine/dbal
```

更新列属性

`change` 方法允许你修改已存在的列为新的类型，或者修改列的属性。例如，你可能想要增加 `string`

类型列的尺寸，让我们将 `name` 列的尺寸从 25 增加到 50：

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

我们还可以修改该列允许 NULL 值：

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

重命名列

要重命名一个列，可以使用表结构构建器上的 `renameColumn` 方法，在重命名一个列之前，确保

`doctrine/dbal` 依赖已经添加到 `composer.json` 文件：

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

注：暂不支持 `enum` 类型的列的修改和重命名。

删除列

要删除一个列，使用 schema 构建器上的 `dropColumn` 方法：

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

你可以传递列名数组到 `dropColumn` 方法从表中删除多个列：

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

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

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

7、索引

创建索引

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

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

此外，你可以在定义列之后创建索引，例如：

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

你甚至可以传递列名数组到索引方法来创建组合索引：

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

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

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

可用索引类型

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

删除索引

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

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

如果要传递列数组到删除索引方法，那么相应的索引名称将会通过数据表名、列和关键类型来自动生成：

```
Schema::table('geo', function ($table) {
    $table->dropIndex(['state']); // Drops index 'geo_state_index'
});
```

外键约束

Laravel 还提供了创建外键约束的支持，用于在数据库层面强制引用完整性。例如，我们在 `posts`

表中定义了一个引用 `users` 表的 `id` 列的 `user_id` 列：

```
Schema::table('posts', function ($table) {
    $table->integer('user_id')->unsigned();
    $table->foreign('user_id')->references('id')->on('users');
});
```

你还可以为约束的“on delete”和“on update”属性指定期望的动作：

```
$table->foreign('user_id')
    ->references('id')->on('users')
    ->onDelete('cascade');
```

要删除一个外键，可以使用 `dropForeign` 方法。外键约束和索引使用同样的命名规则——连接表名、外键名然后加上“_foreign”后缀：

```
$table->dropForeign('posts_user_id_foreign');
```

或者，你还可以传递在删除时会自动使用基于惯例的约束名数值数组：

```
$table->dropForeign(['user_id']);
```

你可以在迁移时通过以下方法启用或关闭外键约束：

```
Schema::enableForeignKeyConstraints();  
Schema::disableForeignKeyConstraints();
```

10.5 填充数据

1、简介

Laravel 使用填充类和测试数据提供了一个简单方法来填充数据到数据库。所有的填充类都位于 database/seeds 目录。填充类的类名完全由你自定义，但最好还是遵循一定的规则，比如可读性，例如 UserTableSeeder 等等。安装完 Laravel 后，会默认提供一个 DatabaseSeeder 类。从这个类中，你可以使用 call 方法来运行其他填充类，从而允许你控制填充顺序。

2、编写填充器

要生成一个填充器，可以通过 Artisan 命令 make:seeder。所有框架生成的填充器都位于 database/seeders 目录：

```
php artisan make:seeder UserTableSeeder
```

一个填充器类默认只包含一个方法：run。当 Artisan 命令 db:seed 运行时该方法被调用。在 run 方法中，可以插入任何你想插入数据库的数据，你可以使用查询构建器手动插入数据，也可以使用 Eloquent 模型工厂。

举个例子，让我们修改 Laravel 安装时自带的 DatabaseSeeder 类，添加一个数据库插入语句到

run 方法：

```
<?php
```

```
use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;

class DatabaseSeeder extends Seeder{
    /**
     * 运行数据库填充
     *
     * @return void
     */
    public function run()
    {
        DB::table('users')->insert([
            'name' => str_random(10),
            'email' => str_random(10).'@gmail.com',
            'password' => bcrypt('secret'),
        ]);
    }
}
```

使用模型工厂

当然，手动指定每一个模型填充的属性是很笨重累赘的，取而代之的，我们可以使用模型工厂来方便的生成大量的数据库记录。首先，查看[模型工厂文档](#)来学习如何定义工厂，定义工厂后，可以使用帮助函数 `factory` 来插入记录到数据库。

举个例子，让我们创建 50 个用户并添加关联关系到每个用户：

```
/**
 * 运行数据库填充
 *
 * @return void
 */
public function run(){
    factory('App\User', 50)->create()->each(function($u) {
        $u->posts()->save(factory('App\Post')->make());
    });
}
```

调用额外的填充器

在 `DatabaseSeeder` 类中，你可以使用 `call` 方法执行额外的填充类，使用 `call` 方法允许你将数据库填充分解成多个文件，这样单个填充器类就不会变得无比巨大，只需简单将你想要运行的填充器类名传递过去即可：

```
/**  
 * 运行数据库填充  
 *  
 * @return void  
 */  
  
public function run(){  
    $this->call(UserTableSeeder::class);  
    $this->call(PostsTableSeeder::class);  
    $this->call(CommentsTableSeeder::class);  
}
```

3、运行填充器

编写好填充器类之后，可以使用 Artisan 命令 `db:seed` 来填充数据库。默认情况下，`db:seed` 命令运行可以用来运行其它填充器类的 `DatabaseSeeder` 类，但是，你也可以使用 `--class` 选项来指定你想要运行的独立的填充器类：

```
php artisan db:seed  
php artisan db:seed --class=UserTableSeeder
```

你还可以使用 `migrate:refresh` 命令来填充数据库，该命令还可以回滚并重新运行迁移，这在需

要完全重建数据库时很有用：

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

10.6 Redis

1、简介

[Redis](#) 是一个开源的、高级的键值对存储系统，经常被用作数据结构服务器，因为其支持字符串、Hash、列表、集合和有序集合等数据结构。在 [Laravel](#) 中使用 [Redis](#) 之前，需要通过 Composer 安装 [predis/predis](#) 包：

```
composer require predis/predis
```

配置

应用的 Redis 配置位于配置文件 [config/database.php](#)。在这个文件中，可以看到包含被应用使用的 Redis 服务器的 [redis](#) 数组：

```
'redis' => [  
  
    'cluster' => false,  
  
    'default' => [  
        'host'      => '127.0.0.1',  
        'port'      => 6379,  
        'database'  => 0,  
    ],
```

],

默认服务器配置可以满足开发需要，不过，你可以基于自己的环境修改该数组。配置文件中定义的每个 Redis 服务器需要一个名字并指定该 Redis 服务器使用的主机和接口。

`cluster` 选项告知 Laravel Redis 客户端在多个 Redis 节点间执行客户端分片，从而形成节点池并创建大量有效的 RAM。然而，客户端分片并不处理故障转移，所以，非常适合从另一个主数据存储那里获取有效的缓存数据。

此外，你可以在 Redis 连接定义中定义 `options` 数组值，从而允许你指定一系列 Predis [客户端选项](#)。

如果 Redis 服务器要求认证信息，你可以通过添加 `password` 配置项到 Redis 服务器配置数组来提供密码。

注意：如果你通过 PECL 安装 PHP 的 Redis 扩展，需要在 `config/app.php` 文件中修改 Redis 的别名。

2、与 Redis 交互

你可以通过调用 [Redis 门面上的方法](#) 来与 Redis 进行交互，该门面支持动态方法，所以你可以调用任何 [Redis 命令](#)，对应命令将会直接传递给 Redis，在本例中，我们通过调用 [Redis 门面上的 get 方法](#) 来调用 Redis 上的 GET 命令：

```
<?php
```

```
namespace App\Http\Controllers;

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

class UserController extends Controller{

    /**
     * 显示指定用户属性
     *
     * @param int $id
     * @return Response
     */

    public function showProfile($id)
    {
        $user = Redis::get('user:profile:'.$id);
        return view('user.profile', ['user' => $user]);
    }
}
```

当然，如上所述，可以在 `Redis` 门面上调用任何 Redis 命令。Laravel 使用魔术方法将命令传递给

Redis 服务器，所以只需简单传递参数和 Redis 命令如下：

```
Redis::set('name', 'Taylor');
$values = Redis::lrange('names', 5, 10);
```

此外还可以使用 `command` 方法传递命令到服务器，该方法接收命令名作为第一个参数，参数值数组作为第二个参数：

```
$values = Redis::command('lrange', ['name', 5, 10]);
```

使用多个 Redis 连接

你可以通过调用 `Redis::connection` 方法获取 Redis 实例：

```
$redis = Redis::connection();
```

这将会获取默认 Redis 服务器实例。如果你没有使用服务器集群，可以传递服务器名到 `connection`

方法来获取指定 Redis 配置中定义的指定服务器：

```
$redis = Redis::connection('other');
```

管道命令

当你需要在一次操作中发送多个命令到服务器的时候应该使用管道，`pipeline` 方法接收一个参数：

接收 Redis 实例的闭包。你可以将所有 Redis 命令发送到这个 Redis 实例，然后这些命令会在一次操作中被执行：

```
Redis::pipeline(function ($pipe) {
    for ($i = 0; $i < 1000; $i++) {
        $pipe->set("key:$i", $i);
    }
});
```

3、发布/订阅

Redis 还提供了调用 Redis 的 `publish` 和 `subscribe` 命令的接口。这些 Redis 命令允许你在给定“频道”监听消息，你可以从另外一个应用发布消息到这个频道，甚至使用其它编程语言，从而允许你在不同的应用/进程之间轻松通信。

首先，让我们使用 `subscribe` 方法通过 Redis 在一个频道上设置监听器。由于调用 `subscribe` 方法会开启一个常驻进程，我们将在 Artisan 命令中调用该方法：

```
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;
use Illuminate\Support\Facades\Redis;

class RedisSubscribe extends Command{
    /**
     * 控制台命令名称
     *
     * @var string
     */
    protected $signature = 'redis:subscribe';

    /**
     * 控制台命令描述
     *
     * @var string
     */
    protected $description = 'Subscribe to a Redis channel';

    /**
     * 执行控制台命令
     *
     * @return mixed
     */
}
```

```
public function handle()
{
    Redis::subscribe(['test-channel'], function($message) {
        echo $message;
    });
}
```

现在，我们可以使用 `publish` 发布消息到该频道：

```
Route::get('publish', function () {
    // 路由逻辑...
    Redis::publish('test-channel', json_encode(['foo' => 'bar']));
});
```

通配符订阅

使用 `psubscribe` 方法，你可以订阅到一个通配符定义的频道，这在所有相应频道上获取所有消息时很有用。`$channel` 名将会作为第二个参数传递给提供的回调闭包：

```
Redis::psubscribe(['*'], function($message, $channel) {
    echo $message;
});

Redis::psubscribe(['users.*'], function($message, $channel) {
    echo $message;
});
```

11. Eloquent ORM

11.1 起步

1、简介

Laravel 自带的 [Eloquent ORM](#) 提供了一个美观、简单的与[数据库](#)打交道的 ActiveRecord 实现，每张数据表都对应一个与该表进行交互的“[模型](#)”，模型允许你在表中进行数据查询，以及插入、更新、删除等操作。

在开始之前，确保在 `config/database.php` 文件中[配置](#)好了数据库连接。更多关于数据库配置的信息，请查看[文档](#)。

2、定义模型

作为开始，让我们创建一个 Eloquent 模型，模型通常位于 `app` 目录下，你也可以将其放在其他可以被 `composer.json` 文件自动加载的地方。所有 Eloquent 模型都继承自 `Illuminate\Database\Eloquent\Model` 类。

创建模型实例最简单的办法就是使用 Artisan 命令 `make:model`：

```
php artisan make:model User
```

如果你想要在生成模型时生成[数据库迁移](#)，可以使用 `--migration` 或 `-m` 选项：

```
php artisan make:model User --migration
php artisan make:model User -m
```

Eloquent 模型约定

现在，让我们来看一个 `Flight` 模型类例子，我们将用该类获取和存取数据表 `flights` 中的信息：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model{
```

```
//  
}
```

表名

注意我们并没有告诉 Eloquent 我们的 `Flight` 模型使用哪张表。默认规则是模型类名的复数作为与其对应的表名，除非在模型类中明确指定了其它名称。所以，在本例中，Eloquent 认为 `Flight` 模型存储记录在 `flights` 表中。你也可以在模型中定义 `table` 属性来指定自定义的表名：

```
<?php  
  
namespace App;  
  
use Illuminate\Database\Eloquent\Model;  
  
class Flight extends Model{  
    /**  
     * 关联到模型的数据表  
     *  
     * @var string  
     */  
    protected $table = 'my_flights';  
}
```

主键

Eloquent 默认每张表的主键名为 `id`，你可以在模型类中定义一个 `$primaryKey` 属性来覆盖该约定。

此外，Eloquent 默认主键字段是自增的整型数据，这意味着主键将会被自动转化为 `int` 类型，如果你想要使用非自增或非数字类型主键，必须在对应模型中设置 `$incrementing` 属性为 `false`。

时间戳

默认情况下，Eloquent 期望 `created_at` 和 `updated_at` 已经存在于数据表中，如果你不想要这些 Laravel 自动管理的列，在模型类中设置 `$timestamps` 属性为 `false`：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model{
    /**
     * 表明模型是否应该被打上时间戳
     *
     * @var bool
     */
    public $timestamps = false;
}
```

如果你需要自定义时间戳格式，设置模型中的`$dateFormat` 属性。该属性决定日期被如何存储到数据库中，以及模型被序列化为数组或 JSON 时日期的格式：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model{
    /**
     * 模型日期列的存储格式
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

数据库连接

默认情况下，所有的 Eloquent 模型使用应用配置中的默认数据库连接，如果你想要为模型指定不同的连接，可以通过`$connection` 属性来设置：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model{

    /**
     * The connection name for the model.
     *
     * @var string
     */

    protected $connection = 'connection-name';

}
```

3、获取模型

创建完模型及其关联的数据表后，就要准备从数据库中获取数据。将 Eloquent 模型看作功能强大的查询构建器，你可以使用它来流畅的查询与其关联的数据表。例如：

```
<?php

use App\Flight;

$flights = App\Flight::all();
```

```
foreach ($flights as $flight) {  
    echo $flight->name;  
}
```

添加额外约束

Eloquent 的 `all` 方法返回模型表的所有结果，由于每一个 Eloquent 模型都是一个查询构建器，

你还可以添加约束条件到查询，然后使用 `get` 方法获取对应结果：

```
$flights = App\Flight::where('active', 1)  
            ->orderBy('name', 'desc')  
            ->take(10)  
            ->get();
```

注意：由于 Eloquent 模型本质上就是查询构建器，你可以在 Eloquent 查询中使用查询构建器的
所有方法。

集合

对 Eloquent 中获取多个结果的方法（比如 `all` 和 `get`）而言，其返回值是

`Illuminate\Database\Eloquent\Collection` 的一个实例，`Collection` 类提供了多个有用的函数
来处理 Eloquent 结果集：

```
$flights = $flights->reject(function ($flight) {  
    return $flight->cancelled;  
});
```

当然，你也可以像数组一样循环遍历该集合：

```
foreach ($flights as $flight) {  
    echo $flight->name;  
}
```

组块结果集

如果你需要处理成千上万个 Eloquent 结果，可以使用 `chunk` 命令。`chunk` 方法会获取一个“组块”的 Eloquent 模型，并将其填充到给定闭包进行处理。使用 `chunk` 方法能够在处理大量数据集合时有效减少内存消耗：

```
Flight::chunk(200, function ($flights) {  
    foreach ($flights as $flight) {  
        //  
    }  
});
```

传递给该方法的第一个参数是你想要获取的“组块”数目，闭包作为第二个参数被调用用于处理每一个从数据库获取的区块数据。

使用游标

`cursor` 方法允许你使用游标迭代处理数据库记录，一次只执行单个查询，在处理大批量数据时，`cursor` 方法可大幅减少内存消耗：

```
foreach (Flight::where('foo', 'bar')->cursor() as $flight) {  
    //  
}
```

4、获取单个模型/聚合

当然，除了从给定表中获取所有记录之外，还可以使用 `find` 和 `first` 获取单个记录。这些方法返回单个模型实例而不是返回模型集合：

```
// 通过主键获取模型...
$flight = App\Flight::find(1);
// 获取匹配查询条件的第一个模型...
$flight = App\Flight::where('active', 1)->first();
```

还可以通过传递主键数组来调用 `find` 方法，这将会返回匹配记录集合：

```
$flights = App\Flight::find([1, 2, 3]);
```

Not Found 异常

有时候你可能想要在模型找不到的时候抛出异常，这在路由或控制器中非常有用，`findOrFail` 和 `firstOrFail` 方法会获取查询到的第一个结果。然而，如果没有任何查询结果，`Illuminate\Database\Eloquent\ModelNotFoundException` 异常将会被抛出：

```
$model = App\Flight::findOrFail(1);
$model = App\Flight::where('legs', '>', 100)->firstOrFail();
```

如果异常没有被捕获，那么 HTTP 404 响应将会被发送给用户，所以在使用这些方法的时候没有必要对返回 404 响应编写明确的检查：

```
Route::get('/api/flights/{id}', function ($id) {
    return App\Flight::findOrFail($id);
});
```

获取聚合

当然，你还可以使用查询构建器聚合方法，例如 `count`、`sum`、`max`，以及其他查询构建器提供的聚合方法。这些方法返回计算后的结果而不是整个模型实例：

```
$count = App\Flight::where('active', 1)->count();
$max = App\Flight::where('active', 1)->max('price');
```

5、插入/更新模型

插入

想要在数据库中插入新的记录，只需创建一个新的模型实例，设置模型的属性，然后调用 `save` 方法：

```
<?php

namespace App\Http\Controllers;

use App\Flight;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class FlightController extends Controller{
    /**
     * 创建一个新的航班实例
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Validate the request...

        $flight = new Flight;
```

```
$flight->name = $request->name;  
  
$flight->save();  
}  
}
```

在这个例子中，我们只是简单分配 HTTP 请求中的 `name` 参数值给 `App\Flight` 模型实例的那么属性，当我们调用 `save` 方法时，一条记录将会被插入数据库。`created_at` 和 `updated_at` 时间戳在 `save` 方法被调用时会自动被设置，所以没必要手动设置它们。

更新

`save` 方法还可以用于更新数据库中已存在的模型。要更新一个模型，应该先获取它，设置你想要更新的属性，然后调用 `save` 方法。同样，`updated_at` 时间戳会被自动更新，所以没必要手动设置其值：

```
$flight = App\Flight::find(1);  
$flight->name = 'New Flight Name';  
$flight->save();
```

更新操作还可以同时修改给定查询提供的多个模型实例，在本例中，所有有效且 `destination=San Diego` 的航班都被标记为延迟：

```
App\Flight::where('active', 1)  
    ->where('destination', 'San Diego')  
    ->update(['delayed' => 1]);
```

`update` 方法要求以数组形式传递键值对参数，代表着数据表中应该被更新的列。

注：通过 Eloquent 进行批量更新时，`saved` 和 `updated` 模型事件将不会在更新模型时触发。这是因为在进行批量更新时并没有从数据库获取模型。

批量赋值

还可以使用 `create` 方法保存一个新的模型。该方法返回被插入的模型实例。但是，在此之前，你需要指定模型的 `fillable` 或 `guarded` 属性，因为所有 Eloquent 模型都通过批量赋值（Mass Assignment）进行保护。

当用户通过 HTTP 请求传递一个不被期望的参数值时就会出现安全隐患，然后该参数以不被期望的方式修改数据库中的列值。例如，恶意用户通过 HTTP 请求发送一个 `is_admin` 参数，然后该参数映射到模型的 `create` 方法，从而允许用户将自己变成管理员。

所以，你应该在模型中定义哪些属性是可以进行赋值的，使用模型上的 `$fillable` 属性即可实现。

例如，我们设置 `Flight` 模型上的 `name` 属性可以被赋值：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model{
    /**
     * 可以被批量赋值的属性.
     *
     * @var array
     */
    protected $fillable = ['name'];
}
```

设置完可以被赋值的属性之后，我们就可以使用 `create` 方法在数据库中插入一条新的记录。

`create` 方法返回保存后的模型实例：

```
$flight = App\Flight::create(['name' => 'Flight 10']);
```

黑名单属性

`$fillable` 就像是可以被赋值属性的“白名单”，还可以选择使用`$guarded`。`$guarded` 属性包含你不想被赋值的属性数组。所以不被包含在其中的属性都是可以被赋值的，因此，`$guarded` 功能就像“黑名单”。当然，这两个属性你只能同时使用其中一个——而不能一起使用，因为它们是互斥的：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model{

    /**
     * 不能被批量赋值的属性
     *
     * @var array
     */
    protected $guarded = ['price'];

}
```

如果你想要让所有属性都是可批量赋值的，可以将`$guarded` 属性设置为空数组：

```
/**
 * The attributes that aren't mass assignable.
*
* @var array
```

```
*/  
protected $guarded = [];
```

其它创建方法

还有其它两种可以用来创建模型的方法：`firstOrCreate` 和 `firstOrNew`。`firstOrCreate` 方法先尝试通过给定列/值对在数据库中查找记录，如果没有找到的话则通过给定属性创建一个新的记录。

`firstOrNew` 方法和 `firstOrCreate` 方法一样先尝试在数据库中查找匹配的记录，如果没有找到，则返回一个的模型实例。注意通过 `firstOrNew` 方法返回的模型实例并没有持久化到数据库中，你还需要调用 `save` 方法手动持久化：

```
// 通过属性获取航班，如果不存在则创建...  
$flight = App\Flight::firstOrCreate(['name' => 'Flight 10']);  
  
// 通过属性获取航班，如果不存在初始化一个新的实例...  
$flight = App\Flight::firstOrNew(['name' => 'Flight 10']);
```

6、删除模型

要删除一个模型，调用模型实例上的 `delete` 方法：

```
$flight = App\Flight::find(1);  
$flight->delete();
```

通过主键删除模型

在上面的例子中，我们在调用 `delete` 方法之前从数据库中获取该模型，然而，如果你知道模型的主键的话，可以调用 `destroy` 方法直接删除而不需要获取它：

```
App\Flight::destroy(1);
App\Flight::destroy([1, 2, 3]);
App\Flight::destroy(1, 2, 3);
```

通过查询删除模型

当然，你还可以通过查询删除多个模型，在本例中，我们删除所有被标记为无效的航班：

```
$deletedRows = App\Flight::where('active', 0)->delete();
```

注：通过 Eloquent 进行批量删除时，`deleting` 和 `deleted` 模型事件在删除模型时不会被触发，

这是因为在进行模型删除时不会获取模型。

软删除

除了从数据库删除记录外，Eloquent 还可以对模型进行“软删除”。当模型被软删除后，它们并没有真的从数据库删除，而是在模型上设置一个 `deleted_at` 属性并插入数据库，如果模型有一个非空 `deleted_at` 值，那么该模型已经被软删除了。要启用模型的软删除功能，可以使用模型上的

`Illuminate\Database\Eloquent\SoftDeletes` trait 并添加 `deleted_at` 列到 `$dates` 属性：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Flight extends Model{
    use SoftDeletes;

    /**
     * 应该被调整为日期的属性
     *
     * @var array
     */
}
```

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

当然，应该添加 `deleted_at` 列到数据表。 Laravel Schema 构建器包含一个帮助函数来创建该列：

```
Schema::table('flights', function ($table) {
    $table->softDeletes();
});
```

现在，当调用模型的 `delete` 方法时，`deleted_at` 列将被设置为当前日期和时间，并且，当查询一个使用软删除的模型时，被软删除的模型将会自动从查询结果中排除。

判断给定模型实例是否被软删除，可以使用 `trashed` 方法：

```
if ($flight->trashed()) {
    //
}
```

查询被软删除的模型

包含软删除模型

正如上面提到的，软删除模型将会自动从查询结果中排除，但是，如果你想要软删除模型出现在查询结果中，可以使用 `withTrashed` 方法：

```
$flights = App\Flight::withTrashed()
    ->where('account_id', 1)
    ->get();
```

`withTrashed` 方法也可以用于关联查询中：

```
$flight->history()->withTrashed()->get();
```

只获取软删除模型

`onlyTrashed` 方法之获取软删除模型：

```
$flights = App\Flight::onlyTrashed()
```

```
->where('airline_id', 1)  
->get();
```

恢复软删除模型

有时候你希望恢复一个被软删除的模型，可以使用 `restore` 方法：

```
$flight->restore();
```

你还可以在查询中使用 `restore` 方法来快速恢复多个模型：

```
App\Flight::withTrashed()  
    ->where('airline_id', 1)  
    ->restore();
```

和 `withTrashed` 方法一样，`restore` 方法也可以用于关联查询：

```
$flight->history()->restore();
```

永久删除模型

有时候你真的需要从数据库中删除一个模型，可以使用 `forceDelete` 方法：

```
// 强制删除单个模型实例...  
$flight->forceDelete();  
// 强制删除所有关联模型...  
$flight->history()->forceDelete();
```

7、查询作用域

全局作用域

全局作用域允许我们为给定模型的所有查询添加条件约束。 Laravel 自带的软删除功能就使用了全局作用域来从数据库中拉出所有没有被删除的模型。编写自定义的全局作用域可以提供一种方便的、简单的方式来确保给定模型的每个查询都有特定的条件约束。

编写全局作用域

自定义全局作用域很简单，首先定义一个实现 `Illuminate\Database\Eloquent\Scope` 接口的类，

该接口要求你实现一个方法：`apply`。需要的话可以在 `apply` 方法中添加 `where` 条件到查询：

```
<?php

namespace App\Scopes;

use Illuminate\Database\Eloquent\Scope;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Builder;

class AgeScope implements Scope{

    /**
     * Apply the scope to a given Eloquent query builder.
     *
     * @param \Illuminate\Database\Eloquent\Builder $builder
     * @param \Illuminate\Database\Eloquent\Model $model
     * @return void
     */

    public function apply(Builder $builder, Model $model)
    {
        return $builder->where('age', '>', 200);
    }
}
```

注 : Laravel 应用默认并没有为作用域预定义文件夹 , 所以你可以按照自己的喜好在 `app` 目录下创建 `Scopes` 目录。

应用全局作用域

要将全局作用域分配给模型 , 需要重写给定模型的 `boot` 方法并使用 `addGlobalScope` 方法 :

```
<?php

namespace App;

use App\Scopes\AgeScope;
use Illuminate\Database\Eloquent\Model;

class User extends Model{

    /**
     * The "booting" method of the model.
     *
     * @return void
     */
    protected static function boot()
    {

        parent::boot();

        static::addGlobalScope(new AgeScope);
    }
}
```

添加作用域后，如果使用 `User::all()` 查询则会生成如下 SQL 语句：

```
select * from `users` where `age` > 200
```

匿名的全局作用域

Eloquent 还允许我们使用闭包定义全局作用域，这在实现简单作用域的时候特别有用，这样的话，我们就没必要定义一个单独的类了：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Builder;

class User extends Model{

    /**
     * The "booting" method of the model.
     *
     * @return void
     */

    protected static function boot()
    {
        parent::boot();

        static::addGlobalScope('age', function(Builder $builder) {
            $builder->where('age', '>', 200);
        });
    }
}
```

```
});  
}  
}
```

我们还可以通过以下方式移除全局作用：

```
User::withoutGlobalScope('age')->get();
```

移除全局作用域

如果想要在给定查询中移除指定全局作用域，可以使用 `withoutGlobalScope`：

```
User::withoutGlobalScope(AgeScope::class)->get();
```

如果你想要移除某几个或全部全局作用域，可以使用 `withoutGlobalScopes` 方法：

```
User::withoutGlobalScopes()->get();  
User::withoutGlobalScopes([FirstScope::class, SecondScope::class])  
->get();
```

本地作用域

本地作用域允许我们定义通用的约束集合以便在应用中复用。例如，你可能经常需要获取最受欢迎的用户，要定义这样的一个作用域，只需简单在对应 Eloquent 模型方法前加上一个 `scope` 前缀。

作用域总是返回查询构建器实例：

```
<?php  
  
namespace App;  
  
use Illuminate\Database\Eloquent\Model;  
  
class User extends Model{
```

```
/*
 * 只包含活跃用户的查询作用域
 *
 * @return \Illuminate\Database\Eloquent\Builder
 */
public function scopePopular($query)
{
    return $query->where('votes', '>', 100);
}

/*
 * 只包含激活用户的查询作用域
 *
 * @return \Illuminate\Database\Eloquent\Builder
 */
public function scopeActive($query)
{
    return $query->where('active', 1);
}
```

使用本地作用域

作用域被定义好了之后，就可以在查询模型的时候调用作用域方法，但调用时不需要加上 `scope`

前缀，你甚至可以在同时调用多个作用域，例如：

```
$users = App\User::popular()->active()->orderBy('created_at')->get();
```

动态作用域

有时候你可能想要定义一个可以接收参数的作用域，你只需要将额外的参数添加到你的作用域即

可。作用域参数应该被定义在 `$query` 参数之后：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
```

```
class User extends Model{
    /**
     * 只包含给用类型用户的查询作用域
     *
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopeOfType($query, $type)
    {
        return $query->where('type', $type);
    }
}
```

现在，你可以在调用作用域时传递参数了：

```
$users = App\User::ofType('admin')->get();
```

8、事件

Eloquent 模型可以触发事件，允许你在模型生命周期中的多个时间点调用如下这些方法：

`creating, created, updating, updated, saving, saved, deleting, deleted, restoring, restored`。事

件允许你在一个指定模型类每次保存或更新的时候执行代码。

基本使用

一个新模型被首次保存的时候，`creating` 和 `created` 事件会被触发。如果一个模型已经在数据库

中存在并调用 `save` 方法，`updating/updated` 事件会被触发，无论是创建还是更新，`saving/saved`

事件都会被调用。

举个例子，我们在[服务提供者](#)中定义一个 Eloquent 事件监听器，在事件监听器中，我们会调用给

定模型的 `isValid` 方法，如果模型无效会返回 `false`。如果从 Eloquent 事件监听器中返回 `false`

则取消 `save/update` 操作：

```
<?php

namespace App\Providers;

use App\User;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider{

    /**
     * 启动所有应用服务
     *
     * @return void
     */

    public function boot()
    {
        User::creating(function ($user) {
            return $user->isValid();
        });
    }

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

观察者

如果你在给定模型中监听多个事件，可以使用观察者来对所有监听器进行分组，观察者类拥有反射你想要监听的 Eloquent 事件对应的方法名，每个方法接收模型作为唯一参数。 Laravel 并没有为监听器提供默认目录，所以你可以创建任意目录来存放观察者类：

```
<?php

namespace App\Observers;

use App\User;

class UserObserver
{
    /**
     * Listen to the User created event.
     *
     * @param  User  $user
     * @return void
     */
    public function created(User $user)
    {
        //
    }

    /**
     * Listen to the User deleting event.
     */
}
```

```
* @param User $user
* @return void
*/
public function deleting(User $user)
{
    //
}
```

要监听观察者，使用你想要观察模型的 `observe` 方法，你可以在某个服务提供者的 `boot` 方法中注册观察者，在本例中，我们在 `AppServiceProvider` 中注册观察者：

```
<?php

namespace App\Providers;

use App\User;
use App\Observers\UserObserver;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        User::observe(UserObserver::class);
    }

    /**
     * Register the service provider.
     *
```

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

11.2 关联关系

1、简介

数据表经常要与其它表做关联，比如一篇博客文章可能有很多评论，或者一个订单会被关联到下单用户，Eloquent 使得组织和处理这些关联关系变得简单，并且支持多种不同类型的关联关系：

- 一对一
- 一对多
- 多对多
- 远层一对多
- 多态关联
- 多对多的多态关联

2、定义关联关系

Eloquent 关联关系以 Eloquent 模型类方法的形式被定义。和 Eloquent 模型本身一样，关联关系也是强大的查询构建器，定义关联关系为函数能够提供功能强大的方法链和查询能力。例如：

```
$user->posts()->where('active', 1)->get();
```

但是，在深入使用关联关系之前，让我们先学习如何定义每种关联类型：

一对一

一对一关联是一个非常简单的关联关系，例如，一个 `User` 模型有一个与之对应的 `Phone` 模型。要定义这种模型，我们需要将 `phone` 方法置于 `User` 模型中，`phone` 方法会调用 Eloquent 模型基类上 `hasOne` 方法并返回其结果：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 获取关联到用户的手机
     */
    public function phone()
    {
        return $this->hasOne('App\Phone');
    }
}
```

传递给 `hasOne` 方法的第一个参数是关联模型的名称，关联关系被定义后，我们可以使用 Eloquent 的[动态属性](#)获取关联记录。动态属性允许我们访问关联函数就像它们是定义在模型上的属性一样：

```
$phone = User::find(1)->phone;
```

Eloquent 默认关联关系的外键基于模型名称，在本例中，`Phone` 模型默认有一个 `user_id` 外键，如果你希望重写这种约定，可以传递第二个参数到 `hasOne` 方法：

```
return $this->hasOne('App\Phone', 'foreign_key');
```

此外，Eloquent 假设外键应该在父级上有一个与之匹配的 `id`，换句话说，Eloquent 将会通过 `user` 表的 `id` 值去 `phone` 表中查询 `user_id` 与之匹配的 `Phone` 记录。如果你想要关联关系使用其他值

而不是 `id`，可以传递第三个参数到 `hasOne` 来指定自定义的主键：

```
return $this->hasOne('App\Phone', 'foreign_key', 'local_key');
```

定义相对的关联

我们可以从 `User` 中访问 `Phone` 模型，相应的，我们也可以在 `Phone` 模型中定义关联关系从而让我们可以拥有该 `phone` 的 `User`。我们可以使用 `belongsTo` 方法定义与 `hasOne` 关联关系相对的关联：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Phone extends Model{
    /**
     * 获取手机对应的用户
     */
    public function user()
    {
        return $this->belongsTo('App\User');
    }
}
```

在上面的例子中，Eloquent 将会尝试通过 `Phone` 模型的 `user_id` 去 `User` 模型查找与之匹配的记录。Eloquent 通过关联关系方法名并在方法名后加 `_id` 后缀来生成默认的外键名。然而，如果 `Phone` 模型上的外键不是 `user_id`，也可以将自定义的键名作为第二个参数传递到 `belongsTo` 方法：

```
/**
 * 获取手机对应的用户
 */
public function user(){
    return $this->belongsTo('App\User', 'foreign_key');
```

```
}
```

如果父模型不使用 `id` 作为主键，或者你希望使用别的列来连接子模型，可以将父表自定义键作为第三个参数传递给 `belongsTo` 方法：

```
/**  
 * 获取手机对应的用户  
 */  
public function user(){  
    return $this->belongsTo('App\User', 'foreign_key', 'other_key');  
}
```

一对多

“一对多”是用于定义单个模型拥有多个其它模型的关联关系。例如，一篇博客文章拥有无数评论，和其他关联关系一样，一对多关联通过在 Eloquent 模型中定义方法来定义：

```
<?php  
  
namespace App;  
  
use Illuminate\Database\Eloquent\Model;  
  
class Post extends Model{  
    /**  
     * 获取博客文章的评论  
     */  
    public function comments()  
    {
```

```
    return $this->hasMany('App\Comment');

}

}
```

记住，Eloquent 会自动判断 `Comment` 模型的外键，为方便起见，Eloquent 将拥有者模型名称加上 `_id` 后缀作为外键。因此，在本例中，Eloquent 假设 `Comment` 模型上的外键是 `post_id`。

关联关系被定义后，我们就可以通过访问 `comments` 属性来访问评论集合。记住，由于 Eloquent 提供“动态属性”，我们可以像访问模型的属性一样访问关联方法：

```
$comments = App\Post::find(1)->comments;

foreach ($comments as $comment) {
    //
}
```

当然，由于所有关联同时也是 [查询构建器](#)，我们可以添加更多的条件约束到通过调用 `comments` 方法获取到的评论上：

```
$comments = App\Post::find(1)->comments()->where('title', 'foo')->
first();
```

和 `hasOne` 方法一样，你还可以通过传递额外参数到 `hasMany` 方法来重新设置外键和本地主键：

```
return $this->hasMany('App\Comment', 'foreign_key');

return $this->hasMany('App\Comment', 'foreign_key', 'local_key');
```

一对多（逆向）

现在我们可以访问文章的所有评论了，接下来让我们定义一个关联关系允许通过评论访问所属文章。要定义与 `hasMany` 相对的关联关系，需要在子模型中定义一个关联方法去调用 `belongsTo` 方

法：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model{
    /**
     * 获取评论对应的博客文章
     */
    public function post()
    {
        return $this->belongsTo('App\Post');
    }
}
```

关联关系定义好之后，我们可以通过访问动态属性 `post` 来获取一条 `Comment` 对应的 `Post`：

```
$comment = App\Comment::find(1);
echo $comment->post->title;
```

在上面这个例子中，Eloquent 尝试匹配 `Comment` 模型的 `post_id` 与 `Post` 模型的 `id`，Eloquent 通过关联方法名加上 `_id` 后缀生成默认外键，当然，你也可以通过传递自定义外键名作为第二个参数传递到 `belongsTo` 方法，如果你的外键不是 `post_id`，或者你想自定义的话：

```
/**
 * 获取评论对应的博客文章
 */
public function post(){
    return $this->belongsTo('App\Post', 'foreign_key');
}
```

如果你的父模型不使用 `id` 作为主键，或者你希望通过其他列来连接子模型，可以将自定义键名作为第三个参数传递给 `belongsTo` 方法：

```
/**
```

```
* 获取评论对应的博客文章
*/
public function post(){
    return $this->belongsTo('App\Post', 'foreign_key', 'other_key');
}
```

多对多

多对多关系比 `hasOne` 和 `hasMany` 关联关系要稍微复杂一些。这种关联关系的一个例子就是一个用户有多个角色，同时一个角色被多个用户共用。例如，很多用户可能都有一个“Admin”角色。要定义这样的关联关系，需要三个数据表：`users`、`roles` 和 `role_user`，`role_user` 表按照关联模型名的字母顺序命名，并且包含 `user_id` 和 `role_id` 两个列。

多对多关联通过编写一个调用 Eloquent 基类上的 `belongsToMany` 方法的函数来定义，例如，我们在 `User` 模型上定义 `roles` 方法：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 用户角色
     */
    public function roles()
    {
        return $this->belongsToMany('App\Role');
    }
}
```

关联关系被定义之后，可以使用动态属性 `roles` 来访问用户的角色：

```
$user = App\User::find(1);
```

```
foreach ($user->roles as $role) {  
    //  
}
```

当然，和所有其它关联关系类型一样，你可以调用 `roles` 方法来添加条件约束到关联查询上：

```
$roles = App\User::find(1)->roles()->orderBy('name')->get();
```

正如前面所提到的，为了决定关联关系连接表的表名，Eloquent 以字母顺序连接两个关联模型的名字。然而，你可以重写这种约定——通过传递第二个参数到 `belongsToMany` 方法：

```
return $this->belongsToMany('App\Role', 'user_roles');
```

除了自定义连接表的表名，你还可以通过传递额外参数到 `belongsToMany` 方法来自定义该表中字段的列名。第三个参数是你定义的关系模型的外键名称，第四个参数你要连接到的模型的外键名称：

```
return $this->belongsToMany('App\Role', 'user_roles', 'user_id', 'role_id');
```

定义相对的关联关系

要定义与多对多关联相对的关联关系，只需在关联模型中调用一下 `belongsToMany` 方法即可。

让我们在 `Role` 模型中定义 `users` 方法：

```
<?php  
  
namespace App;  
  
use Illuminate\Database\Eloquent\Model;  
  
class Role extends Model{  
    /**  
     * 角色用户  
     */  
    public function users()
```

```
{  
    return $this->belongsToMany('App\User');  
}  
}
```

正如你所看到的，定义的关联关系和与其对应的 `User` 中定义的一模一样，只是前者引用 `App\Role`，后者引用 `App\User`，由于我们再次使用了 `belongsToMany` 方法，所有的常用表和键自定义选项在定义与多对多相对的关联关系时都是可用的。

获取中间表字段

正如你已经学习到的，处理多对多关联要求一个中间表。Eloquent 提供了一些有用的方法来与其进行交互，例如，我们假设 `User` 对象有很多与之关联的 `Role` 对象，访问这些关联关系之后，我们可以使用模型上的 `pivot` 属性访问中间表：

```
$user = App\User::find(1);  
  
foreach ($user->roles as $role) {  
    echo $role->pivot->created_at;  
}
```

注意我们获取到的每一个 `Role` 模型都被自动赋上了 `pivot` 属性。该属性包含一个代表中间表的模型，并且可以像其它 Eloquent 模型一样使用。

默认情况下，只有模型键才能用在 `pivot` 对象上，如果你的 `pivot` 表包含额外的属性，必须在定义关联关系时进行指定：

```
return $this->belongsToMany('App\Role')->withPivot('column1', 'column2');
```

如果你想要你的 `pivot` 表自动包含 `created_at` 和 `updated_at` 时间戳，在关联关系定义时使用 `withTimestamps` 方法：

```
return $this->belongsToMany('App\Role')->withTimestamps();
```

通过中间表字段过滤关联关系

你还可以在定义关联关系的时候使用 `wherePivot` 和 `wherePivotIn` 方法过滤 `belongsToMany` 返回的结果集：

```
return $this->belongsToMany('App\Role')->wherePivot('approved', 1);  
  
return $this->belongsToMany('App\Role')->wherePivotIn('priority', [1, 2]);
```

远层的一对多

“远层一对多”关联为通过中间关联访问远层的关联关系提供了一个便利之道。例如，`Country` 模型通过中间的 `User` 模型可能拥有多个 `Post` 模型。在这个例子中，你可以轻易的聚合给定国家的所有文章，让我们看看定义这个关联关系需要哪些表：

```
countries  
    id - integer  
    name - string  
  
users  
    id - integer  
    country_id - integer  
    name - string  
  
posts  
    id - integer  
    user_id - integer  
    title - string
```

尽管 `posts` 表不包含 `country_id` 列，`hasManyThrough` 关联提供了通过 `$country->posts` 来访问一

个国家的所有文章。要执行该查询，Eloquent 在中间表`$users` 上检查 `country_id`，查找到相匹配的用户 ID 后，通过用户 ID 来查询 `posts` 表。

既然我们已经查看了该关联关系的数据表结构，接下来让我们在 `Country` 模型上进行定义：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Country extends Model{
    /**
     * 获取指定国家的所有文章
     */
    public function posts()
    {
        return $this->hasManyThrough('App\Post', 'App\User');
    }
}
```

第一个传递到 `hasManyThrough` 方法的参数是最终我们希望访问的模型的名称，第二个参数是中间模型名称。

当执行这种关联查询时通常 Eloquent 外键规则会被使用，如果你想要自定义该关联关系的外键，可以将它们作为第三个、第四个参数传递给 `hasManyThrough` 方法。第三个参数是中间模型的外键名，第四个参数是最终模型的外键名。

```
class Country extends Model{
    public function posts()
    {
        return $this->hasManyThrough('App\Post', 'App\User', 'country_id', 'user_id');
    }
}
```

多态关联

表结构

多态关联允许一个模型在单个关联下属于多个不同模型。例如，假设应用用户既可以对文章进行评论也可以对视频进行评论，使用多态关联，你可以在这两种场景下使用单个 `comments` 表，首先，让我们看看构建这种关联关系需要的表结构：

```
posts
id - integer
title - string
body - text

videos
id - integer
title - string
url - string

comments
id - integer
body - text
commentable_id - integer
commentable_type - string
```

两个重要的需要注意的列是 `comments` 表上的 `commentable_id` 和 `commentable_type`。

`commentable_id` 列对应 `Post` 或 `Video` 的 ID 值，而 `commentable_type` 列对应所属模型的类名。

当访问 `commentable` 关联时，ORM 根据 `commentable_type` 字段来判断所属模型的类型并返回相

应模型实例。

模型结构

接下来，让我们看看构建这种关联关系需要在模型中定义什么：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
     * Get all of the owning commentable models.
     */
    public function commentable()
    {
        return $this->morphTo();
    }
}

class Post extends Model
{
    /**
     * Get all of the post's comments.
     */
}
```

```
public function comments()
{
    return $this->morphMany('App\Comment', 'commentable');
}

class Video extends Model
{
    /**
     * Get all of the video's comments.
     */
    public function comments()
    {
        return $this->morphMany('App\Comment', 'commentable');
    }
}
```

获取多态关联

数据表和模型定义好以后，可以通过模型访问关联关系。例如，要访问一篇文章的所有评论，可以使用以下代码：

```
$post = App\Post::find(1);

foreach ($post->comments as $comment) {
    //
}
```

你还可以通过调用 `morphTo` 方法从多态模型中获取多态关联的所属对象。在本例中，就是 `Comment` 模型中的 `commentable` 方法。因此，我们可以用动态属性的方式访问该方法：

```
$comment = App\Comment::find(1);

$commentable = $comment->commentable;
```

`Comment` 模型的 `commentable` 关联返回 `Post` 或 `Video` 实例，这取决于哪个类型的模型拥有该评论。

自定义多态类型

默认情况下，[Laravel](#) 使用完全限定类名来存储关联模型的类型。举个例子，上面示例中的 `Comment` 可能属于某个 `Post` 或 `Video`，默认的 `commentable_type` 可能是 `App\Post` 或 `App\Video`。不过，有时候你可能需要解除数据库和应用内部结构之间的耦合，这样的情况下，可以定义一个 `morphMap` 关联来告知 Eloquent 为每个模型使用自定义名称替代完整类名：

```
use Illuminate\Database\Eloquent\Relations\Relation;

Relation::morphMap([
    'posts' => App\Post::class,
    'videos' => App\Video::class,
]);
```

你可以在 `AppServiceProvider` 的 `boot` 方法中注册这个 `morphMap`，如果需要的话，也可以创建一个独立的服务提供者来实现这一功能。

多对多的多态关联

表结构

除了传统的多态关联，还可以定义“多对多”的多态关联，例如，一个博客的 Post 和 Video 模型可能共享一个 Tag 模型的多态关联。使用对多对的多态关联允许你在博客文章和视频之间有唯一的标签列表。首先，让我们看看表结构：

```
posts
    id - integer
    name - string

videos
    id - integer
    name - string

tags
    id - integer
    name - string

taggables
    tag_id - integer
    taggable_id - integer
    taggable_type - string
```

模型结构

接下来，我们准备在模型中定义该关联关系。Post 和 Video 模型都有一个 tags 方法调用 Eloquent 基类的 morphToMany 方法：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model{
```

```
/**
 * 获取指定文章所有标签
 */
public function tags()
{
    return $this->morphToMany('App\Tag', 'taggable');
}
```

定义相对的关联关系

接下来，在 Tag 模型中，应该为每一个关联模型定义一个方法，例如，我们定义一个 posts 方法和 videos 方法：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Tag extends Model{
    /**
     * 获取所有分配该标签的文章
     */
    public function posts()
    {
        return $this->morphedByMany('App\Post', 'taggable');
    }

    /**
     * 获取分配该标签的所有视频
     */
    public function videos()
    {
        return $this->morphedByMany('App\Video', 'taggable');
    }
}
```

获取关联关系

定义好数据库和模型后可以通过模型访问关联关系。例如，要访问一篇文章的所有标签，可以使

用动态属性 `tags`：

```
$post = App\Post::find(1);

foreach ($post->tags as $tag) {
    //
}
```

还可以通过访问调用 `morphedByMany` 的方法名从多态模型中获取多态关联的所属对象。在本例中，

就是 Tag 模型中的 `posts` 或者 `videos` 方法：

```
$tag = App\Tag::find(1);

foreach ($tag->videos as $video) {
    //
}
```

3、关联查询

由于 Eloquent 所有关联关系都是通过函数定义，你可以调用这些方法来获取关联关系的实例而不需要再去手动执行关联查询。此外，所有 Eloquent 关联关系类型同时也是查询构建器，允许你在最终在数据库执行 SQL 之前继续添加条件约束到关联查询上。

例如，假定在一个博客系统中一个 User 模型有很多相关的 Post 模型：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     *
     */
}
```

```
* 获取指定用户的所有文章
*/
public function posts()
{
    return $this->hasMany('App\Post');
}
```

你可以像这样查询 `posts` 关联并添加额外的条件约束到该关联关系上：

```
$user = App\User::find(1);
$user->posts()->where('active', 1)->get();
```

你可以在关联关系上使用任何[查询构建器](#)！

关联方法 VS 动态属性

如果你不需要添加额外的条件约束到 Eloquent 关联查询，你可以简单通过动态属性来访问关联

对象，例如，还是拿 `User` 和 `Post` 模型作为例子，你可以像这样访问用户的所有文章：

```
$user = App\User::find(1);

foreach ($user->posts as $post) {
    //
}
```

动态属性就是“[懒惰式加载](#)”，意味着当你真正访问它们的时候才会加载关联数据。正因为如此，开发者经常使用[渴求式加载](#)来预加载他们知道在加载模型时要被访问的关联关系。渴求式加载有效减少了必须要被执行以加载模型关联的 SQL 查询。

查询已存在的关联关系

访问一个模型的记录的时候，你可能希望基于关联关系是否存在来限制查询结果的数目。例如，

假设你想要获取所有至少有一个评论的博客文章，要实现这个，可以传递关联关系的名称到 `has`

方法：

```
// 获取所有至少有一条评论的文章...
$posts = App\Post::has('comments')->get();
```

你还可以指定操作符和大小来自定义查询：

```
// 获取所有至少有三条评论的文章...
$posts = Post::has('comments', '>=', 3)->get();
```

还可以使用“.”来构造嵌套 has 语句，例如，你要获取所有至少有一条评论及投票的所有文章：

```
// 获取所有至少有一条评论获得投票的文章...
$posts = Post::has('comments.votes')->get();
```

如果你需要更强大的功能 可以使用 `whereHas` 和 `orWhereHas` 方法将 `where` 条件放到 `has` 查询上，

这些方法允许你添加自定义条件约束到关联关系条件约束，例如检查一条评论的内容：

```
// 获取所有至少有一条评论包含 foo 字样的文章
$posts = Post::whereHas('comments', function ($query) {
    $query->where('content', 'like', 'foo%');
})->get();
```

无关联结果查询

访问一个模型的记录时，你可能需要基于缺失关联结果的模型对查询结果进行限定。例如，假设

你想要获取所有没有评论的博客文章，可以传递关联关系名称到 `doesntHave` 方法来实现：

```
$posts = App\Post::doesntHave('comments')->get();
```

如果你需要更多功能，可以使用 `whereDoesntHave` 方法添加更多“where”条件到 `doesntHave` 查询，

该方法允许你添加自定义约束条件到关联关系约束，例如检查评论内容：

```
$posts = Post::whereDoesntHave('comments', function ($query) {
    $query->where('content', 'like', 'foo%');
})->get();
```

统计关联模型

如果你想要在不加载关联关系的情况下统计关联结果数目，可以使用 `withCount` 方法，该方法会放置一个`{relation}_count` 字段到结果模型。例如：

```
$posts = App\Post::withCount('comments')->get();

foreach ($posts as $post) {
    echo $post->comments_count;
}
```

你可以像添加约束条件到查询一样来添加多个关联关系的“计数”：

```
$posts = Post::withCount(['votes', 'comments' => function ($query) {
    $query->where('content', 'like', 'foo%');
}])->get();

echo $posts[0]->votes_count;
echo $posts[0]->comments_count;
```

4、渴求式加载

当以属性方式访问数据库关联关系的时候，关联关系数据时“懒惰式加载”的，这意味着关联关系数据直到第一次访问的时候才被加载。然而，Eloquent 可以在查询父级模型的同时“渴求式加载”关联关系。渴求式加载缓解了 N+1 查询问题，要阐明 N+1 查询问题，考虑下关联到 Author 的 Book 模型：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Book extends Model{
    /**
     * 获取写这本书的作者
     */
    public function author()
    {
        return $this->belongsTo('App\Author');
    }
}
```

现在，让我们获取所有书及其作者：

```
$books = App\Book::all();

foreach ($books as $book) {
    echo $book->author->name;
}
```

该循环先执行 1 次查询获取表中的所有书，然后另一个查询获取每一本书的作者，因此，如果有 25 本书，要执行 26 次查询：1 次是获取书本身，剩下的 25 次查询是为每一本书获取其作者。

谢天谢地，我们可以使用渴求式加载来减少该操作到 2 次查询。当查询的时候，可以使用 with 方

法指定应该被渴求式加载的关联关系：

```
$books = App\Book::with('author')->get();  
  
foreach ($books as $book) {  
    echo $book->author->name;  
}
```

在该操作中，只执行两次查询即可：

```
select * from books  
select * from authors where id in (1, 2, 3, 4, 5, ...)
```

渴求式加载多个关联关系

有时候你需要在单个操作中渴求式加载几个不同的关联关系。要实现这个，只需要添加额外的参

数到 `with` 方法即可：

```
$books = App\Book::with('author', 'publisher')->get();
```

嵌套的渴求式加载

要渴求式加载嵌套的关联关系，可以使用“.”语法。例如，让我们在一个 Eloquent 语句中渴求式加载所有书的作者及所有作者的个人联系方式：

```
$books = App\Book::with('author.contacts')->get();
```

带条件约束的渴求式加载

有时候我们希望渴求式加载一个关联关系，但还想为渴求式加载指定更多的查询条件：

```
$users = App\User::with(['posts' => function ($query) {  
    $query->where('title', 'like', '%first%');  
}])->get();
```

在这个例子中，Eloquent 只渴求式加载 title 包含 first 的文章。当然，你可以调用其它查询构

建器来自定义渴求式加载操作：

```
$users = App\User::with(['posts' => function ($query) {
    $query->orderBy('created_at', 'desc');
}])->get();
```

懒惰渴求式加载

有时候你需要在父模型已经被获取后渴求式加载一个关联关系。例如，这在你需要动态决定是否加载关联模型时可能很有用：

```
$books = App\Book::all();

if ($someCondition) {
    $books->load('author', 'publisher');
}
```

如果你需要设置更多的查询条件到渴求式加载查询上，可以传递一个闭包到 `load` 方法：

```
$books->load(['author' => function ($query) {
    $query->orderBy('published_date', 'asc');
}]);
```

5、插入&更新关联模型

save 方法

Eloquent 提供了便利的方法来添加新模型到关联关系。例如，也许你需要插入新的 `Comment` 到 `Post` 模型，你可以从关联关系的 `save` 方法直接插入 `Comment` 而不是手动设置 `Comment` 的 `post_id`

属性：

```
$comment = new App\Comment(['message' => 'A new comment.']);
$post = App\Post::find(1);
```

```
$post->comments()->save($comment);
```

注意我们没有用动态属性方式访问 `comments`，而是调用 `comments` 方法获取关联关系实例。`save` 方法会自动添加 `post_id` 值到新的 `Comment` 模型。

如果你需要保存多个关联模型，可以使用 `saveMany` 方法：

```
$post = App\Post::find(1);

$post->comments()->saveMany([
    new App\Comment(['message' => 'A new comment.']),
    new App\Comment(['message' => 'Another comment.']),
]);
```

create 方法

除了 `save` 和 `saveMany` 方法外，还可以使用 `create` 方法，该方法接收属性数组、创建模型、然后插入数据库。`save` 和 `create` 的不同之处在于 `save` 接收整个 Eloquent 模型实例而 `create` 接收原生 PHP 数组：

```
$post = App\Post::find(1);

$comment = $post->comments()->create([
    'message' => 'A new comment.',
]);
```

使用 `create` 方法之前确保先浏览属性[批量赋值文档](#)。

从属关联关系

更新 `belongsTo` 关联的时候，可以使用 `associate` 方法，该方法会在子模型设置外键：

```
$account = App\Account::find(10);
$user->account()->associate($account);
$user->save();
```

移除 `belongsTo` 关联的时候，可以使用 `dissociate` 方法。该方法在子模型上取消外键和关联：

```
$user->account()->dissociate();
$user->save();
```

多对多关联

附加/分离

处理多对多关联的时候，Eloquent 提供了一些额外的帮助函数来使得处理关联模型变得更加方便。

例如，让我们假定一个用户可能有多个角色同时一个角色属于多个用户，要通过在连接模型的中

间表中插入记录附加角色到用户上，可以使用 `attach` 方法：

```
$user = App\User::find(1);
$user->roles()->attach($roleId);
```

附加关联关系到模型，还可以以数组形式传递额外被插入数据到中间表：

```
$user->roles()->attach($roleId, ['expires' => $expires]);
```

当然，有时候有必要从用户中移除角色，要移除一个多对多关联记录，使用 `detach` 方法。`detach`

方法将会从中间表中移除相应的记录；然而，两个模型在数据库中都保持不变：

```
// 从指定用户中移除角色...
$user->roles()->detach($roleId);
// 从指定用户移除所有角色...
$user->roles()->detach();
```

为了方便，`attach` 和 `detach` 还接收数组形式的 ID 作为输入：

```
$user = App\User::find(1);
$user->roles()->detach([1, 2, 3]);
```

```
$user->roles()->attach([1 => ['expires' => $expires], 2, 3]);
```

同步

你还可以使用 `sync` 方法构建多对多关联。`sync` 方法接收数组形式的 ID 并将其放置到中间表。任何不在该数组中的 ID 对应记录将会从中间表中移除。因此，该操作完成后，只有在数组中的 ID 对应记录还存在于中间表：

```
$user->roles()->sync([1, 2, 3]);
```

你还可以和 ID 一起传递额外的中间表值：

```
$user->roles()->sync([1 => ['expires' => true], 2, 3]);
```

如果你不想要脱离存在的 ID，可以使用 `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中已存在的行，可以使用 `updateExistingPivot` 方法。该方法接收中间记录外键和属性数组进行更新：

```
$user = App\User::find(1);

$user->roles()->updateExistingPivot($roleId, $attributes);
```

6、触发父级时间戳

当一个模型属于另外一个时，例如 `Comment` 属于 `Post`，子模型更新时父模型的时间戳也被更新将很有用，例如，当 `Comment` 模型被更新时，你可能想要“触发”创建其所属模型 `Post` 的 `updated_at` 时间戳。Eloquent 使得这项操作变得简单，只需要添加包含关联关系名称的 `touches` 属性到子模型中即可：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model{
    /**
     * 要触发的所有关联关系
     *
     * @var array
     */
    protected $touches = ['post'];

    /**
     * 评论所属文章
     */
    public function post()
    {
        return $this->belongsTo('App\Post');
    }
}
```

现在，当你更新 `Comment` 时，所属模型 `Post` 将也会更新其 `updated_at` 值，从而方便得知何时更

新 Post 模型缓存：

```
$comment = App\Comment::find(1);
$comment->text = 'Edit to this comment!';
$comment->save();
```

11.3 集合

1、简介

Eloquent 返回的所有的包含多条记录的结果集都是 Illuminate\Database\Eloquent\Collection 对象的实例，包括通过 get 方法或者通过访问关联关系获取的结果。Eloquent 集合对象继承自 Laravel 的集合基类，因此很自然的继承了很多处理 Eloquent 模型底层数组的方法。

当然，所有集合也是迭代器，允许你像 PHP 数组一样对其进行循环：

```
$users = App\User::where('active', 1)->get();

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

不过，集合使用直观的接口提供了各种映射/简化操作，因此比数组更加强大。例如，我们可以通过以下方式移除所有无效的模型并聚合还存在的用户的姓名：

```
$users = App\User::where('active', 1)->get();

$names = $users->reject(function ($user) {
```

```
return $user->active === false;  
})->map(function ($user) {  
    return $user->name;  
});
```

注：尽管大多数 Eloquent 集合返回的是一个新的 Eloquent 集合实例，但是 `pluck`、`keys`、`zip`、`collapse`、`flatten` 和 `flip` 方法返回的是集合基类实例。类似地，如果 `map` 操作返回的集合不包含任何 Eloquent 模型，将会被自动转化成集合基类。

2、可用方法

集合基类

所有的 Eloquent 集合继承自 Laravel 集合对象基类，因此，它们继承所有集合基类提供的强大方法：[集合方法大全](#)。

3、自定义集合

如果你需要在自己扩展的方法中使用自定义的集合对象，可以重写模型上的 `newCollection` 方法：

```
<?php  
  
namespace App;  
  
use App\CustomCollection;  
use Illuminate\Database\Eloquent\Model;  
  
class User extends Model{  
    /**  
     * 创建一个新的 Eloquent 集合实例  
     *  
     * @param array $models
```

```
* @return \Illuminate\Database\Eloquent\Collection
*/
public function newCollection(array $models = [])
{
    return new CustomCollection($models);
}
```

定义好 `newCollection` 方法后，无论何时 Eloquent 返回该模型的 `Collection` 实例你都会获取到自定义的集合。如果你想要在应用中的每一个模型中使用自定义集合，需要在模型基类中重写 `newCollection` 方法。

11.4 访问器&修改器

1、简介

访问器和修改器允许你在获取模型属性或设置其值时格式化 Eloquent 属性。例如，你可能想要使用 [Laravel 加密器](#)对存储在数据库中的数据进行加密，并且在 Eloquent 模型中访问时自动进行解密。

除了自定义访问器和修改器，Eloquent 还可以自动转换日期字段为 Carbon 实例甚至将文本转换为 JSON。

2、访问器 & 修改器

定义访问器

要定义一个访问器，需要在模型中创建一个 `getFooAttribute` 方法，其中 `Foo` 是你想要访问的字段名（使用驼峰式命名规则）。在本例中，我们将会为 `first_name` 属性定义一个访问器，该访问

器在获取 `first_name` 的值时被 Eloquent 自动调用：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 获取用户的名字
     *
     * @param string $value
     * @return string
     */
    public function getFirstNameAttribute($value)
    {
        return ucfirst($value);
    }
}
```

正如你所看到的，该字段的原生值被传递给访问器，然后返回处理过的值。要访问该值只需要简

单访问 `first_name` 即可：

```
$user = App\User::find(1);
$firstName = $user->first_name;
```

定义修改器

要定义一个修改器，需要在模型中定义 `setFooAttribute` 方法，其中 `Foo` 是你想要访问的字段（使

用驼峰式命名规则）。接下来让我们为 `first_name` 属性定义一个修改器，当我们为模型上的

`first_name` 赋值时该修改器会被自动调用：

```
<?php

namespace App;
```

```
use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 设置用户的名字
     *
     * @param string $value
     * @return string
     */
    public function setFirstNameAttribute($value)
    {
        $this->attributes['first_name'] = strtolower($value);
    }
}
```

该修改器获取要被设置的属性值，允许你操纵该值并设置 Eloquent 模型内部属性值为操作后的值。例如，如果你尝试设置 `Sally` 的 `first_name` 属性：

```
$user = App\User::find(1);
$user->first_name = 'Sally';
```

在本例中，`setFirstNameAttribute` 方法会被调用，传入参数为 `Sally`，修改器会对其调用 `strtolower` 函数并将处理后的值设置为内部属性的值。

3、日期修改器

默认情况下，Eloquent 将会转化 `created_at` 和 `updated_at` 列的值为 Carbon 实例，该类继承自 PHP 原生的 `Datetime` 类，并提供了各种有用的方法。

你可以自定义哪些字段被自动调整修改，甚至可以通过重写模型中的 `$dates` 属性完全禁止调整：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
```

```
class User extends Model{
    /**
     * 应该被调整为日期的属性
     *
     * @var array
     */
    protected $dates = ['created_at', 'updated_at', 'disabled_at'];
}
```

如果字段是日期格式时，你可以将其值设置为 UNIX 时间戳，日期字符串（`Y-m-d`），日期-时间字符串，`Datetime/Carbon` 实例，日期的值将会自动以正确格式存储到数据库中：

```
$user = App\User::find(1);
$user->disabled_at = Carbon::now();
$user->save();
```

正如上面提到的，当获取被罗列在`$dates` 数组中的属性时，它们会被自动转化为 `Carbon` 实例，允许你在属性上使用任何 `Carbon` 的方法：

```
$user = App\User::find(1);
return $user->disabled_at->getTimestamp();
```

默认情况下，时间戳的格式是“`Y-m-d H:i:s`”，如果你需要自定义时间戳格式，在模型中设置`$dateFormat` 属性，该属性决定日期属性存储在数据库以及序列化为数组或 JSON 时的格式：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model{
    /**
     * 模型日期的存储格式
     *
     * @var string
     */
}
```

```
protected $dateFormat = 'U';
}
```

4、属性转换

模型中的 `$casts` 属性为属性字段转换到通用数据类型提供了便利方法。`$casts` 属性是数组格式，其键是要被转换的属性名称，其值时你想要转换的类型。目前支持的转换类型包括：

`integer`, `real`, `float`, `double`, `string`, `boolean`, `object`, `array`, `collection`, `date` 和 `datetime`。

例如，让我们转换 `is_admin` 属性，将其由 `integer` 类型（0 或 1）转换为 `boolean` 类型：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 应该被转化为原生类型的属性
     *
     * @var array
     */
    protected $casts = [
        'is_admin' => 'boolean',
    ];
}
```

现在 `is_admin` 属性在被访问时总是被转换为 `boolean`，即使底层存储在数据库中的值是 `integer`：

```
$user = App\User::find(1);

if ($user->is_admin) {
    //
}
```

数组&JSON 转换

`array` 类型转换在处理被存储为序列化 JSON 格式的字段时特别有用，例如，如果数据库有一个 `JSON` 或 `TEXT` 字段类型包含了序列化 JSON，添加 `array` 类型转换到该属性将会在 Eloquent 模型中访问其值时自动将其反序列化为 PHP 数组：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 应该被转化为原生类型的属性
     *
     * @var array
     */
    protected $casts = [
        'options' => 'array',
    ];
}
```

类型转换被定义后，访问 `options` 属性将会自动从 JSON 反序列化为 PHP 数组，反之，当你设置 `options` 属性的值时，给定数组将会自动转化为 JSON 以供存储：

```
$user = App\User::find(1);
$options = $user->options;
$options['key'] = 'value';
$user->options = $options;
$user->save();
```

11.5 序列化

1、简介

当构建 JSON API 时，经常需要转化模型和关联关系为数组或 JSON。Eloquent 提供了便捷方法以便实现这些转换，以及控制哪些属性被包含到序列化中。

2、序列化模型&集合

序列化为数组

要转化模型及其加载的关联关系为数组，可以使用 `toArray` 方法。这个方法是递归的，所以所有属性及其关联对象属性（包括关联的关联）都会被转化为数组：

```
$user = App\User::with('roles')->first();
return $user->toArray();
```

还可以转化整个模型集合为数组：

```
$users = App\User::all();
return $users->toArray();
```

序列化为 JSON

要转化模型为 JSON，可以使用 `toJson` 方法，和 `toArray` 一样，`toJson` 方法也是递归的，所有属性及其关联属性都会被转化为 JSON：

```
$user = App\User::find(1);
return $user->toJson();
```

你还可以转化模型或集合为字符串，这将会自动调用 `toJson` 方法：

```
$user = App\User::find(1);
return (string) $user;
```

由于模型和集合在转化为字符串的时候会被转化为 JSON，你可以从应用的路由或控制器中直接

返回 Eloquent 对象：

```
Route::get('users', function(){
    return App\User::all();
});
```

3、在 JSON 中隐藏属性

有时候你希望在模型数组或 JSON 显示中隐藏某些属性，比如密码，要实现这个，在定义模型的时候设置 `$hidden` 属性：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 在数组中隐藏的属性
     *
     * @var array
     */
    protected $hidden = ['password'];
}
```

注意：如果要隐藏关联关系，使用关联关系的方法名，而不是动态属性名。

此外，可以使用 `visible` 属性来定义模型数组和 JSON 显示的属性白名单：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
```

```
class User extends Model{
    /**
     * 在数组中显示的属性
     *
     * @var array
     */
    protected $visible = ['first_name', 'last_name'];
}
```

临时暴露隐藏属性

如果你想要在特定模型中临时显示隐藏的属性，可以使用 `makeVisible` 方法，该方法以方法链的方式返回模型实例：

```
return $user->makeVisible('attribute')->toArray();
```

类似的，如果你想要隐藏给定模型实例上某些显示的属性，可以使用 `makeHidden` 方法：

```
return $user->makeHidden('attribute')->toArray();
```

4、追加值到 JSON

有时候，需要添加数据库中没有的字段到数组中，要实现这个功能，首先要为这个值定义一个访问器：

```
<?php

namespace App;
```

```
use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 为用户获取管理员标识
     *
     * @return bool
     */
    public function getIsAdminAttribute()
    {
        return $this->attributes['admin'] == 'yes';
    }
}
```

定义好访问器后，添加字段名到该模型的 `appends` 属性。需要注意的是，尽快访问器使用“camel case”形式定义，属性名通常以“snake case”的方式被引用：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    /**
     * 追加到模型数组表单的访问器
     *
     * @var array
     */
    protected $appends = ['is_admin'];
}
```

字段被添加到 `appends` 列表之后，将会被包含到模型数组和 JSON 中，`appends` 数组中的属性还

会遵循模型中配置的 `visible` 和 `hidden` 设置。

12. Artisan Console

12.1 控制台命令

1、简介

Artisan 是 Laravel 自带的命令行接口名称，它为我们在开发过程中提供了很多有用的命令。想要查看所有可用的 Artisan 命令，可使用 `list` 命令：

```
php artisan list
```

每个命令都可以用 `help` 指令显示命令描述及命令参数和选项。想要查看帮助界面，只需要在命令前加上 `help` 就可以了：

```
php artisan help migrate
```

2、编写命令

除了 Artisan 提供的系统命令之外，还可以构建自己的命令。你可以将自定义命令存放在 `app/Console/Commands` 目录；当然，你可以自己选择存放位置，只要该命令可以被 Composer 自动加载即可。

生成命令

要创建一个新命令，你可以使用 Artisan 命令 `make:console`，该命令会在 `app/Console/Commands` 目录下创建一个新的命令类。如果该目录不存在，不用担心，因为它将会在你首次运行 Artisan 命

令 `make:command` 时被创建。生成的命令将会包含默认的属性设置以及所有命令都共有的方法：

```
php artisan make:command SendEmails
```

命令结构

命令生成以后，需要填写该类的 `signature` 和 `description` 属性，这两个属性在调用 `list` 显示命令的时候会被用到。

`handle` 方法在命令执行时被调用，你可以将所有命令逻辑都放在这个方法里面。

注：为了更好地实现代码复用，最佳实践是保持控制台命令的轻量并让它们延迟到应用服务中完成任务。在下面的例子中，注意我们注入了一个服务类来完成发送邮件这样的“繁重”任务。

下面让我们来看一个例子，注意我们可以在命令类的构造函数中注入任何依赖，Laravel [服务提供者](#) 将会在构造函数中自动注入所有依赖类型提示：

```
<?php

namespace App\Console\Commands;

use App\User;
use App\DripEmailer;
use Illuminate\Console\Command;

class SendEmails extends Command
{
```

```
/**  
 * The name and signature of the console command.  
 *  
 * @var string  
 */  
  
protected $signature = 'email:send {user}';  
  
/**  
 * The console command description.  
 *  
 * @var string  
 */  
  
protected $description = 'Send drip e-mails to a user';  
  
/**  
 * The drip e-mail service.  
 *  
 * @var DripEmailer  
 */  
  
protected $drip;  
  
/**  
 * Create a new command instance.  
 *  
 * @param DripEmailer $drip  
 * @return void  
 */  
  
public function __construct(DripEmailer $drip)
```

```
{  
    parent::__construct();  
  
    $this->drip = $drip;  
}  
  
/**  
 * Execute the console command.  
 *  
 * @return mixed  
 */  
public function handle()  
{  
    $this->drip->send(User::find($this->argument('user')));  
}  
}
```

闭包命令

基于闭包的命令和闭报路由相对控制器一样，为以类的方式定义控制台命令提供了可选方案，在

`app/Console/Kernel.php` 文件的 `commands` 方法中，Laravel 加载了 `routes/console.php` 文件：

```
/**  
 * Register the Closure based commands for the application.  
 *  
 * @return void  
 */
```

```
protected function commands()
{
    require base_path('routes/console.php');
}
```

尽管找个文件没有定义 HTTP 路由，但是它定义了基于控制台的应用入口（和路由作用一样），在这个文件中，你可以使用 `Artisan::command` 方法定义所有基于闭包的路由。`command` 方法接收两个参数 —— 命令标识和接收命令参数和选项的闭包：

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

该闭包被绑定到底层命令实例，所有你可以像在完整的命令类中一样访问所有辅助函数。

类型提示依赖

除了接收命令参数和选项外，闭包命令还可以类型提示服务容器之外解析的额外依赖：

```
use App\User;
use App\DripEmailer;

Artisan::command('email:send {user}', function (DripEmailer $drip, $user) {
    $drip->send(User::find($user));
});
```

闭包命令描述

定义基于闭包的命令时，可以使用 `describe` 方法来添加命令描述，这个描述将会在运行 `php artisan list` 或 `php artisan help` 命令时显示：

```
Artisan::command('build {project}', function ($project) {
    $this->info("Building {$project}!");
})->describe('Build the project');
```

3、定义期望输入

编写控制台命令的时候，通常通过参数和选项收集用户输入，Laravel 使这项操作变得很方便：在命令中使用 `signature` 属性来定义我们期望的用户输入。`signature` 属性通过一个优雅的、路由风格的语法允许你定义命令的名称、参数以及选项。

参数

所有用户提供的参数和选项都包含在大括号里，下面这个例子定义的命令要求用户输入必选参数

user：

```
/**
 * 控制台命令名称
 *
 * @var string
 */
protected $signature = 'email:send {user}';
```

你还可以让该参数可选并定义默认的可选参数值：

```
// 选项参数...
email:send {user?}
// 带默认值的选项参数...
```

```
email:send {user=foo}
```

选项

选项，和参数一样，是用户输入的另一种格式，不同之处在于选项前面有两个短划线（-），有两种类型的选项：接收值和不接收值的。不接收值的选项一般用作布尔开关。我们来看一个这种类型的选项：

```
/**  
 * 控制台命令名称  
 *  
 * @var string  
 */  
protected $signature = 'email:send {user} {--queue}';
```

在本例中，`--queue` 开关在调用 Artisan 命令的时候被指定。如果`--queue` 开关被传递，其值是 `true`，否则其值是 `false`：

```
php artisan email:send 1 --queue
```

带值的选项

接下来，我们来看一个带值的选项，如果用户必须为选项指定值，需要通过`=`进行分配：

```
/**  
 * 控制台命令名称  
 *  
 * @var string  
 */  
protected $signature = 'email:send {user} {--queue=}';
```

在这个例子中，用户可以通过这样的方式传值：

```
php artisan email:send 1 --queue=default
```

还可以给选项分配默认值，如果用户没有传递值给选项，将会使用默认值：

```
email:send {user} {--queue=default}
```

选项简写

如果想要为命令选项分配一个简写，可以在选项前指定并使用分隔符|将简写和完整选项名分开：

```
email:send {user} {--Q|queue}
```

输入数组

如果你想要定义参数和选项以便指定输入数组，可以使用字符`*`，首先，让我们看一个指定数组参数的例子：

```
email:send {user*}
```

调用这个方法时，`user` 参数会顺序传递到命令行，例如，下面的命令会设置 `user` 的值为 `['foo', 'bar']`：

```
php artisan email:send foo bar
```

定义一个期望输入数组的选项时，每个传递给命令的选项值都应该加上选项名前缀：

```
email:send {user} {--id=*}
```

```
php artisan email:send --id=1 --id=2
```

输入描述

你可以通过冒号将参数和描述进行分隔的方式分配描述到输入参数和选项，如果你需要一些空间来定义命令，可以通过换行来定义命令：

```
/**  
 * 控制台命令名称  
 *  
 * @var string  
 */  
  
protected $signature = 'email:send  
    {user : The ID of the user}  
    {--queue= : Whether the job should be queued}';
```

4、命令 I/O

获取输入

在命令被执行的时候，很明显，你需要访问命令获取的参数和选项的值。使用 `argument` 和 `option` 方法即可实现：

```
/**  
 * Execute the console command.  
 *  
 * @return mixed
```

```
/*
public function handle()
{
    $userId = $this->argument('user');

    //
}
```

如果需要以数组方式返回所有参数的值，调用 `arguments` 方法：

```
$arguments = $this->arguments();
```

选项值和参数值的获取一样简单，使用 `option` 方法，要以数组方式返回所有选项值，可以调用 `options` 方法：

```
// Retrieve a specific option...
$queueName = $this->option('queue');

// Retrieve all options...
$options = $this->options();
```

如果参数或选项不存在，返回 null。

输入提示

除了显示 `输出` 之外，你可能还要在命令执行期间要用户提供输入。`ask` 方法将会使用给定问题提示用户，接收输入，然后返回用户输入到命令：

```
/**  
 * 执行控制台命令  
 *  
 * @return mixed  
 */  
public function handle()  
{  
    $name = $this->ask('What is your name?');  
}
```

`secret` 方法和 `ask` 方法类似，但用户输入在终端对他们而言是不可见的，这个方法在问用户一些敏感信息如密码时很有用：

```
$password = $this->secret('What is the password?');
```

让用户确认

如果你需要让用户确认信息，可以使用 `confirm` 方法，默认情况下，该方法返回 `false`，如果用户输入 `y`，则该方法返回 `true`：

```
if ($this->confirm('Do you wish to continue? [y|N]')) {  
    //  
}
```

自动完成

`anticipate` 方法可用于为可能的选项提供自动完成功能，用户仍然可以选择答案，而不管这些选择：

```
$name = $this->anticipate('What is your name?', ['Taylor', 'Dayle']);
```

给用户提供选择

如果你需要给用户预定义的选择，可以使用 `choice` 方法。用户选择答案的索引，但是返回给你的

是答案的值。如果用户什么都没选的话你可以设置默认返回的值：

```
$name = $this->choice('What is your name?', ['Taylor', 'Dayle'], false);
```

编写输出

要将输出发送到控制台，使用 `line`, `info`, `comment`, `question` 和 `error` 方法，每个方法都会使用相应的 ANSI 颜色以作标识。例如，要显示一条信息消息给用户， 使用 `info` 方法在终端显示为绿色：

```
/**  
 * 执行控制台命令  
 *  
 * @return mixed  
 */  
public function handle(){  
    $this->info('Display this on the screen');  
}
```

要显示一条错误消息，使用 `error` 方法。错误消息文本通常是红色：

```
$this->error('Something went wrong!');
```

如果你想要显示原生输出，可以使用 `line` 方法，该方法输出的字符不带颜色：

```
$this->line('Display this on the screen');
```

表格布局

`table` 方法使输出多行/列格式的数据变得简单，只需要将头和行传递给该方法，宽度和高度将基于给定数据自动计算：

```
$headers = ['Name', 'Email'];  
$users = App\User::all(['name', 'email'])->toArray();  
$this->table($headers, $users);
```

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();
```

想要了解更多，查看 [Symfony 进度条组件文档](#)。

4、注册命令

命令编写完成后，需要注册到 Artisan 才可以使用，这可以在 `app\Console\Kernel.php` 文件中完成。在该文件中，你会在 `commands` 属性中看到一个命令列表，要注册你的命令，只需将其加到该列表中即可。当 Artisan 启动的时候，该属性中列出的命令将会被[服务容器](#)解析并通过 Artisan 注册：

```
protected $commands = [
    'Commands\SendEmails'
];
```

5、通过代码调用命令

有时候你可能希望在 CLI 之外执行 Artisan 命令，比如，你可能希望在路由或控制器中触发 Artisan 命令，你可以使用 Artisan 门面上的 `call` 方法来完成这个。`call` 方法接收被执行的命令名称作为第一个参数，命令参数数组作为第二个参数，退出代码被返回：

```
Route::get('/foo', function () {
    $exitCode = Artisan::call('email:send', [
        'user' => 1, '--queue' => 'default'
    ]);
});
```

使用 Artisan 门面上的 `queue` 方法，你甚至可以将 Artisan 命令放到队列中，这样它们就可以通过后台的队列工作者来处理。在使用此方法之前，确保你配置好了队列并且运行了队列监听器：

```
Route::get('/foo', function () {
    Artisan::queue('email:send', [
        'user' => 1, '--queue' => 'default'
    ]);
});
```

如果你需要指定不接收字符串的选项值，例如 `migrate:refresh` 命令上的 `--force` 标识，可以传递布尔值 `true` 或 `false`：

```
$exitCode = Artisan::call('migrate:refresh', [
    '--force' => true,
]);
```

通过其他命令调用命令

有时候你希望从一个已存在的 Artisan 命令中调用其它命令。你可以通过使用 `call` 方法来实现这一目的。`call` 方法接收命令名称和数组形式的命令参数：

```
/**
```

```
* 执行控制台命令
*
* @return mixed
*/
public function handle(){
    $this->call('email:send', [
        'user' => 1, '--queue' => 'default'
    ]);
}
```

如果你想要调用其它控制台命令并阻止其所有输出，可以使用 `callSilent` 方法。`callSilent` 方法和 `call` 方法用法一致：

```
$this->callSilent('email:send', [
    'user' => 1, '--queue' => 'default'
]);
```

12.2 任务调度

1、简介

在以前，开发者需要为每一个需要调度的任务编写一个 Cron 条目，这是很让人头疼的事。你的任务调度不在源码控制中，你必须使用 SSH 登录到服务器然后添加这些 Cron 条目。

Laravel 命令调度器允许你平滑而又富有表现力地在 Laravel 中定义命令调度，并且服务器上只需要一个 Cron 条目即可。任务调度定义在 `app/Console/Kernel.php` 文件的 `schedule` 方法中，该方法中已经包含了一个示例。你可以自由地添加你需要的调度任务到 `Schedule` 对象。

开启调度器

下面是你唯一需要添加到服务器的 Cron 条目，如果你不知道如何添加 Cron 条目到服务器，可以考虑使用诸如 Laravel Forge 这样的服务来为管理 Cron 条目：

```
* * * * * php /path/to/artisan schedule:run >> /dev/null 2>&1
```

该 Cron 将会每分钟调用一次 Laravel 命令调度器，然后，Laravel 评估你的调度任务并运行到期的任务。

2、定义调度

你可以在 `App\Console\Kernel` 类的 `schedule` 方法中定义所有调度任务。开始之前，让我们看一个调度任务的例子，在这个例子中，我们将会在每天午夜调度一个被调用的闭包。在这个闭包中我们将会执行一个数据库查询来清空表：

```
<?php

namespace App\Console;

use DB;

use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;

class Kernel extends ConsoleKernel{

    /**
     * 应用提供的 Artisan 命令
     *
     * @var array
     */
    protected $commands = [
        App\Console\Commands\Inspire::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();
}
```

除了调度闭包调用外，还可以调度 [Artisan 命令](#)和操作系统命令。例如，可以使用 `command` 方法

通过命令名或类来调度一个 Artisan 命令：

```
$schedule->command('emails:send --force')->daily();
$schedule->command(EmailsCommand::class, ['--force'])->daily();
```

`exec` 命令可用于发送命令到操作系统：

```
$schedule->exec('node /home/forge/script.js')->daily();
```

调度常用选项

当然，你可以分配多种调度到任务：

方法	描述
<code>->cron('* * * * *');</code>	在自定义 Cron 调度上运行任务

方法	描述
<code>->everyMinute();</code>	每分钟运行一次任务
<code>->everyFiveMinutes();</code>	每五分钟运行一次任务
<code>->everyTenMinutes();</code>	每十分钟运行一次任务
<code>->everyThirtyMinutes();</code>	每三十分钟运行一次任务
<code>->hourly();</code>	每小时运行一次任务
<code>->daily();</code>	每天凌晨零点运行任务
<code>->dailyAt('13:00');</code>	每天 13:00 运行任务
<code>->twiceDaily(1, 13);</code>	每天 1:00 & 13:00 运行任务
<code>->weekly();</code>	每周运行一次任务
<code>->monthly();</code>	每月运行一次任务
<code>->monthlyOn(4, '15:00');</code>	每月 4 号 15:00 运行一次任务
<code>->quarterly();</code>	每个季度运行一次

方法	描述
<code>->yearly();</code>	每年运行一次
<code>->timezone('America/New_York');</code>	设置时区

这些方法可以和额外的约束一起联合起来创建一周特定时间运行的更加细粒度的调度，例如，要在每周一调度一个命令：

```
$schedule->call(function () {
    // 每周星期一 13:00 运行一次...
})->weekly()->mondays()->at('13:00');

// Run hourly from 8 AM to 5 PM on weekdays...
$schedule->command('foo')

->weekdays()
->hourly()
->timezone('America/Chicago')
->between('8:00', '17:00');
```

下面是额外的调度约束列表：

方法	描述
<code>->weekdays();</code>	只在工作日运行任务

方法	描述
<code>->sundays();</code>	每个星期天运行任务
<code>->mondays();</code>	每个星期一运行任务
<code>->tuesdays();</code>	每个星期二运行任务
<code>->wednesdays();</code>	每个星期三运行任务
<code>->thursdays();</code>	每个星期四运行任务
<code>->fridays();</code>	每个星期五运行任务
<code>->saturdays();</code>	每个星期六运行任务
<code>->between(\$start, \$end);</code>	基于特定时间段运行任务
<code>->when(Closure);</code>	基于特定测试运行任务

介于时间的约束条件

`between` 方法用于限定一天中特定时间段的任务执行：

```
$schedule->command('reminders:send')
    ->hourly()
    ->between('7:00', '22:00');
```

类似地，`unlessBetween` 方法用于排除指定时间段任务的执行：

```
$schedule->command('reminders:send')
    ->hourly()
    ->unlessBetween('23:00', '4:00');
```

真理测试的约束条件

`when` 方法用于限制任务基于给定真理测试的结果执行。换句话说，如果给定闭包返回 `true`，只要没有其它约束条件阻止任务运行，该任务就会执行：

```
$schedule->command('emails:send')->daily()->when(function () {
    return true;
});
```

`skip` 方法和 `when` 相反，如果 `skip` 方法返回 `true`，调度任务将不会执行：

```
$schedule->command('emails:send')->daily()->skip(function () {
    return true;
});
```

使用 `when` 方法链的时候，调度命令将只会执行返回 `true` 的 `when` 方法。

避免任务重叠

默认情况下，即使前一个任务仍然在运行调度任务也会运行，要避免这样的情况，可使用

`withoutOverlapping` 方法：

```
$schedule->command('emails:send')->withoutOverlapping();
```

在本例中，Artisan 命令 `emails:send` 每分钟都会运行，如果该命令没有在运行的话。如果你的任务在执行时经常大幅度的变化，那么 `withoutOverlapping` 方法就非常有用，你不必再去预测给定

任务到底要消耗多长时间。

3、任务输出

Laravel 调度器为处理调度任务输出提供了多个方便的方法。首先，使用 `sendOutputTo` 方法，你可以发送输出到文件以便稍后检查：

```
$schedule->command('emails:send')
    ->daily()
    ->sendOutputTo($filePath);
```

如果你想要追加输出到给定文件，可以使用 `appendOutputTo` 方法：

```
$schedule->command('emails:send')
    ->daily()
    ->appendOutputTo($filePath);
```

使用 `emailOutputTo` 方法，你可以将输出发送到电子邮件，注意输出必须首先通过 `sendOutputTo` 方法发送到文件。还有，使用电子邮件发送任务输出之前，应该配置 Laravel 的[电子邮件服务](#)：

```
$schedule->command('foo')
    ->daily()
    ->sendOutputTo($filePath)
    ->emailOutputTo('foo@example.com');
```

注：`emailOutputTo` 和 `sendOutputTo` 方法只对 `command` 方法有效，不支持 `call` 方法。

4、任务钩子

使用 `before` 和 `after` 方法，你可以指定在调度任务完成之前和之后要执行的代码：

```
$schedule->command('emails:send')
    ->daily()
    ->before(function () {
        // Task is about to start...
```

```
    })
    ->after(function () {
        // Task is complete...
    });
});
```

ping URL

使用 `pingBefore` 和 `thenPing` 方法 , 调度器可以在任务完成之前和之后自动 ping 给定的 URL。该方法在通知外部服务时很有用 , 例如 [Laravel Envoyer](#) , 在调度任务开始或完成的时候 :

```
$schedule->command('emails:send')
    ->daily()
    ->pingBefore($url)
    ->thenPing($url);
```

使用 `pingBefore($url)` 或 `thenPing($url)` 特性需要安装 HTTP 库 Guzzle , 可以使用 Composer 包管理器来安装 Guzzle 到项目 :

```
composer require guzzlehttp/guzzle
```

13. 测试

13.1 起步

1、简介

Laravel 植根于 [测试](#) , 实际上 , 内置使用 [PHPUnit](#) 对测试提供支持是即开即用的 , 并且 [phpunit.xml](#) 文件已经为应用设置好了。框架还提供了方便的辅助方法允许你对应用进行富有表现力的测试。

`tests` 目录中提供了一个 `ExampleTest.php` 文件 , 安装完新的 Laravel 应用后 , 只需简单在命令

行运行 `phpunit` 来运行测试。

2、环境

运行测试的时候， Laravel 会自动设置环境为 `testing`。 Laravel 在测试时自动配

置 `session` 和 `cache` 驱动为数组驱动，这意味着测试时不会持久化存储 `session` 和 `cache`。

如果需要的话你也可以创建其它测试环境配置。`testing` 环境变量可以在 `phpunit.xml` 文件中配置，但是要确保在运行命令之前使用 Artisan 命令 `config:clear` 清除配置缓存。

3、创建&运行测试

要创建一个新的测试用例，可以使用 Artisan 命令 `make:test`：

```
php artisan make:test UserTest
```

该命令将会在 `tests` 目录下生成一个新的 `UserTest` 类。然后你可以使用 [PHPUnit](#) 定义测试方法。

要运行测试，只需从终端执行 `phpunit` 命令即可：

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class UserTest extends TestCase{
    /**
     * A basic test example.
}
```

```
* @return void
*/
public function testExample()
{
    $this->assertTrue(true);
}
}
```

注：如果你在测试类中重写了 `setUp` 方法，必须在该方法中调用 `parent::setUp`。

13.2 应用测试

1、简介

Laravel 为生成 HTTP 请求、测试输出、以及填充表单提供了流式 API。举个例子，我们看

下 `tests` 目录下包含的 `ExampleTest.php` 文件：

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase{
    /**
     * 基本功能测试示例
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->visit('/')
            ->see('Laravel 5')
            ->dontSee('Rails');
    }
}
```

```
}
```

`visit` 方法生成了一个 GET 请求，`see` 方法对我们从应用返回响应中应该看到的给定文本进行断言。`dontSee` 方法对给定文本没有从应用响应中返回进行断言。在 Laravel 中这是最基本的有效性测试。

你还可以使用 `visitRoute` 方法通过命名路由生成 GET 请求：

```
$this->visitRoute('profile');
```

```
$this->visitRoute('profile', ['user' => 1]);
```

2、与应用交互

当然，除了对响应文本进行断言之外还有做更多测试，让我们看一些点击链接和填充表单的例子：

点击链接

在本测试中，我们将为应用生成请求，在返回的响应中“点击”链接，然后对访问 URI 进行断言。

例如，假定响应中有一个“关于我们”的链接：

```
<a href="/about-us">About Us</a>
```

现在，让我们编写一个测试点击链接并断言用户访问页面是否正确：

```
public function testBasicExample(){
```

```
    $this->visit('/')
```

```
    ->click('About Us')
```

```
    ->seePageIs('/about-us');
```

```
}
```

还可以使用 `seeRouteIs` 方法检查用户是否到达正确的命名路由：

```
->seeRouteIs('profile', ['user' => 1]);
```

处理表单

Laravel 还为处理表单提供了多个方法。`type`、`select`、`check`、`attach` 和 `press` 方法允许你与所有表单输入进行交互。例如，我们假设这个表单存在于应用注册页面：

```
<form action="/register" method="POST">
    {!! csrf_field() !!}

    <div>
        Name: <input type="text" name="name">
    </div>

    <div>
        <input type="checkbox" value="yes" name="terms"> Accept Terms
    </div>

    <div>
        <input type="submit" value="Register">
    </div>
</form>
```

我们可以编写测试来完成表单并检查结果：

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

```
}
```

当然，如果你的表单包含其他输入比如单选按钮或下拉列表，也可以轻松填写这些类型的字段。

这里是所有表单操作方法列表：

方法	描述
<code>\$this->type(\$text, \$elementName)</code>	“Type” 文本到给定字段
<code>\$this->select(\$value, \$elementName)</code>	“Select” 单选框或下拉列表
<code>\$this->check(\$elementName)</code>	“Check” 复选框
<code>\$this->attach(\$pathToFile, \$elementName)</code>	“Attach” 文件到表单
<code>\$this->press(\$buttonTextOrElementName)</code>	“Press” 给定文本或 name 的按钮
<code>\$this->uncheck(\$elementName)</code>	“Uncheck”复选框

文件上传

如果表单包含 `file` 输入类型，可以使用 `attach` 方法添加文件到表单：

```
public function testPhotoCanBeUploaded(){
    $this->visit('/upload')
        ->attach($pathToFile, 'photo')
        ->press('Upload')
        ->see('Upload Successful!');
```

```
}
```

3、测试 JSON API

Laravel 还提供多个帮助函数用于测试 JSON API 及其响应。例如，`json`、`get`、`post`、`put`、`patch` 和 `delete` 方法用于通过多种 HTTP 请求方式发出请求。你还可以轻松传递数据和请求头到这些方法。作为开始，我们编写测试来生成 POST 请求到 `/user` 并断言返回的数据是否是我们所期望的：

```
<?php

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->json('POST', '/user', ['name' => 'Sally'])
            ->seeJson(['created' => true,]);
    }
}
```

`seeJson` 方法将给定数组转化为 JSON，然后验证应用返回的整个 JSON 响应中的 JSON 片段。

因此，如果在 JSON 响应中有其他属性，只要给定片段存在的话测试依然会通过。

验证 JSON 值匹配

如果你想要验证给定数组和应用返回的 JSON 能够精确匹配，使用 `seeJsonEquals` 方法：

```
<?php

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->json('POST', '/user', ['name' => 'Sally'])
            ->seeJsonEquals(['created' => true,]);
    }
}
```

验证数据结构匹配

还可以验证 JSON 响应是否与指定数据结构匹配，我们使用 `seeJsonStructure` 方法来实现这一功能：

```
<?php
```

```

class ExampleTest extends TestCase{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->get('/user/1')
            ->seeJsonStructure([
                'name',
                'pet' => [
                    'name', 'age'
                ]
            ]);
    }
}

```

上面的例子演示了期望获取一个包含 `name` 和嵌套 `pet` 对象（该对象包含 `name` 和 `age` 属性）的 JSON 数据。如果 JSON 响应中包含其它额外键 `seeJsonStructure` 也不会失败，例如，如果 `pet` 对象包含 `weight` 属性测试仍将通过。

你可以使用`*`来断言返回 JSON 结构包含一个列表，该列表中的每个数据项都包含至少如下示例中列出的属性：

```

<?php

class ExampleTest extends TestCase{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        // Assert that each user in the list has at least an id, name and email attribute.
    }
}

```

```
$this->get('/users')
    ->seeJsonStructure([
        '*' => [
            'id', 'name', 'email'
        ]
    ]);
}
```

你还可以使用嵌套的`*`，在这种场景中，我们可以断言 JSON 响应中的每个用户都包含一个给定属性集合，而且每个用户的每个 `pet` 都包含给定属性集合：

```
$this->get('/users')
    ->seeJsonStructure([
        '*' => [
            'id', 'name', 'email', `pets` => [
                '*' => [
                    'name', 'age'
                ]
            ]
        ]
    ]);
});
```

4、Session/认证

Laravel 提供了多个辅助函数用于在测试期间处理 `Session`，首先，可以使用 `withSession` 方法设置 session 值到给定数组。这在测试请求前获取 session 数据时很有用：

```
<?php

class ExampleTest extends TestCase{
    public function testApplication()
    {
```

```
$this->withSession(['foo' => 'bar'])  
    ->visit('/');  
}  
}
```

当然，session 的最常见的作用就是维护认证用户的状态。辅助函数 `actingAs` 为认证给定用户是当前用户提供了简单的实现方法，例如，我们使用模型工厂生成和认证用户：

```
<?php  
  
class ExampleTest extends TestCase{  
    public function testApplication()  
    {  
        $user = factory(App\User::class)->create();  
  
        $this->actingAs($user)  
            ->withSession(['foo' => 'bar'])  
            ->visit('/')  
            ->see('Hello, ' . $user->name);  
    }  
}
```

还可以通过传递 `guard` 名称作为 `actingAs` 函数的第二个参数的方式来指定使用哪个 `guard` 来认证给定用户：

```
$this->actingAs($user, 'backend')
```

5、禁止中间件

测试应用时，为某些测试禁止中间件很方便。这种机制允许你将路由和控制器与中间件孤立开来

做测试， Laravel 包含了一个简单的 `WithoutMiddleware` trait，可以使用该 trait 自动在测试类中

禁止所有中间件：

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase{
    use WithoutMiddleware;
    //
}
```

如果你只想在某些方法中禁止中间件，可以在测试方法中调用 `withoutMiddleware` 方法：

```
<?php

class ExampleTest extends TestCase{
    /**
     * 基本功能测试示例
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->withoutMiddleware();

        $this->visit('/')
            ->see('Laravel 5');
    }
}
```

6、自定义 HTTP 请求

如果你想要在应用中生成自定义 HTTP 请求并获取完整的 `Illuminate\Http\Response` 对象，可以使用 `call` 方法：

```
public function testApplication(){
    $response = $this->call('GET', '/');
    $this->assertEquals(200, $response->status());
}
```

如果你要生成 `POST`、`PUT` 或者 `PATCH` 请求可以在请求中传入输入数据数组，在路由或控制器中可以访问 `Request` 实例访问请求数据：

```
$response = $this->call('POST', '/user', ['name' => 'Taylor']);
```

7、PHPUnit 断言方法

Laravel 为 PHPUnit 测试提供了额外的断言方法：

方法	描述
<code>->assertResponseOk();</code>	断言客户端响应状态码是否为 200
<code>->assertResponseStatus(\$code);</code>	断言客户端响应状态码是否是给定\$code
<code>->assertViewHas(\$key, \$value = null);</code>	断言响应视图是否包含给定的绑定数据段
<code>->assertViewHasAll(array \$bindings);</code>	断言视图是否包含给定绑定数据列表

方法	描述
<code>->assertViewMissing(\$key);</code>	断言响应视图缺失绑定数据片段
<code>->assertRedirectedTo(\$uri, \$with = []);</code>	断言客户端是否重定向到给定 URI
<code>->assertRedirectedToRoute(\$name, \$parameters = [], \$with = []);</code>	断言客户端是否重定向到给定路由
<code>->assertRedirectedToAction(\$name, \$parameters = [], \$with = []);</code>	断言客户端是否重定向到给定 action
<code>->assertSessionHas(\$key, \$value = null);</code>	断言 session 是否包含给定值
<code>->assertSessionHasAll(array \$bindings);</code>	断言 session 是否保护眼给定值列表
<code>->assertSessionHasErrors(\$bindings = [], \$format = null);</code>	断言 session 是否包含错误绑定
<code>->assertHasOldInput();</code>	断言 session 包含上次输入数据
<code>->assertSessionMissing(\$key);</code>	断言 session 缺失给定 key

13.3 数据库

1、简介

Laravel 提供了多个有用的工具让[测试数据库](#)驱动的应用变得更加简单。首先，你可以使用辅助函

数 `seeInDatabase` 来断言 数据库中的数据是否和给定数据集合匹配。例如 ,如果你想要通过 email 值为 `sally@example.com` 的条件去数据表 `users` 查询是否存在该记录 ，我们可以这样做：

```
public function testDatabase(){
    // 调用应用...
    $this->seeInDatabase('users', ['email' => 'sally@example.com']);
}
```

当然，`seeInDatabase` 方法和其它类似辅助方法都是为了方便起见进行的封装，你也可以使用其它 PHPUnit 内置的断言方法来进行测试。

2、每次测试后重置数据库

每次测试后重置数据库通常很有用，这样的话上次测试的数据不会影响下一次测试。

使用迁移

一种重置数据库状态的方式是每次测试后回滚数据库并在下次测试前重新迁移。 Laravel 提供了一个简单的 `DatabaseMigrations` trait 来自动为你处理。在测试类上简单使用该 trait 如下：

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase{
    use DatabaseMigrations;

    /**
     * 基本功能测试示例

```

```
* @return void
*/
public function testBasicExample()
{
    $this->visit('/')
        ->see('Laravel 5');
}
```

使用事务

另一种重置数据库状态的方式是将每一个测试用例封装到一个数据库事务中， Laravel 提供了方便的 `DatabaseTransactions` trait 自动为你处理：

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase{
    use DatabaseTransactions;

    /**
     * 基本功能测试示例
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->visit('/')
            ->see('Laravel 5');
    }
}
```

注：该 trait 只在事务中封装默认数据库连接。如果你的应用使用了多个数据库连接，需要手动处理这些连接的事务逻辑。

3、编写工厂

测试时，通常需要在执行测试前插入新数据到数据库。在创建测试数据时，Laravel 允许你使用“factories”为每个 [Eloquent 模型](#) 定义默认的属性值集合，而不用手动为每一列指定值。作为开始，我们看一下 `database/factories/ModelFactory.php` 文件，该文件包含了一个工厂定义：

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

在闭包中，作为工厂定义，我们返回该[模型](#)上所有属性默认测试值。该闭包接收 PHP 库 [Faker](#) 实例，从而允许你方便地为测试生成多种类型的随机数据。

当然，你可以添加更多工厂到 `ModelFactory.php` 文件。你还可以为每个模型创建额外的工厂文件以便更好地组织管理，例如，你可以在 `database/factories` 目录下创建 `UserFactory.php` 和 `CommentFactory.php` 文件。`factories` 目录下的所有文件都会被 Laravel 自动加载。

工厂状态

状态允许你在任意组合中定义可用于[模型工厂](#)的离散修改，例如，`User` 模型可能有一个 `delinquent` 状态用于修改某个默认属性值，你可以使用 `state` 方法来定义状态转化：

```
$factory->state(App\User::class, 'delinquent', function ($faker) {
```

```
return [
    'account_status' => 'delinquent',
];
});
```

4、使用工厂

创建模型

定义好工厂后，可以在测试或数据库填充文件中通过全局的 `factory` 方法使用它们来生成模型实例，所以，让我们看一些创建模型的例子，首先，我们使用 `make` 方法，该方法创建模型但不将其保存到数据库：

```
public function testDatabase(){
    $user = factory(App\User::class)->make();
    // 用户模型测试...
}
```

还可以创建多个模型集合或者创建给定类型的模型：

```
// 创建 3 个 App\User 实例...
$users = factory(App\User::class, 3)->make();
// 创建 1 个 App\User "admin" 实例...
$user = factory(App\User::class, 'admin')->make();
// 创建 3 个 App\User "admin" 实例...
$users = factory(App\User::class, 'admin', 3)->make();
```

应用状态

还可以应用任意状态到模型，如果你想要应用多个状态转化到模型，需要指定每个你想要应用的

状态名：

```
$users = factory(App\User::class, 5)->states('deliquent')->make();  
$users = factory(App\User::class, 5)->states('premium', 'deliquent')->make();
```

覆盖属性

如果你想要覆盖模型中的某些默认值，可以传递数组值到模型，只有指定值才会被替换，剩下值

保持工厂指定的默认值不变：

```
$user = factory(App\User::class)->make([  
    'name' => 'Abigail',  
]);
```

持久化模型

`create` 方法不仅能创建模型实例，还可以使用 Eloquent 的 `save` 方法将它们保存到数据库：

```
public function testDatabase()  
{  
    // Create a single App\User instance...  
  
    $user = factory(App\User::class)->create();  
  
    // Create three App\User instances...  
  
    $users = factory(App\User::class, 3)->create();
```

```
// Use model in tests...
}
```

你仍然可以通过传递数组到 `create` 方法覆盖模型上的属性：

```
$user = factory(App\User::class)->create([
    'name' => 'Abigail',
]);
```

关联关系

在本例中，我们添加一个关联到创建的模型，使用 `create` 方法创建多个模型的时候，会返回一个

[Eloquent 集合](#) 实例，从而允许你使用集合提供的所有方法，例如 `each`：

```
$users = factory(App\User::class, 3)
    ->create()
    ->each(function($u) {
        $u->posts()->save(factory(App\Post::class)->make());
    });
});
```

关联关系 & 属性闭包

还可以使用工厂中的闭包属性添加关联关系到模型，例如，如果你想要在创建 `Post` 的时候创建一个新的 `User` 实例，可以这么做：

```
$factory->define(App\Post::class, function ($faker) {
    return [
        'title' => $faker->title,
        'content' => $faker->paragraph,
        'user_id' => function () {
```

```
        return factory(App\User::class)->create()->id;  
    }  
];  
});
```

这些闭包还接收包含它们的工厂属性数组：

```
$factory->define(App\Post::class, function ($faker) {  
    return [  
        'title' => $faker->title,  
        'content' => $faker->paragraph,  
        'user_id' => function () {  
            return factory(App\User::class)->create()->id;  
        },  
        'user_type' => function (array $post) {  
            return App\User::find($post['user_id'])->type;  
        }  
    ];  
});
```

13.4 模拟

1、简介

测试 Laravel 应用的时候，你可能还想要“[模拟](#)”应用的特定状态，以便在测试中不让它们真的执行。例如，测试触发[事件](#)的控制器时，你可能想要模拟事件监听器以便它们不在测试期间真的执行。这样的话你就可以只测试控制器的 HTTP 响应，而不必担心事件监听器的执行，因为事件监听器可以在它们自己的测试用例中被测试。

Laravel 开箱为模拟事件、[任务](#)以及工厂提供了辅助函数，这些辅助函数主要是在 [Mockery](#) 之上提供了一个方便的层这样你就不必手动调用复杂的 Mockery 方法。当然，你也可以使用 [Mockery](#) 或 PHPUnit 来创建自己的模拟。

2、事件

使用 Mocks

如果你在重度使用 Laravel 的事件系统，可能想要在测试时模拟特定事件。例如，如果你在测试用户注册，你可能不想所有 `UserRegistered` 事件的处理器都被触发，因为这可能会发送欢迎邮件，等等。

Laravel 提供了一个方便的 `expectsEvents` 方法来验证期望的事件被触发，但同时阻止该事件的其它处理器运行：

```
<?php

use App\Events\UserRegistered;

class ExampleTest extends TestCase
{
    /**
     * Test new user registration.
     */
    public function testUserRegistration()
    {
        $this->expectsEvents(UserRegistered::class);

        // Test user registration...
    }
}
```

可以使用 `doesn'tExpectEvents` 方法来验证给定事件没有被触发：

```
<?php

use App\Events\OrderShipped;
use App\Events\OrderFailedToShip;

class ExampleTest extends TestCase
{
    /**
     * Test order shipping.
     */
    public function testOrderShipping()
    {
        $this->expectsEvents(OrderShipped::class);
        $this->doesn'tExpectEvents(OrderFailedToShip::class);

        // Test order shipping...
    }
}
```

如果你想要阻止所有事件运行，可以使用 `withoutEvents` 方法，当这个方法被调用时，所有事件的监听器都会被模拟：

```
<?php

class ExampleTest extends TestCase{
    public function testUserRegistration()
    {
```

```
$this->withoutEvents();
// 测试用户注册代码...
}
}
```

使用 Fakes

作为模拟的可选方案，你可以使用 [Event 门面](#) 的 `fake` 方法来阻止所有事件监听器的执行。你可以断言事件被触发甚至检查它们接收的数据。使用 `fake` 的时候，断言在测试代码执行后执行：

```
<?php

use App\Events\OrderShipped;
use App\Events\OrderFailedToShip;
use Illuminate\Support\Facades\Event;

class ExampleTest extends TestCase
{
    /**
     * 测试订单发货
     */
    public function testOrderShipping()
    {
        Event::fake();

        // 执行订单发货...

        Event::assertFired(OrderShipped::class, function ($e) use ($order) {

```

```
        return $e->order->id === $order->id;

    });

    Event::assertNotFired(OrderFailedToShip::class);

}

}
```

3、任务

使用 Mocks

有时候，你可能想要在请求时简单测试控制器分发的指定任务，这允许你孤立的测试路由/控制器——将其从任务逻辑中分离出去，当然，接下来你可以在一个独立测试类中测试任务本身。

Laravel 提供了一个方便的 `expectsJobs` 方法来验证期望的任务被分发，但该任务本身不会被测试：

```
<?php

use App\Jobs\ShipOrder;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
    {
        $this->expectsJobs(ShipOrder::class);
```

```
// Test order shipping...
}

}
```

注：这个方法只检查通过 `DispatchesJobs` trait 分发方法或辅助函数 `dispatch` 分发的任务，并不检查直接通过 `Queue::push` 分发的任务。

和事件模拟辅助函数一样，你还可以使用 `doesntExpectJobs` 方法测试一个任务没有被分发：

```
<?php

use App\Jobs\ShipOrder;

class ExampleTest extends TestCase
{
    /**
     * Test order cancellation.
     */
    public function testOrderCancellation()
    {
        $this->doesntExpectJobs(ShipOrder::class);

        // Test order cancellation...
    }
}
```

或者，你还可以使用 `withoutJobs` 方法忽略所有分发任务。当该方法在测试方法中被调用时，所有该测试期间被分发的任务都会被废弃：

```
<?php

use App\Jobs\ShipOrder;

class ExampleTest extends TestCase
{
    /**
     * 测试订单取消
     */
    public function testOrderCancellation()
    {
        $this->withoutJobs();

        // 测试订单取消...
    }
}
```

使用 Fakes

作为 mock 的可选方案，你可以使用 Queue 门面的 fake 方法来阻止任务被推送到队列。之后你可以断言任务是否被推送到队列。甚至检查接收的数据，使用 fakes 的时候，断言会在测试代码执行后进行：

```
<?php

use App\Jobs\ShipOrder;
```

```
use Illuminate\Support\Facades\Queue;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
    {
        Queue::fake();

        // 订单发货...

        Queue::assertPushed(ShipOrder::class, function ($job) use ($order) {
            return $job->order->id === $order->id;
        });

        // 断言任务是否被推送到给定队列...
        Queue::assertPushedOn('queue-name', ShipOrder::class);

        // 断言任务未被推送...
        Queue::assertNotPushed(AnotherJob::class);
    }
}
```

4、邮件 Fakes

你可以使用 `Mail` 门面的 `fake` 方法阻止邮件被发送。然后断言邮件是否被发送到用户，甚至可以检查接收的数据，使用 `fakes` 的时候，断言会在测试代码执行后进行：

```
<?php

use App\Mail\OrderShipped;
use Illuminate\Support\Facades\Mail;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
    {
        Mail::fake();

        // 订单发货...
        Mail::assertSent(OrderShipped::class, function ($mail) use ($order) {
            return $mail->order->id === $order->id;
        });

        // 断言消息被发送到给定用户...
        Mail::assertSentTo([$user], OrderShipped::class);

        // 断言邮件未被发送...
        Mail::assertNotSent(AnotherMailable::class);
    }
}
```

5、通知 Fakes

你可以使用 `Notification` 门面的 `fake` 方法来阻止通知被发送，之后断言通知是否被发送给用户，

甚至可以检查接收的数据。使用 `fakes` 的时候，断言会在测试代码执行后进行：

```
<?php

use App\Notifications\OrderShipped;
use Illuminate\Support\Facades\Notification;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
    {
        Notification::fake();

        // 订单发货...
        Notification::assertSentTo(
            $user,
            OrderShipped::class,
            function ($notification, $channels) use ($order) {
                return $notification->order->id === $order->id;
            }
        );

        // 断言通知被发送到给定用户...
        Notification::assertSentTo(
            [$user], OrderShipped::class
        );
    }
}
```

```
// 断言通知未被发送...
Notification::assertNotSentTo(
    [$user], AnotherNotification::class
);
}

}
```

6、门面

不同于传统的静态方法调用，门面可以被模拟。这与传统静态方法相比是一个巨大的优势，并且你可以对依赖注入进行测试。测试的时候，你可能经常想要在控制器中模拟 [Laravel 门面](#) 的调用，例如，看看下面的控制器动作：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

class UserController extends Controller{
    /**
     * 显示应用用户列表
     *
     * @return Response
     */
    public function index()
```

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

我们可以通过使用 `shouldReceive` 方法模拟 `Cache` 门面的调用，该方法返回一个 [Mockery](#) 模拟的实例，由于门面通过 Laravel [服务容器](#) 进行解析和管理，所以它们比通常的静态类更具有可测试性。例如，我们可以来模拟 `Cache` 门面 `get` 方法的调用：

```
<?php  
  
class FooTest extends TestCase{  
    public function testGetIndex()  
    {  
        Cache::shouldReceive('get')  
            ->once()  
            ->with('key')  
            ->andReturn('value');  
  
        $this->visit('/users')->see('value');  
    }  
}
```

注：不要模拟 `Request` 门面，取而代之地，可以在测试时传递期望输入到 HTTP 辅助函数如 `call` 和 `post`。

14. 官方包

14.1 Laravel Cashier

1、简介

[Laravel Cashier](#) 为通过 [Stripe](#) 实现[订阅支付](#)服务提供了一个优雅的流式接口。它封装了几乎所有你恐惧编写的样板化的订阅支付代码。除了基本的订阅管理外，Cashier 还支持处理[优惠券](#)、订阅升级/替换、订阅“数量”、取消宽限期，甚至生成 PDF [发票](#)。

注：如果你只需要一次性支付，并不提供订阅，就不应该使用 Cashier，而是直接使用 Stripe 和 [Braintree](#)SDK。

2、配置

Stripe

Composer

首先，添加 Cashier 包到 `composer.json` 文件并运行 `composer update` 命令：

```
"laravel/cashier": "~7.0"
```

服务提供者

接 下 来 ， 在 `config/app.php` 配 置 文 件 中 注 册 [服 务 提 供 者](#)：

```
Laravel\Cashier\CashierServiceProvider。
```

数据库迁移

使用 Cashier 之前，我们需要准备好数据库。我们需要添加一个字段到 `users` 表，还要创建新

的 `subscriptions` 表来处理所有用户订阅：

```
Schema::table('users', function ($table) {  
    $table->string('stripe_id')->nullable();  
    $table->string('card_brand')->nullable();  
    $table->string('card_last_four')->nullable();  
    $table->timestamp('trial_ends_at')->nullable();  
});  
  
Schema::create('subscriptions', function ($table) {  
    $table->increments('id');  
    $table->integer('user_id');  
    $table->string('name');  
    $table->string('stripe_id');  
    $table->string('stripe_plan');  
    $table->integer('quantity');  
    $table->timestamp('trial_ends_at')->nullable();  
    $table->timestamp('ends_at')->nullable();  
    $table->timestamps();  
});
```

创建好迁移后，只需简单运行 `migrate` 命令，相应修改就会更新到数据库。

Billable 模型

接下来，添加 `Billable` trait 到 `User` 模型类，这个 trait 提供了多个方法以便执行常用支付任务，

例如创建订阅、使用优惠券以及更新信用卡信息：

```
use Laravel\Cashier\Billable;

class User extends Authenticatable
{
    use Billable;
}
```

API Key

最后，在配置文件 `config/services.php` 中配置 Stripe 的 key，你可以在 Stripe 官网个人中心的控制面板中获取这些 Stripe API key 信息：

```
'stripe' => [
    'model'  => App\User::class,
    'secret' => env('STRIPE_API_SECRET'),
],
```

Braintree

Braintree 注意事项

对于很多操作，Stripe 和 Braintree 实现的 Cashier 功能都是一样的，两者都提供了通过信用卡进行订阅支付的功能，但是 Braintree 还支持通过 PayPal 支付。不过，Braintree 也缺失一些 Stripe 支持的功能，在决定使用 Stripe 还是 Braintree 之前，需要考虑以下几点：

- Braintree 支持 PayPal 而 Stripe 不支持；
- Braintree 不支持 `increment` 和 `decrement` 方法，这个是 Braintree 级别的限制，不是 Cashier 级别的；

- Braintree 不支持基于百分比的折扣，这个也是 Braintree 级别的限制，不是 Cashier 级别的

Composer

首先，添加针对 Braintree 的 Cashier 扩展包到 `composer.json` 文件并运行 `composer update` 命令：

```
"laravel/cashier-braintree": "~2.0"
```

服务提供者

接下来，在配置文件 `config/app.php` 中注册服务提供者 `Laravel\Cashier\CashierServiceProvider`。

计划信用优惠券

在使用基于 Braintree 的 Cashier 之前，需要在 Braintree 的控制面板中定义一个 `plan-credit` 折扣，这个折扣将会用于从年到月或者从月到年支付的优惠额度比例分配。

配置在 Braintree 控制面板中的这个折扣值可以是你希望的任何值，Cashier 将会在每次使用优惠券的时候通过自定义的值来重写这个默认定义的值。该优惠券之所以是必须的是因为 Braintree 不支持本地根据订阅时长进行优惠额度的按比例分配。

数据库迁移

使用 Cashier 之前，需要准备好数据库。我们需要添加额外的字段到 `users` 表并创建一个新的 `subscriptions` 表用于存放所有用户订阅：

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

```
$table->string('braintree_id')->nullable();
$table->string('paypal_email')->nullable();
$table->string('card_brand')->nullable();
$table->string('card_last_four')->nullable();
$table->timestamp('trial_ends_at')->nullable();

});

Schema::create('subscriptions', function ($table) {
    $table->increments('id');
    $table->integer('user_id');
    $table->string('name');
    $table->string('braintree_id');
    $table->string('braintree_plan');
    $table->integer('quantity');
    $table->timestamp('trial_ends_at')->nullable();
    $table->timestamp('ends_at')->nullable();
    $table->timestamps();
});

});
```

迁移被创建之后，只需要执行 Artisan 命令 `migrate` 即可。

Billable 模型

接下来，添加 `Billable` trait 到模型定义：

```
use Laravel\Cashier\Billable;

class User extends Authenticatable
{
```

```
use Billable;  
}
```

API Key

接下来，需要在配置文件 `services.php` 中配置如下选项：

```
'braintree' => [  
    'model' => App\User::class,  
    'environment' => env('BRAINTREE_ENV'),  
    'merchant_id' => env('BRAINTREE_MERCHANT_ID'),  
    'public_key' => env('BRAINTREE_PUBLIC_KEY'),  
    'private_key' => env('BRAINTREE_PRIVATE_KEY'),  
,
```

然后你需要在服务提供者 `AppServiceProvider` 的 `boot` 方法中添加对 Braintree SDK 的调用：

```
\Braintree_Configuration::environment(config('services.braintree.environment'));  
\Braintree_Configuration::merchantId(config('services.braintree.merchant_id'));  
\Braintree_Configuration::publicKey(config('services.braintree.public_key'));  
\Braintree_Configuration::privateKey(config('services.braintree.private_key'));
```

货币配置

Cashier 默认货币是美元（ USD ），你可以在某个服务提供者的 `boot` 方法中通过调用 `Cashier::useCurrency` 方法来更改默认的货币， `useCurrency` 方法接收两个字符串参数：货币及货币符号：

```
use Laravel\Cashier\Cashier;  
  
Cashier::useCurrency('eur', '€');
```

3、订阅

创建订阅

要创建一个订阅，首先要获取一个账单模型的实例，通常是 `App\User` 的实例。获取到该模型实例之后，你可以使用 `newSubscription` 方法来创建该模型的订阅：

```
$user = User::find(1);  
  
$user->newSubscription('main', 'monthly')->create($creditCardToken);
```

第一个传递给 `newSubscription` 方法的参数是该订阅的名字，如果应用只有一个订阅，可以将其称作 `main` 或 `primary`，第二个参数用于指定用户订阅的 Stripe/Braintree 计划，该值对应 Stripe 或 Braintree 中相应计划的 id。

`create` 方法会自动创建这个 Stripe 订阅，同时更新数据库中 Stripe 的客户 ID（即 `users` 表中的 `stripe_id`）和其它相关的账单信息。

额外的用户信息

如果你想要指定额外的客户信息，你可以将其作为第二个参数传递给 `create` 方法：

```
$user->newSubscription('main', 'monthly')->create($creditCardToken, [  
    'email' => $email,  
]);
```

要了解更多 Stripe 支持的字段，可以查看 Stripe 关于[创建消费者的文档](#)或者相应的 [Braintree 文档](#)。

优惠券

如果你想要在创建订阅的时候使用优惠券，可以使用 `withCoupon` 方法：

```
$user->newSubscription('main', 'monthly')
    ->withCoupon('code')
    ->create($creditCardToken);
```

检查订阅状态

用户订阅你的应用后，你可以使用各种便利的方法来简单检查订阅状态。首先，如果用户有一个有效的订阅，则 `subscribed` 方法返回 `true`，即使订阅现在处于[试用期](#)：

```
if ($user->subscribed('main')) {
    //
}
```

`subscribed` 方法还可以用于[路由中间件](#)，基于用户订阅状态允许你对路由和控制器的访问进行过滤：

```
public function handle($request, Closure $next){
    if ($request->user() && ! $request->user()->subscribed('main'))
    {
        // This user is not a paying customer...
        return redirect('billing');
    }

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

如果你想要判断一个用户是否还在试用期，可以使用 `onTrial` 方法，该方法对于还处于试用期的用户显示警告信息很有用：

```
if ($user->subscription('main')->onTrial()) {  
    //  
}
```

`subscribedToPlan` 方法可用于判断用户是否基于 Stripe/Braintree ID 订阅了给定的计划，在本例中，我们会判断用户的 `main` 订阅是否订阅了 `monthly` 计划：

```
if ($user->subscribedToPlan('monthly', 'main')) {  
    //  
}
```

已取消的订阅状态

要判断用户是否曾经是有效的订阅者，但现在取消了订阅，可以使用 `cancelled` 方法：

```
if ($user->subscription('main')->cancelled()) {  
    //  
}
```

你还可以判断用户是否曾经取消过订阅，但现在仍然在“宽限期”直到完全失效。例如，如果一个用户在 3 月 5 号取消了一个实际有效期到 3 月 10 号的订阅，该用户处于“宽限期”直到 3 月 10 号。注意 `subscribed` 方法在此期间仍然返回 `true`。

```
if ($user->subscription('main')->onGracePeriod()) {  
    //
```

```
}
```

修改计划

用户订阅应用后，偶尔想要改变到新的订阅计划，要将用户切换到新的订阅，传递计划标识到 `swap` 方法：

```
$user = App\User::find(1);  
$user->subscription('main')->swap('provider-plan-id');
```

如果用户在试用，试用期将会被维护。还有，如果订阅存在多个，数量也可以被维护。

如果你想要切换计划并取消用户所在的所有试用期，你可以使用 `skipTrial` 方法：

```
$user->subscription('main')  
    ->skipTrial()  
    ->swap('provider-plan-id');
```

订阅数量

注：只有 Stripe 版本的 Cashier 支持订阅数量，Braintree 没有提供类似 Stripe“数量”这样的特性。

有时候订阅也会被数量影响，例如，应用中每个账户每月需要付费\$10，要简单增加或减少订阅数量，使用 `incrementQuantity` 和 `decrementQuantity` 方法：

```
$user = User::find(1);  
  
$user->subscription('main')->incrementQuantity();
```

```
// Add five to the subscription's current quantity...
$user->subscription('main')->incrementQuantity(5);

$user->subscription('main')->decrementQuantity();

// Subtract five to the subscription's current quantity...
$user->subscription('main')->decrementQuantity(5);
```

你也可以使用 `updateQuantity` 方法指定数量：

```
$user->subscription('main')->updateQuantity(10);
```

想要了解更多订阅数量信息，查阅相关 [Stripe 文档](#)。

订阅税金

要指定用户支付订阅的税率，实现账单模型的 `taxPercentage` 方法，并返回一个在 0 到 100 之间的数值，不要超过两位小数：

```
public function taxPercentage() {
    return 20;
}
```

这将使你可以在模型基础上使用税率，对跨越不同国家不同税率的用户很有用。

注：`taxPercentage` 方法只能用于订阅支付，如果你使用 Cashier 生成一次性账单，需要手动指定税率。

取消订阅

要取消订阅，可以调用用户订阅上的 `cancel` 方法：

```
$user->subscription('main')->cancel();
```

当订阅被取消时，Cashier 将会自动设置数据库中的 `ends_at` 字段。该字段用于了解 `subscribed` 方法什么时候开始返回 `false`。例如，如果客户 3 月 1 号份取消订阅，但订阅直到 3 月 5 号才会结束，那么 `subscribed` 方法继续返回 `true` 直到 3 月 5 号。

你可以使用 `onGracePeriod` 方法判断用户是否已经取消订阅但仍然在“宽限期”：

```
if ($user->subscription('main')->onGracePeriod()) {  
    //  
}
```

如果你想要立即取消订阅，调用用户订阅上的方法 `cancelNow` 即可：

```
$user->subscription('main')->cancelNow();
```

恢复订阅

如果用户已经取消订阅但想要恢复该订阅，可以使用 `resume` 方法，前提是该用户必须在宽限期内：

```
$user->subscription('main')->resume();
```

如果该用户取消了一个订阅然后在订阅失效之前恢复了这个订阅，则不会立即支付该账单，取而代之的，他们的订阅只是被重新激活，并回到正常的支付周期。

更新信用卡

`updateCard` 方法可用于更新用户的信用卡信息，该方法接收一个 Stripe 令牌并设置新的信用卡

作为支付源：

```
$user->updateCard($creditCardToken);
```

4、订阅试用期

带信用卡信息

如果你想要在提供给用户试用期的同时预先收集支付方式信息，可以在创建订阅的时候使用 `trialDays` 方法：

```
$user = User::find(1);

$user->newSubscription('main', 'monthly')
    ->trialDays(10)
    ->create($creditCardToken);
```

该方法会在数据库订阅记录上设置试用期结束日期，以便告知 Stripe/Braintree 在此之前不要计算用户的账单信息。

注：如果用户的订阅没有在试用期结束之前取消，则会在试用期结束时立即支付，所以要确保通知用户试用期结束时间。

可以使用用户实例或订阅实例上的 `onTrial` 方法判断用户是否处于试用期，下面两个例子作用是等价的：

```
if ($user->onTrial('main')) {
    //
}
```

```
if ($user->subscription('main')->onTrial()) {  
    //  
}
```

不带信用卡信息

如果你不想在提供试用期的时候收集用户支付方式信息，只需设置用户记录的 `trial_ends_at` 字段为期望的试用期结束日期即可，这通常在用户注册期间完成：

```
$user = User::create([  
    // Populate other user properties...  
    'trial_ends_at' => Carbon::now()->addDays(10),  
]);
```

注：确保已添加 `trial_ends_at` 日期修改器到模型定义。

Cashier 将这种类型的试用期看作“一般体验”，因为这种使用并没有附加到任何已经在的订阅，如果当前日期没有超过 `trial_ends_at` 的值，`User` 实例上的 `onTrial` 方法将返回 `true`：

```
if ($user->onTrial()) {  
    // User is within their trial period...  
}
```

如果你想要知道用户是否在“一般”试用期并且还没有创建实际的订阅还可以使用 `onGenericTrial` 方法：

```
if ($user->onGenericTrial()) {  
    // User is within their "generic" trial period...  
}
```

一旦你准备好为用户创建实际的订阅，可以使用 `newSubscription` 方法：

```
$user = User::find(1);  
  
$user->newSubscription('main', 'monthly')->create($creditCardToken);
```

5、处理 Stripe Webhooks

Stripe 和 Braintree 都可以通过 webhooks 通知应用各种事件，要处理 Stripe webhooks，需要定义一个指向 Cashier webhook 控制器的路由，这个控制器将会处理所有输入 webhook 请求并将它们分发到合适的控制器方法：

```
Route::post(  
    'stripe/webhook',  
    '\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook'  
>;
```

注：注册好控制器后，还要在 Stripe 控制面板中配置 webhook URL。

默认情况下，这个控制器将会自动对支付失败次数（这个次数可以在 Stripe 设置中定义）过多的订阅进行取消；此外，我们很快会发现，你可以扩展这个控制器来处理任何你想要处理的 webhook 事件。

Webhooks & CSRF 防护

由于 Stripe webhook 需要绕开 Laravel 的 CSRF 保护，所以需要将其罗列到 `VerifyCsrfToken` 中间件的排除列表或者将其置于 `web` 中间件组之外：

```
protected $except = [  
    'stripe/*',  
];
```

定义 Webhook 事件处理器

Cashier 会基于支付失败次数自动取消订阅，但是如果你想要处理额外的 Stripe webhook 事件，扩展 Webhook 控制器即可。定义的方法名需要与 Cashier 约定的格式保持一致，特别是方法名需要以 `handle` 开头并且是想要处理的 Stripe webhook 的驼峰格式。例如，如果你想要处理 `invoice.payment_succeeded` webhook，则需要添加一个 `handleInvoicePaymentSucceeded` 方法到控制器：

```
<?php  
  
namespace App\Http\Controllers;  
  
use Laravel\Cashier\Http\Controllers\WebhookController as CashierController;  
  
class WebhookController extends CashierController  
{  
    /**
```

```
* Handle a Stripe webhook.  
*  
* @param array $payload  
* @return Response  
*/  
  
public function handleInvoicePaymentSucceeded($payload)  
{  
    // Handle The Event  
}  
}
```

失败的订阅

如果用户的信用卡过期怎么办？不用担心——Cashier webhook 控制器可以轻松为你取消该用户的订阅，正如上面所提到的，你所需要做的只是将路由指向该控制器：

```
Route::post(  
    'stripe/webhook',  
    '\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook'  
>;
```

就是这样，失败的支付将会被控制器捕获和处理，该控制器将会在 Stripe 判断订阅支付失败次数（通常是 3 次）达到上限时取消该用户的订阅。

6、处理 Braintree Webhooks

Stripe 和 Braintree 都可以通过 webhooks 通知应用各种事件，要处理 Braintree webhooks，需

要定义一个指向 Cashier webhook 控制器的路由，这个控制器将会处理所有输入 webhook 请求并将它们分发到合适的控制器方法：

```
Route::post(
    'braintree/webhook',
    '\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook'
);
```

注：注册好控制器后，还要在 Braintree 控制面板中配置 webhook URL。

默认情况下，这个控制器将会自动对支付失败次数（这个次数可以在 Braintree 设置中定义）过多的订阅进行取消；此外，我们很快会发现，你可以扩展这个控制器来处理任何你想要处理的 webhook 事件。

Webhooks & CSRF 防护

由于 Braintree webhook 需要绕开 Laravel 的 CSRF 保护，所以需要将其罗列到 `VerifyCsrfToken` 中间件的排除列表或者将其置于 `web` 中间件组之外：

```
protected $except = [
    'braintree/*',
];
```

定义 Webhook 事件处理器

Cashier 会基于支付失败次数自动取消订阅，但是如果你想要处理额外的 Braintree webhook 事件，扩展 Webhook 控制器即可。定义的方法名需要与 Cashier 约定的格式保持一致，特别是方

方法名需要以 `handle` 开头并且是想要处理的 Braintree webhook 的驼峰格式。例如，如果你想要处理 `dispute_opened` webhook，则需要添加一个 `handleDisputeOpened` 方法到控制器：

```
<?php

namespace App\Http\Controllers;

use Braintree\WebhookNotification;
use Laravel\Cashier\Http\Controllers\WebhookController as CashierController;

class WebhookController extends CashierController
{
    /**
     * Handle a Braintree webhook.
     *
     * @param  WebhookNotification  $webhook
     * @return Response
     */
    public function handleDisputeOpened(WebhookNotification $notification)
    {
        // Handle The Event
    }
}
```

失败的订阅

如果用户的信用卡过期怎么办？不用担心——Cashier webhook 控制器可以轻松为你取消该用户

的订阅，正如上面所提到的，你所需要做的只是将路由指向该控制器：

```
Route::post(
    'braintree/webhook',
    '\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook'
);
```

就是这样，失败的支付将会被控制器捕获和处理，该控制器将会在 Braintree 判断订阅支付失败次数（通常是 3 次）达到上限时取消该用户的订阅。不要忘记在 Braintree 控制面板中配置 webhook URI。

7、一次性支付

基本支付

注：使用 Stripe 时，charge 方法可以接收应用所使用货币对应的最小单位金额，但是使用 Braintree 时，必须传递完整的美元金额到 charge 方法。

如果你想要使用订阅客户的信用卡一次性结清账单，可以使用账单模型实例上的 charge 方法：

```
// Stripe Accepts Charges In Cents...
$user->charge(100);

// Braintree Accepts Charges In Dollars...
$user->charge(1);
```

charge 方法接收一个数组作为第二个参数，允许你传递任何你想要传递的底层 Stripe/Braintree

账单创建参数，创建账单时我们可以参考 Stripe 或者 Braintree [文档](#)提供的可用选项：

```
$user->charge(100, [
    'custom_option' => $value,
]);
```

如果支付失败 `charge` 方法将抛出异常，如果支付成功，该方法会返回完整的 Stripe / Braintree 响应：

```
try {
    $response = $user->charge(100);
} catch (Exception $e) {
    //
}
```

带发票的支付

有时候你需要创建一个一次性支付并且同时生成对应发票以便为用户提供一个 PDF 单据，`invoiceFor` 方法可以帮助我们实现这个需求。例如，让我们为用户的“一次性费用”生成一张 \$5.00 的发票：

```
// Stripe Accepts Charges In Cents...
$user->invoiceFor('One Time Fee', 500);

// Braintree Accepts Charges In Dollars...
```

```
$user->invoiceFor('One Time Fee', 5);
```

该单据会通过用户信用卡立即支付，`invoiceFor` 方法还可以接收一个数组作为第三个参数，从而允许你传递任意你希望的选项到底层 Stripe/Braintree 支付创建：

```
$user->invoiceFor('One Time Fee', 500, [  
    'custom-option' => $value,  
]);
```

注：`invoiceFor` 方法会创建一个对失败支付进行重试的 Stripe 单据，如果你不想要单据重试失败的支付，需要在首次支付失败后使用 Stripe API 关闭它们。

8、发票

你可以使用 `invoices` 方法轻松获取账单模型的发票数组：

```
$invoices = $user->invoices();  
  
// Include pending invoices in the results...  
  
$invoices = $user->invoicesIncludingPending();
```

当列出客户发票时，你可以使用发票的辅助函数来显示相关的发票信息。例如，你可能想要在表格中列出每张发票，从而方便用户下载它们：

```
<table>  
    @foreach ($invoices as $invoice)
```

```
<tr>

<td>{{ $invoice->date()->toFormattedDateString() }}</td>

<td>{{ $invoice->total() }}</td>

<td><a href="/user/invoice/{{ $invoice->id }}">Download</a></td>

</tr>

@endforeach

</table>
```

生成 PDF 发票

在生成 PDF 发票之前，需要安装 PHP 库 `dompdf`：

```
composer require dompdf/dompdf
```

然后，在路由或控制器中，使用 `downloadInvoice` 方法生成发票的 PDF 下载，该方法将会自动

生成相应的 HTTP 响应发送下载到浏览器：

```
use Illuminate\Http\Request;

Route::get('user/invoice/{invoice}', function (Request $request, $invoiceId) {
    return $request->user()->downloadInvoice($invoiceId, [
        'vendor' => 'Your Company',
        'product' => 'Your Product',
    ]);
});
```

14.2 Envoy Task Runner

1、简介

[Laravel Envoy](#) 为运行在远程主机上的通用任务定义提供了一套干净的、最简化的语法。使用 [Blade](#)

风格语法，你可以轻松为开发设置任务，Artisan [命令](#)，以及更多，目前，[Envoy](#) 只支持 Mac 和 Linux 操作系统。

安装

首先，使用 Composer 的 `global` 命令安装 Envoy：

```
composer global require "laravel/envoy=~1.0"
```

由于全局的 Composer 库有时候会导致包版本冲突，所以需要考虑使用 cgr，该命令是 composer global require 命令的替代实现，cgr 库的安装指南可以在 [GitHub](#) 上找到。

注：确保 `~/composer/vendor/bin` 目录已经在系统路径 PATH 中，否则在终端中由于找不到 `envoy` 而无法执行该命令。

更新 Envoy

还可以使用 Composer 保持安装的 Envoy 是最新版本，因为 `composer global update` 命令将更新所有全局安装的 Composer 包：

```
composer global update
```

2、编写任务

所有的 Envoy 任务都定义在项目根目录下的 `Envoy.blade.php` 文件中，下面是一个让你上手的示例：

```
@servers(['web' => 'user@192.168.1.1'])

@task('foo', ['on' => 'web'])
    ls -la
@endtask
```

正如你所看到的，`@servers` 数组定义在文件顶部，从而允许你在任务声明中使用 `on` 选项引用这些服务器，在 `@task` 声明中，应该放置将要在服务器上运行的 Bash 代码。

你可以通过指定服务器 IP 地址为 127.0.0.1 强制脚本在本地运行：

```
@servers(['localhost' => '127.0.0.1'])
```

起步

有时候，你需要在执行 Envoy 任务之前执行一些 PHP 代码，可以在 Envoy 文件中使用 `@setup` 指令来声明变量和要执行的 PHP 代码：

```
@setup
$now = new DateTime();
$environment = isset($env) ? $env : "testing";
@endsetup
```

如果你需要在任务执行之前引入其它 PHP 文件，可以在 `Envoy.blade.php` 文件顶部使用 `@include` 来引入：

```
@include('vendor/autoload.php')

@task('foo')
# ...
@endtask
```

变量

如果需要的话，你可以使用命令行传递选项值到 Envoy 任务：

```
envoy run deploy --branch=master
```

你可以在任务中通过 Blade 的“echo”语法访问该选项，当然，你还可以使用 `if` 语句并在任务中循环，例如，下面我们在执行 `git pull` 命令前验证 `$branch` 变量是否存在：

```
@servers(['web' => '192.168.1.1'])

@task('deploy', ['on' => 'web'])
    cd site
    @if ($branch)
        git pull origin {{ $branch }}
    @endif
    php artisan migrate
@endtask
```

Story

`story` 通过一个便捷的名字对任务集合进行分组，从而允许你将小而专注的任务组合成大的任务，例如，`deploy` story 将会通过在其定义中罗列任务名的方式运行 `git` 和 `composer` 任务：

```
@servers(['web' => '192.168.1.1'])

@story('deploy')
    git
    composer
```

```
@endstory

@task('git')
    git pull origin master
@endtask

@task('composer')
    composer install
@endtask
```

story 被编写好之后，就可以像普通任务一样运行：

```
envoy run deploy
```

多个服务器

你可以轻松地在多主机上运行同一个任务，首先，添加额外服务器到 `@servers` 声明，每个服务器应该被指配一个唯一的名字。定义好服务器后，在任务的 `on` 数组中列出所有服务器即可：

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2']])
    cd site
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

平行运行

默认情况下，该任务将会依次在每个服务器上执行，这意味着，该任务在第一台服务器上运行完成后才会开始在第二台服务器运行。如果你想要在多个服务器上平行运行，添加 `parallel` 选项到任务声明：

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2'], 'parallel' => true])
    cd site
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

3、运行任务

要运行一个在 `Envoy.blade.php` 文件中定义的任务，需要执行 Envoy 的 `run` 命令，然后传递你要执行的任务或 `story` 的名字。Envoy 将会运行命令并从服务打印输出：

```
envoy run task
```

确认任务执行

如果你想要在服务器上运行给定任务之前弹出弹出提示进行确认，可以在任务声明中使用 `confirm` 指令：

```
@task('deploy', ['on' => 'web', 'confirm' => true])
```

```
cd site  
git pull origin {{ $branch }}  
php artisan migrate  
@endtask
```

4、通知

Slack

Envoy 还支持在任务执行之后发送通知到 [Slack](#)。`@slack` 指令接收一个 Slack 钩子 URL 和频道名称，你可以通过在 Slack 控制面板中创建“Incoming WebHooks”集成来获取 webhook URL，需要传递完整的 webhook URL 到 `@slack` 指令：

```
@after  
  @slack('webhook-url', '#bots')  
@endafter
```

你可以提供下面两种其中之一作为频道参数：

- 发送通知到频道: `#channel`
- 发送通知到用户: `@user`

14.3 Laravel Scout

1、简介

[Laravel Scout](#) 为 [Eloquent 模型](#) 全文[搜索](#)实现提供了简单的、基于驱动的解决方案，通过使用模型观察者，Scout 会自动同步更新模型记录的[索引](#)。

目前，Scout 通过 [Algolia](#) 驱动提供搜索功能，不过，编写自定义驱动很简单，你可以很轻松地通过自己的搜索实现来扩展 Scout。

2、安装

首先，我们通过 Composer 包管理器来安装 Scout：

```
composer require laravel/scout
```

接下来，需要添加 `ScoutServiceProvider` 到配置文件 `config/app.php` 的 `providers` 数组：

```
Laravel\Scout\ScoutServiceProvider::class,
```

注册 Scout 服务提供者之后，还需要通过 Artisan 命令 `vendor:publish` 发布 Scout 配置，该命令会发布配置文件 `scout.php` 到 `config` 目录：

```
php artisan vendor:publish --provider="Laravel\Scout\ScoutServiceProvider"
```

最后，如果你想要模型变得可搜索，需要添加 `Laravel\Scout\Searchable` trait 到模型类，该 trait 会注册模型观察者来保持搜索驱动与模型记录数据的一致性：

```
<?php  
  
namespace App;
```

```
use Laravel\Scout\Searchable;  
use Illuminate\Database\Eloquent\Model;  
  
class Post extends Model  
{  
    use Searchable;  
}
```

队列

尽管不强制使用 Scout，不过在使用这个库之前强烈建议考虑配置一个队列驱动。运行一个队列进程将允许 Scout 把所有同步模型信息到搜索索引的操作推送到队列中，从而为应用的 web 界面提供更快的响应时间。

配置好队列驱动后，在配置文件 `config/scout.php` 中设置 `queue` 选项的值为 `true`：

```
'queue' => true,
```

驱动预备知识

Algolia

使用 Algolia 驱动的话，需要在配置文件 `config/scout.php` 中设置 Algolia 的 `id` 和 `secret` 信息。

配置好之后，还需要通过 Composer 包管理器安装 Algolia PHP SDK：

```
composer require algolia/algoliasearch-client-php
```

3、配置

配置模型索引

每个 Eloquent 模型都是通过给定的搜索“索引”进行同步，该索引包含了所有可搜索的模型记录，换句话说，你可以将索引看作是一个 MySQL 数据表。默认情况下，每个模型都会被持久化到与模型对应表名（通常是模型名称的复数形式）相匹配的索引中，不过，你可以通过重写模型中的 `searchableAs` 方法来覆盖这一默认设置：

```
<?php

namespace App;

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

class Post extends Model
{
    use Searchable;

    /**
     * 获取模型的索引名称.
     *
     * @return string
     */
    public function searchableAs()
```

```
{  
    return 'posts_index';  
}  
}
```

配置搜索数据

默认情况下，模型以完整的 `toArray` 格式持久化到搜索索引，如果你想要自定义被持久化到搜索索引的数据，可以重写模型上的 `toSearchableArray` 方法：

```
<?php  
  
namespace App;  
  
use Laravel\Scout\Searchable;  
use Illuminate\Database\Eloquent\Model;  
  
class Post extends Model  
{  
    use Searchable;  
  
    /**  
     * 获取模型的索引数据数组  
     *  
     * @return array  
     */  
    public function toSearchableArray()  
}
```

```
{  
    $array = $this->toArray();  
  
    // 自定义数组...  
  
    return $array;  
}  
}
```

4、索引

批量导入

如果你想要安装 Scout 到已存在的项目，你可能已经有了想要导入搜索驱动的数据库记录，Scout 提供了 Artisan 命令 `import` 用于导入所有已存在的数据到搜索索引：

```
php artisan scout:import "App\Post"
```

添加记录

添加 `Laravel\Scout\Searchable` trait 到模型之后，剩下需要做的就是保存模型实例，然后该实例会自动被添加到模型索引，如果你配置了 Scout 使用队列，该操作会被推送到队列在后台执行：

```
$order = new App\Order;  
  
// ...  
  
$order->save();
```

通过查询添加

如果你想要通过 Eloquent 查询添加模型集合到搜索索引，可以在 Eloquent 查询之后追加

`searchable` 方法调用。`searchable` 方法会分组块进行查询并将结果添加到搜索索引。再次强调，

如果你配置了 Scout 使用队列，所有的组块查询会被推送到队列在后台进行：

```
// 通过 Eloquent 查询添加...
App\Order::where('price', '>', 100)->searchable();

// 还可以通过关联关系添加记录...
$user->orders()->searchable();

// 还可以通过集合添加记录...
$orders->searchable();
```

`searchable` 方法还可以进行“upsort”操作，换句话说，如果模型记录已经存在于索引，则会被更新，如果不存在，则会被添加。

更新记录

要更新可搜索的模型，需要更新模型实例的属性并保存模型到数据库。Scout 会自动持久化更新到搜索索引：

```
$order = App\Order::find(1);

// 更新订单...
```

```
$order->save();
```

还可以使用模型查询提供的 `searchable` 方法更新模型集合，如果模型在搜索索引中不存在，则会被创建：

```
// 通过 Eloquent 查询更新...
App\Order::where('price', '>', 100)->searchable();

// 还可以通过关联关系更新...
$user->orders()->searchable();

// 还可以通过集合更新...
$orders->searchable();
```

移除记录

要从索引中移除记录，只需从数据库中删除记录即可，这种移除方式甚至兼容[软删除](#)模型：

```
$order = App\Order::find(1);

$order->delete();
```

如果你在删除记录前不想获取模型，可以使用模型查询实例或集合上的 `unsearchable` 方法：

```
// 通过 Eloquent 查询移除...
App\Order::where('price', '>', 100)->unsearchable();
```

```
// 还可以通过关联关系移除...
$user->orders()->unsearchable();

// 还可以通过集合移除...
$orders->unsearchable();
```

暂停索引

有时候你需要在不同步模型数据到搜索索引的情况下执行批量的 Eloquent 操作，可以通过 `withoutSyncingToSearch` 方法来实现。该方法接收一个立即被执行的回调，该回调中出现的所有模型操作都不会同步到搜索索引：

```
App\Order::withoutSyncingToSearch(function () {
    // Perform model actions...
});
```

5、搜索

你可以通过 `search` 方法来搜索一个模型，该方法接收一个用于搜索模型的字符串，然后你还需要在这个搜索查询上调用一个 `get` 方法来获取与给定搜索查询相匹配的 Eloquent 模型：

```
$orders = App\Order::search('Star Trek')->get();
```

由于 Scout 搜索返回的是 Eloquent 模型集合，你甚至可以直接从路由或控制器中返回结果，它们将会被自动转换为 JSON 格式：

```
use Illuminate\Http\Request;

Route::get('/search', function (Request $request) {
    return App\Order::search($request->search)->get();
});
```

where 子句

Scout 允许你添加简单的 where 子句到搜索查询，目前，这些子句仅支持简单的数值相等检查，由于搜索索引不是关系型数据库，更多高级的 where 子句暂不支持：

```
$orders = App\Order::search('Star Trek')->where('user_id', 1)->get();
```

分页

除了获取模型集合之外，还可以使用 `paginate` 方法对搜索结果进行分页，该方法返回一个 `Paginator` 实例 —— 就像你[对传统 Eloquent 查询进行分页](#)一样：

```
$orders = App\Order::search('Star Trek')->paginate();
```

你可以通过传入数量作为 `paginate` 方法的第一个参数来指定每页显示多少个模型：

```
$orders = App\Order::search('Star Trek')->paginate(15);
```

获取结果之后，可以使用 `Blade` 显示结果并渲染分页链接，就像对传统 Eloquent 查询进行分页时

一样：

```
<div class="container">  
    @foreach ($orders as $order)  
        {{ $order->price }}  
    @endforeach  
</div>  
  
{{ $orders->links() }}
```

6、自定义引擎

编写引擎

如果某个内置的 Scout [搜索引擎](#)不满足你的需求，可以编写自定义的引擎并将其注册到 Scout，

自定义的引擎需要继承自抽象类 `Laravel\Scout\Engines\Engine`，该抽象类包含了 5 个自定义引擎必须实现的方法：

```
use Laravel\Scout\Builder;  
  
abstract public function update($models);  
abstract public function delete($models);  
abstract public function search(Builder $builder);  
abstract public function paginate(Builder $builder, $perPage, $page);  
abstract public function map($results, $model);
```

这 5 个方法的实现可以参考 `Laravel\Scout\Engines\AlgoliaEngine` 类，这个类为我们学习如何在自定义引擎中实现这些方法提供了最佳范本。

注册引擎

编写好自定义引擎之后，可以通过 Scout 引擎管理器提供的 `extend` 方法将其注册到 Scout。你需要在 `AppServiceProvider`（或者其他服务提供者）的 `boot` 方法中调用这个 `extend` 方法。例如，如果你编写了 `MySqlSearchEngine`，可以这样注册：

```
use Laravel\Scout\EngineManager;

/**
 * 启动任意应用服务.
 *
 * @return void
 */
public function boot()
{
    resolve(EngineManager::class)->extend('mysql', function () {
        return new MySqlSearchEngine;
    });
}
```

引擎被注册之后，可以在配置文件 `config/scout.php` 中将其设置为 Scout 默认的驱动：

```
'driver' => 'mysql',
```

14.4 Laravel Socialite

1、简介

除了传统的基于表格的登录认证，[Laravel](#) 还使用 [Laravel Socialite](#) 提供了一个优雅的流式接口，用于通过 Facebook、Twitter、Google、LinkedIn、GitHub 和 Bitbucket 实现 [OAuth](#) 认证。它几乎可以处理所有你恐惧编写的社会化登录认证代码。

我们不接受新的适配器。

社区驱动的其他平台的适配器罗列在 [Socialite 提供者](#) 网站上。

要使用社会化登录，需要在 `composer.json` 文件中添加依赖：

```
composer require laravel/socialite
```

之后运行 `composer update` 安装依赖。

2、配置

安装完社会化登录库后，在配置文件 `config/app.php` 中注册 [Laravel\Socialite\SocialiteServiceProvider](#)：

```
'providers' => [
    // 其它服务提供者...
    Laravel\Socialite\SocialiteServiceProvider::class,
],
```

还要在 `app` 配置文件中添加 `Socialite` 门面到 `aliases` 数组：

```
'Socialite' => Laravel\Socialite\Facades\Socialite::class,
```

你还需要为应用使用的 OAuth 服务添加认证信息，这些认证信息位于配置文

件 `config/services.php` , 而且对应 key 为 `facebook`、`twitter`、`linkedin`、`google`、`github` 或 `bitbucket` , 配置哪些 key 取决于应用需要的提供者。例如 :

```
'github' => [
    'client_id' => 'your-github-app-id',
    'client_secret' => 'your-github-app-secret',
    'redirect' => 'http://your-callback-url',
],
```

3、基本使用

接下来 , 准备好认证[用户](#) ! 你需要两个路由 : 一个用于重定向用户到 OAuth 提供者 , 另一个用户获取认证后来自提供者的回调。我们使用 `Socialite` 门面访问 Socialite :

```
<?php

namespace App\Http\Controllers\Auth;

use Socialite;

class AuthController extends Controller
{
    /**
     * 将用户重定向到 Github 认证页面
     *
     * @return Response
     */
    public function redirectToProvider()
```

```
{  
    return Socialite::driver('github')->redirect();  
  
}  
  
/**  
 * 从 Github 获取用户信息.  
 *  
 * @return Response  
 */  
  
public function handleProviderCallback()  
{  
    $user = Socialite::driver('github')->user();  
  
    // $user->token;  
}  
}
```

`redirect` 方法将用户发送到 OAuth 提供者，`user` 方法读取请求信息并从提供者中获取用户信息，在重定向用户之前，你还可以在请求上使用 `scope` 方法设置“作用域”，该方法将会重写已存在的所有作用域：

```
return Socialite::driver('github')  
    ->scopes(['scope1', 'scope2'])->redirect();
```

当然，你需要定义路由到控制器方法：

```
Route::get('auth/github', 'Auth\AuthController@redirectToProvider');  
Route::get('auth/github/callback', 'Auth\AuthController@handleProviderCallback');
```

很多 OAuth 提供者在重定向请求中支持可选参数，要在请求中包含可选参数，可以通过一个关联

数组调用 `with` 方法：

```
return Socialite::driver('google')
    ->with(['hd' => 'example.com'])->redirect();
```

使用 `with` 方法的时候，注意不要传递保留关键字作为数组的 key，例如 `state` 或 `response_type`。

获取用户信息

有了用户实例之后，就可以获取更多用户详情：

```
$user = Socialite::driver('github')->user();

// OAuth Two Providers
$token = $user->token;
$refreshToken = $user->refreshToken; // not always provided
$expiresIn = $user->expiresIn;

// OAuth One Providers
$token = $user->token;
$tokenSecret = $user->tokenSecret;

// All Providers
$user->getId();
$user->getNickname();
$user->getName();
$user->getEmail();
$user->getAvatar();
```

15. 附录

15.1 集合

1、简介

`Illuminate\Support\Collection` 类为处理数组数据提供了平滑、方便的封装。例如，查看下面的代码，我们使用辅助函数 `collect` 创建一个新的集合实例，为每一个元素运行 `strtoupper` 函数，然后移除所有空元素：

```
$collection = collect(['taylor', 'abigail', null])->map(function ($name) {
    return strtoupper($name);
})->reject(function ($name) {
    return empty($name);
});
```

正如你所看到的，`Collection` 类允许你使用方法链对底层数组执行匹配和移除操作，通常，每一个 `Collection` 方法都会返回一个新的 `Collection` 实例。

2、创建集合

正如上面所提到的，辅助函数 `collect` 为给定数组返回一个新的 `Illuminate\Support\Collection` 实例，所以，创建集合很简单：

```
$collection = collect([1, 2, 3]);
```

注：默认情况下，[Eloquent 模型](#)的集合总是返回 [Collection](#) 实例。

3、集合方法

[本文档](#)接下来的部分将会讨论 [Collection](#) 类上每一个有效的方法，所有这些方法都可以以方法链的方式流式操作底层数组。此外，几乎每个方法返回一个新的 [Collection](#) 实例，允许你在必要的时候保持原来的集合备份。

方法列表

all()

[all](#) 方法简单返回集合表示的底层数组：

```
collect([1, 2, 3])->all();  
// [1, 2, 3]
```

avg()

[avg](#) 方法返回所有集合项的平均值：

```
collect([1, 2, 3, 4, 5])->avg();  
// 3
```

如果集合包含嵌套的数组或对象，需要指定要使用的键以判定计算哪些值的平均值：

```
$collection = collect([  
    ['name' => 'JavaScript: The Good Parts', 'pages' => 176],  
    ['name' => 'JavaScript: The Definitive Guide', 'pages' => 1096],  
    ...
```

```
]);  
  
$collection->avg('pages');  
// 636
```

chunk()

chunk 方法将一个集合分割成多个小尺寸的小集合：

```
$collection = collect([1, 2, 3, 4, 5, 6, 7]);  
  
$chunks = $collection->chunk(4);  
  
$chunks->toArray();  
  
// [[1, 2, 3, 4], [5, 6, 7]]
```

当处理栅栏系统如 [Bootstrap](#) 时该方法在[视图](#)中尤其有用，建设你有一个想要显示在栅栏中的 [Eloquent](#) 模型集合：

```
@foreach ($products->chunk(3) as $chunk)  
  
    <div class="row">  
  
        @foreach ($chunk as $product)  
  
            <div class="col-xs-4">{{ $product->name }}</div>  
  
        @endforeach  
  
    </div>  
  
@endforeach
```

collapse()

collapse 方法将一个多维数组集合收缩成一个一维数组：

```
$collection = collect([[1, 2, 3], [4, 5, 6], [7, 8, 9]]);

$collapsed = $collection->collapse();

$collapsed->all();

// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

combine()

`combine` 方法可以将一个集合的键和另一个数组或集合的值连接起来：

```
$collection = collect(['name', 'age']);

$combined = $collection->combine(['George', 29]);

$combined->all();

// ['name' => 'George', 'age' => 29]
```

contains()

`contains` 方法判断集合是否包含一个给定项：

```
$collection = collect(['name' => 'Desk', 'price' => 100]);

$collection->contains('Desk');

// true

$collection->contains('New York');

// false
```

你还可以传递一个键值对到 `contains` 方法，这将会判断给定键值对是否存在于集合中：

```
$collection = collect([
```

```
[ 'product' => 'Desk', 'price' => 200],  
[ 'product' => 'Chair', 'price' => 100],  
]);  
  
$collection->contains('product', 'Bookcase');  
// false
```

最后，你还可以传递一个回调到 `contains` 方法来执行自己的真实测试：

```
$collection = collect([1, 2, 3, 4, 5]);  
$collection->contains(function ($key, $value) {  
    return $value > 5;  
});  
// false
```

count()

`count` 方法返回集合中所有项的总数：

```
$collection = collect([1, 2, 3, 4]);  
$collection->count();  
// 4
```

diff()

`diff` 方法将集合和另一个集合或原生 PHP 数组以基于值的方式作比较，这个方法会返回存在于原来集合而不存在于给定集合的值：

```
$collection = collect([1, 2, 3, 4, 5]);
```

```
$diff = $collection->diff([2, 4, 6, 8]);  
$diff->all();  
// [1, 3, 5]
```

diffKeys()

diffKeys 方法会基于键对集合和另一个集合或原生 PHP 数组进行比较。该方法会返回存在于原来集合而不存在于给定集合的键值对：

```
$collection = collect([  
    'one' => 10,  
    'two' => 20,  
    'three' => 30,  
    'four' => 40,  
    'five' => 50,  
]);  
  
$diff = $collection->diffKeys([  
    'two' => 2,  
    'four' => 4,  
    'six' => 6,  
    'eight' => 8,  
]);  
  
$diff->all();  
// ['one' => 10, 'three' => 30, 'five' => 50]
```

each()

each 方法迭代集合中的数据项并传递每个数据项到给定回调：

```
$collection = $collection->each(function ($item, $key) {  
    //  
});
```

如果你想要终止对数据项的迭代，可以从回调返回 `false`：

```
$collection = $collection->each(function ($item, $key) {  
    if /* some condition */ {  
        return false;  
    }  
});
```

every()

`every` 方法创建一个包含数组第 `n-th` 个元素的新集合：

```
$collection = collect(['a', 'b', 'c', 'd', 'e', 'f']);  
  
$collection->every(4);  
// ['a', 'e']
```

还可以选择指定从第几个元素开始：

```
$collection->every(4, 1);  
// ['b', 'f']
```

except()

`except` 方法返回集合中除了指定键的所有集合项：

```
$collection = collect(['product_id' => 1, 'name' => 'Desk', 'price'  
=> 100, 'discount' => false]);  
  
$filtered = $collection->except(['price', 'discount']);
```

```
$filtered->all();
// ['product_id' => 1, 'name' => 'Desk']
```

与 `except` 相对的是 `only` 方法。

filter()

`filter` 方法通过给定回调过滤集合，只有通过给定测试的数据项才会保留下：

```
$collection = collect([1, 2, 3, 4]);

$filtered = $collection->filter(function ($item) {
    return $item > 2;
});

$filtered->all();
// [3, 4]
```

和 `filter` 相对的方法是 `reject`。

first()

`first` 方法返回通过测试集合的第一个元素：

```
collect([1, 2, 3, 4])->first(function ($key, $value) {
    return $value > 2;
});
// 3
```

你还可以调用不带参数的 `first` 方法来获取集合的第一个元素，如果集合是空的，返回 `null`：

```
collect([1, 2, 3, 4])->first();
```

```
// 1
```

flatMap()

flatMap 方法会迭代集合并传递每个值到给定回调，该回调可以自由编辑数据项并将其返回，最后形成一个经过编辑的新集合。然后，这个数组通过层级被扁平化：

```
$collection = collect([
    ['name' => 'Sally'],
    ['school' => 'Arkansas'],
    ['age' => 28]
]);

$flattened = $collection->flatMap(function ($values) {
    return array_map('strtoupper', $values);
});

$flattened->all();
// ['name' => 'SALLY', 'school' => 'ARKANSAS', 'age' => '28'];
```

flatten()

flatten 方法将多维度的集合变成一维的：

```
$collection = collect(['name' => 'taylor', 'languages' => ['php', 'javascript']]);
$flattened = $collection->flatten();
$flattened->all();
// ['taylor', 'php', 'javascript'];
```

还可以选择性传入深度参数：

```
$collection = collect([
    'Apple' => [
        ['name' => 'iPhone 6S', 'brand' => 'Apple'],
    ],
    'Samsung' => [
        ['name' => 'Galaxy S7', 'brand' => 'Samsung']
    ],
]);
```



```
$products = $collection->flatten(1);
```



```
$products->values()->all();
```



```
/*
[
    ['name' => 'iPhone 6S', 'brand' => 'Apple'],
    ['name' => 'Galaxy S7', 'brand' => 'Samsung'],
]
```

```
*/
```

在本例中，调用不提供深度的 `flatten` 方法也会对嵌套数组进行扁平化处理，返回结果是 `['iPhone 6S', 'Apple', 'Galaxy S7', 'Samsung']`。提供深度允许你严格设置被扁平化的数组层级。

flip()

flip 方法将集合的键值做交换：

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);  
$flipped = $collection->flip();  
$flipped->all();  
// ['taylor' => 'name', 'laravel' => 'framework']
```

forget()

forget 方法通过键从集合中移除数据项：

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);  
$collection->forget('name');  
$collection->all();  
// ['framework' => 'laravel']
```

注：不同于大多数的集合方法，**forget** 不返回新的修改过的集合；它只修改所调用的集合。

forPage()

forPage 方法返回新的包含给定页数数据项的集合。该方法接收页码数作为第一个参数，每页显示数据项数作为第二个参数：

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9])->forPage(2, 3);  
$collection->all(); // [4, 5, 6]
```

get()

get 方法返回给定键的数据项，如果键不存在，返回 **null**：

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);  
  
$value = $collection->get('name');  
  
// taylor
```

你可以选择传递默认值作为第二个参数：

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);  
  
$value = $collection->get('foo', 'default-value');  
  
// default-value
```

你甚至可以传递回调作为默认值，如果给定键不存在的话回调的结果将会返回：

```
$collection->get('email', function () {  
  
    return 'default-value';});  
  
// default-value
```

groupBy()

groupBy 方法通过给定键分组集合数据项：

```
$collection = collect([  
  
    ['account_id' => 'account-x10', 'product' => 'Chair'],  
  
    ['account_id' => 'account-x10', 'product' => 'Bookcase'],  
  
    ['account_id' => 'account-x11', 'product' => 'Desk'],  
  
]);  
  
  
$grouped = $collection->groupBy('account_id');
```

```
$grouped->toArray();  
  
/*  
[  
    'account-x10' => [  
        ['account_id' => 'account-x10', 'product' => 'Chair'],  
        ['account_id' => 'account-x10', 'product' => 'Bookcase'],  
    ],  
    'account-x11' => [  
        ['account_id' => 'account-x11', 'product' => 'Desk'],  
    ],  
]  
*/
```

除了传递字符串 key，还可以传递一个回调，回调应该返回分组后的值：

```
$grouped = $collection->groupBy(function ($item, $key) {  
    return substr($item['account_id'], -3);  
});  
  
$grouped->toArray();  
  
/*  
[  
    'x10' => [  
        ['account_id' => 'account-x10', 'product' => 'Chair'],  
        ['account_id' => 'account-x10', 'product' => 'Bookcase'],  
    ],  
]
```

```
'x11' => [
    ['account_id' => 'account-x11', 'product' => 'Desk'],
],
]
*/
```

has()

`has` 方法判断给定键是否在集合中存在：

```
$collection = collect(['account_id' => 1, 'product' => 'Desk']);
$collection->has('email');
// false
```

implode()

`implode` 方法连接集合中的数据项。其参数取决于集合中数据项的类型。如果集合包含数组或对象，应该传递你想要连接的属性键，以及你想要放在值之间的“粘合”字符串：

```
$collection = collect([
    ['account_id' => 1, 'product' => 'Desk'],
    ['account_id' => 2, 'product' => 'Chair'],
]);
$collection->implode('product', ', ');
// Desk, Chair
```

如果集合包含简单的字符串或数值，只需要传递“粘合”字符串作为唯一参数到该方法：

```
collect([1, 2, 3, 4, 5])->implode('-');
// '1-2-3-4-5'
```

intersect()

`intersect` 方法返回两个集合的交集，结果集合将保留原来集合的键：

```
$collection = collect(['Desk', 'Sofa', 'Chair']);
$intersect = $collection->intersect(['Desk', 'Chair', 'Bookcase']);
```

```
$intersect->all();  
// [0 => 'Desk', 2 => 'Chair']
```

isEmpty()

如果集合为空的话 `isEmpty` 方法返回 `true`；否则返回 `false`：

```
collect([])->isEmpty();  
// true
```

keyBy()

`keyBy` 方法将指定键的值作为集合的键，如果多个数据项拥有同一个键，只有最后一个会出现在

新集合里面：

```
$collection = collect([  
    ['product_id' => 'prod-100', 'name' => 'desk'],  
    ['product_id' => 'prod-200', 'name' => 'chair'],  
]);  
  
$keyed = $collection->keyBy('product_id');  
  
$keyed->all();  
  
/*  
[  
    'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],  
    'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],  
]
```

你还可以传递自己的回调到该方法，该回调将会返回经过处理的键的值作为新的集合键：

```
$keyed = $collection->keyBy(function ($item) {
    return strtoupper($item['product_id']);
});

$keyed->all();

/*
[
    'PROD-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'PROD-200' => ['product_id' => 'prod-200', 'name' => 'Chair']
]
*/
```

keys()

`keys` 方法返回所有集合的键：

```
$collection = collect([
    'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$keys = $collection->keys();

$keys->all();
// ['prod-100', 'prod-200']
```

last()

`last` 方法返回通过测试的集合的最后一个元素：

```
collect([1, 2, 3, 4])->last(function ($key, $value) {
    return $value < 3;
});
// 2
```

还可以调用无参的 `last` 方法来获取集合的最后一个元素。如果集合为空，返回 `null`：

```
collect([1, 2, 3, 4])->last();
// 4
```

map()

`map` 方法遍历集合并传递每个值给给定回调。该回调可以修改数据项并返回，从而生成一个新的
经过修改的集合：

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$multiplied = $collection->map(function ($item, $key) {  
    return $item * 2;  
});  
  
$multiplied->all();  
// [2, 4, 6, 8, 10]
```

注：和大多数集合方法一样，`map` 返回新的集合实例；它并不修改所调用的实例。如果你想要改
变原来的集合，使用 `transform` 方法。

mapWithKeys()

`mapWithKeys` 方法对集合进行迭代并传递每个值到给定回调，该回调会返回包含键值对的关联数
组：

```
$collection = collect([  
    [  
        'name' => 'John',  
        'department' => 'Sales',  
        'email' => 'john@example.com'  
    ],  
    [  
        'name' => 'Jane',  
        'department' => 'Marketing',  
        'email' => 'jane@example.com'  
    ]  
])
```

```
]);  
  
$keyed = $collection->mapWithKeys(function ($item) {  
    return [$item['email'] => $item['name']];  
});  
  
$keyed->all();  
  
/*  
[  
    'john@example.com' => 'John',  
    'jane@example.com' => 'Jane',  
]  
*/
```

max()

max 方法返回集合中给定键的最大值：

```
$max = collect(['foo' => 10], ['foo' => 20])->max('foo');  
// 20  
$max = collect([1, 2, 3, 4, 5])->max();  
// 5
```

merge()

merge 方法合并给定数组到集合。该数组中的任何字符串键匹配集合中的字符串键的将会重写集合中的值：

```
$collection = collect(['product_id' => 1, 'name' => 'Desk']);  
$merged = $collection->merge(['price' => 100, 'discount' => false]);  
$merged->all();  
// ['product_id' => 1, 'name' => 'Desk', 'price' => 100, 'discount' => false]
```

如果给定数组的键是数字，数组的值将会附加到集合后面：

```
$collection = collect(['Desk', 'Chair']);
$merged = $collection->merge(['Bookcase', 'Door']);
$merged->all();
// ['Desk', 'Chair', 'Bookcase', 'Door']
```

min()

`min` 方法返回集合中给定键的最小值：

```
$min = collect([('foo' => 10], ['foo' => 20]))->min('foo');
// 10
$min = collect([1, 2, 3, 4, 5])->min();
// 1
```

only()

`only` 方法返回集合中指定键的集合项：

```
$collection = collect(['product_id' => 1, 'name' => 'Desk', 'price' => 100, 'discount' => false]);

$filtered = $collection->only(['product_id', 'name']);

$filtered->all();
// ['product_id' => 1, 'name' => 'Desk']
```

与 `only` 方法相对的是 `except` 方法。

pipe()

`pipe` 方法传递集合到给定回调并返回结果：

```
$collection = collect([1, 2, 3]);

$piped = $collection->pipe(function ($collection) {
    return $collection->sum();
});
```

```
// 6
```

pluck()

`pluck` 方法为给定键获取所有集合值：

```
$collection = collect([
    ['product_id' => 'prod-100', 'name' => 'Desk'],
    ['product_id' => 'prod-200', 'name' => 'Chair'],
]);
$plucked = $collection->pluck('name');
$plucked->all();
// ['Desk', 'Chair']
```

还可以指定你想要结果集合如何设置键：

```
$plucked = $collection->pluck('name', 'product_id');
$plucked->all();
// ['prod-100' => 'Desk', 'prod-200' => 'Chair']
```

pop()

`pop` 方法移除并返回集合中最后面的数据项：

```
$collection = collect([1, 2, 3, 4, 5]);
$collection->pop();
// 5
$collection->all();
// [1, 2, 3, 4]
```

prepend()

`prepend` 方法添加数据项到集合开头：

```
$collection = collect([1, 2, 3, 4, 5]);
$collection->prepend(0);
$collection->all();
// [0, 1, 2, 3, 4, 5]
```

你还可以传递第二个参数到该方法用于设置前置项的键：

```
$collection = collect(['one' => 1, 'two', => 2]);  
  
$collection->prepend(0, 'zero');  
  
$collection->all();  
// ['zero' => 0, 'one' => 1, 'two', => 2]
```

pull()

`pull` 方法通过键从集合中移除并返回数据项：

```
$collection = collect(['product_id' => 'prod-100', 'name' => 'Desk']);  
$collection->pull('name');  
// 'Desk'  
$collection->all();  
// ['product_id' => 'prod-100']
```

push()

`push` 方法附加数据项到集合结尾：

```
$collection = collect([1, 2, 3, 4]);  
$collection->push(5);  
$collection->all();  
// [1, 2, 3, 4, 5]
```

put()

`put` 方法在集合中设置给定键和值：

```
$collection = collect(['product_id' => 1, 'name' => 'Desk']);  
$collection->put('price', 100);  
$collection->all();  
// ['product_id' => 1, 'name' => 'Desk', 'price' => 100]
```

random()

`random` 方法从集合中返回随机数据项：

```
$collection = collect([1, 2, 3, 4, 5]);  
$collection->random();  
// 4 - (retrieved randomly)
```

你可以传递一个整型数据到 `random` 函数，如果该整型数值大于 1，将会返回一个集合：

```
$random = $collection->random(3);
$random->all();
// [2, 4, 5] - (retrieved randomly)
```

reduce()

`reduce` 方法用于减少集合到单个值，传递每个迭代结果到子迭代：

```
$collection = collect([1, 2, 3]);
$total = $collection->reduce(function ($carry, $item) {
    return $carry + $item;
});
// 6
```

在第一次迭代时 `$carry` 的值是 `null`；不过，你可以通过传递第二个参数到 `reduce` 来指定其初始值：

```
$collection->reduce(function ($carry, $item) {
    return $carry + $item;
}, 4);
// 10
```

reject()

`reject` 方法使用给定回调过滤集合，该回调应该为所有它想要从结果集合中移除的数据项返回

`true`：

```
$collection = collect([1, 2, 3, 4]);
$filtered = $collection->reject(function ($value, $key) {
    return $value > 2;
});

$filtered->all();
// [1, 2]
```

和 `reduce` 方法相对的方法是 `filter` 方法。

reverse()

`reverse` 方法将集合数据项的顺序颠倒：

```
$collection = collect([1, 2, 3, 4, 5]);
$reversed = $collection->reverse();
$reversed->all();
// [5, 4, 3, 2, 1]
```

search()

`search` 方法为给定值查询集合，如果找到的话返回对应的键，如果没找到，则返回 `false`：

```
$collection = collect([2, 4, 6, 8]);
$collection->search(4);
// 1
```

上面的搜索使用的是松散比较，要使用严格比较，传递 `true` 作为第二个参数到该方法：

```
$collection->search('4', true);
// false
```

此外，你还可以传递自己的回调来搜索通过测试的第一个数据项：

```
$collection->search(function ($item, $key) {
    return $item > 5;
});
// 2
```

shift()

`shift` 方法从集合中移除并返回第一个数据项：

```
$collection = collect([1, 2, 3, 4, 5]);
$collection->shift();
// 1
$collection->all();
// [2, 3, 4, 5]
```

shuffle()

`shuffle` 方法随机打乱集合中的数据项：

```
$collection = collect([1, 2, 3, 4, 5]);
$shuffled = $collection->shuffle();
$shuffled->all();
// [3, 2, 5, 1, 4] // (generated randomly)
```

slice()

`slice` 方法从给定索开始返回集合的一个切片：

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
$slice = $collection->slice(4);
$slice->all();
// [5, 6, 7, 8, 9, 10]
```

如果你想要限制返回切片的尺寸，将尺寸值作为第二个参数传递到该方法：

```
$slice = $collection->slice(4, 2);
$slice->all();
// [5, 6]
```

返回的切片有新的、数字化索引的键，如果你想要保持原有的键，可以使用 `values` 方法对它们进行重新索引。

sort()

`sort` 方法对集合进行排序，排序后的集合保持原来的数组键，在本例中我们使用 `values` 方法重置键为连续编号索引：

```
$collection = collect([5, 3, 1, 2, 4]);
$sorted = $collection->sort();
$sorted->values()->all();
// [1, 2, 3, 4, 5]
```

如果你需要更加高级的排序，你可以使用自己的算法传递一个回调给 `sort` 方法。参考 PHP 官方文档关于 `usort` 的说明，`sort` 方法底层正是调用了该方法。

注：要为嵌套集合和对象排序，查看 `sortBy` 和 `sortByDesc` 方法。

sortBy()

`sortBy` 方法通过给定键对集合进行排序，排序后的集合保持原有数组索引，在本例中，使用 `values` 方法重置键为连续索引：

```
$collection = collect([
    ['name' => 'Desk', 'price' => 200],
    ['name' => 'Chair', 'price' => 100],
    ['name' => 'Bookcase', 'price' => 150],
]);

$sorted = $collection->sortBy('price');

$sorted->values()->all();

/*
[
    ['name' => 'Chair', 'price' => 100],
    ['name' => 'Bookcase', 'price' => 150],
    ['name' => 'Desk', 'price' => 200],
]
*/
```

你还可以传递自己的回调来判断如何排序集合的值：

```
$collection = collect([
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Chair', 'colors' => ['Black']],
])
```

```
[ 'name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']] );  
  
$sorted = $collection->sortBy(function ($product, $key) {  
    return count($product['colors']);  
});  
  
$sorted->values()->all();  
  
/*  
 [  
     ['name' => 'Chair', 'colors' => ['Black']],  
     ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],  
     ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']]  
 ]  
 */
```

sortByDesc()

该方法和 `sortBy` 用法相同，不同之处在于按照相反顺序进行排序。

splice()

`splice` 方法在从给定位置开始移除并返回数据项切片：

```
$collection = collect([1, 2, 3, 4, 5]);  
$chunk = $collection->splice(2);  
$chunk->all();  
// [3, 4, 5]  
$collection->all();  
// [1, 2]
```

你可以传递参数来限制返回组块的大小：

```
$collection = collect([1, 2, 3, 4, 5]);  
$chunk = $collection->splice(2, 1);  
$chunk->all();  
// [3]  
$collection->all();  
// [1, 2, 4, 5]
```

此外，你可以传递第三个参数来包含新的数据项来替代从集合中移除的数据项：

```
$collection = collect([1, 2, 3, 4, 5]);  
$chunk = $collection->splice(2, 1, [10, 11]);  
$chunk->all();  
// [3]  
$collection->all();  
// [1, 2, 10, 11, 4, 5]
```

split()

split 方法通过给定数值对集合进行分组：

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$groups = $collection->split(3);  
  
$groups->toArray();  
// [[1, 2], [3, 4], [5]]
```

sum()

sum 方法返回集合中所有数据项的和：

```
collect([1, 2, 3, 4, 5])->sum();  
// 15
```

如果集合包含嵌套数组或对象，应该传递一个键用于判断对哪些值进行求和运算：

```
$collection = collect([  
    ['name' => 'JavaScript: The Good Parts', 'pages' => 176],  
    ['name' => 'JavaScript: The Definitive Guide', 'pages' => 109  
],  
]);
```

```
$collection->sum('pages');
// 1272
```

此外，你还可以传递自己的回调来判断对哪些值进行求和：

```
$collection = collect([
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]);
$collection->sum(function ($product) {
    return count($product['colors']);
});
// 6
```

take()

`take` 方法使用指定数目的数据项返回一个新的集合：

```
$collection = collect([0, 1, 2, 3, 4, 5]);
$chunk = $collection->take(3);
$chunk->all();
// [0, 1, 2]
```

你还可以传递负数从集合末尾开始获取指定数目的数据项：

```
$collection = collect([0, 1, 2, 3, 4, 5]);
$chunk = $collection->take(-2);
$chunk->all();
// [4, 5]
```

toArray()

`toArray` 方法将集合转化为一个原生的 PHP 数组。如果集合的值是 [Eloquent 模型](#)，该模型也会被转化为数组：

```
$collection = collect(['name' => 'Desk', 'price' => 200]);
$collection->toArray();

/*
```

```
[  
    ['name' => 'Desk', 'price' => 200],  
]  
*/
```

注：`toArray` 还将所有嵌套对象转化为数组。如果你想要获取底层数组，使用 `all` 方法。

toJson()

`toJson` 方法将集合转化为 JSON：

```
$collection = collect(['name' => 'Desk', 'price' => 200]);  
  
$collection->toJson();  
// '{"name":"Desk","price":200}'
```

transform()

`transform` 方法迭代集合并对集合中每个数据项调用给定回调。集合中的数据项将会被替换成从回调中返回的值：

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$collection->transform(function ($item, $key) {  
    return $item * 2;  
});  
  
$collection->all();  
// [2, 4, 6, 8, 10]
```

注意：不同于大多数其它集合方法，`transform` 修改集合本身，如果你想要创建一个新的集合，使用 `map` 方法。

union()

`union` 方法添加给定数组到集合，如果给定数组包含已经在原来集合中存在的键，原生集合的值会被保留：

```
$collection = collect([1 => ['a'], 2 => ['b']]));
```

```
$union = $collection->union([3 => ['c'], 1 => ['b']]);

$union->all();

// [1 => ['a'], 2 => ['b'], [3 => ['c']]
```

unique()

`unique` 方法返回集合中所有的唯一数据项，返回的集合保持原来的数组键，在本例中我们使用 `values` 方法重置这些键为连续的数字索引：

```
$collection = collect([1, 1, 2, 2, 3, 4, 2]);

$unique = $collection->unique();

$unique->values()->all();

// [1, 2, 3, 4]
```

处理嵌套数组或对象时，可以指定用于判断唯一的键：

```
$collection = collect([
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'iPhone 5', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
    ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],
]);

$unique = $collection->unique('brand');

$unique->values()->all();

/*
```

```
[  
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'  
> ],  
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'pho  
ne'],  
]  
*/
```

你还可以指定自己的回调用于判断数据项唯一性：

```
$unique = $collection->unique(function ($item) {  
    return $item['brand'].$item['type'];  
});  
  
$unique->values()->all();  
  
/*  
[  
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'  
> ],  
    ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'wat  
ch'],  
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'pho  
ne'],  
    ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'w  
atch'],  
]  
*/
```

values()

values 方法使用重置为连续整型数字的键返回新的集合：

```
$collection = collect([  
    10 => ['product' => 'Desk', 'price' => 200],  
    11 => ['product' => 'Desk', 'price' => 200]  
]);  
  
$values = $collection->values();  
  
$values->all();
```

```
/*
[
    0 => ['product' => 'Desk', 'price' => 200],
    1 => ['product' => 'Desk', 'price' => 200],
]
*/
```

where()

`where` 方法通过给定键值对过滤集合：

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->where('price', 100);

$filtered->all();

/*
[
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Door', 'price' => 100],
]
*/
```

检查数据项值时 `where` 方法使用严格条件约束。使用 `whereLoose` 方法过滤松散约束。

whereStrict()

该方法和 `where` 用法一样，不过，所有值都使用严格约束方式进行比较。

whereIn()

`whereIn` 方法通过包含在给定数组中的键值对集合进行过滤：

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
```

```
[ 'product' => 'Chair', 'price' => 100],  
[ 'product' => 'Bookcase', 'price' => 150],  
[ 'product' => 'Door', 'price' => 100],  
]);  
  
$filtered = $collection->whereIn('price', [150, 200]);  
  
$filtered->all();  
  
/*  
[  
    [ 'product' => 'Bookcase', 'price' => 150],  
    [ 'product' => 'Desk', 'price' => 200],  
]  
*/
```

`whereIn` 方法在检查数据项值的时候使用严格的约束，要使用非严格的约束可以使用 `whereInLoose` 方法。

whereLoose()

该方法和 `whereIn` 使用方法相同，不同之处在于 `whereLoose` 在比较值的时候使用非严格的约束。

zip()

`zip` 方法在于集合的值相应的索引处合并给定数组的值：

```
$collection = collect(['Chair', 'Desk']);  
$zipped = $collection->zip([100, 200]);  
$zipped->all();  
// [['Chair', 100], ['Desk', 200]]
```

15.2 辅助函数

1、简介

Laravel 自带了一系列 PHP 辅助函数，很多被框架自身使用，如果你觉得方便的话也可以在代码中使用它们。

2、方法列表

数组函数

array_add()

`array_add` 函数添加给定键值对到数组，如果给定键不存在的话：

```
$array = array_add(['name' => 'Desk'], 'price', 100);
// ['name' => 'Desk', 'price' => 100]
```

array_collapse()

`array_collapse` 函数将多个数组合并成一个：

```
$array = array_collapse([[1, 2, 3], [4, 5, 6], [7, 8, 9]]);
// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

array_divide()

`array_divide` 函数返回两个数组，一个包含原数组的所有键，另外一个包含原数组的所有值：

```
list($keys, $values) = array_divide(['name' => 'Desk']);
// $keys: ['name']
// $values: ['Desk']
```

array_dot()

`array_dot` 函数使用“.”号将将多维数组转化为一维数组：

```
$array = array_dot(['foo' => ['bar' => 'baz']]);
// ['foo.bar' => 'baz'];
```

array_except()

`array_except` 方法从数组中移除给定键值对：

```
$array = ['name' => 'Desk', 'price' => 100];

$array = array_except($array, ['price']);
// ['name' => 'Desk']
```

array_first()

`array_first` 方法返回通过测试数组的第一个元素：

```
$array = [100, 200, 300];

$value = array_first($array, function ($key, $value) {
    return $value >= 150;});
// 200
```

默认值可以作为第三个参数传递给该方法，如果没有值通过测试的话返回默认值：

```
$value = array_first($array, $callback, $default);
```

array_flatten()

array_flatten 方法将多维数组转化为一维数组：

```
$array = ['name' => 'Joe', 'languages' => ['PHP', 'Ruby']];  
  
$array = array_flatten($array);  
// ['Joe', 'PHP', 'Ruby'];
```

array_forget()

array_forget 方法使用“.”号从嵌套数组中移除给定键值对：

```
$array = ['products' => ['desk' => ['price' => 100]]];  
  
array_forget($array, 'products.desk');  
// ['products' => []]
```

array_get()

array_get 方法使用“.”号从嵌套数组中获取值：

```
$array = ['products' => ['desk' => ['price' => 100]]];  
  
$value = array_get($array, 'products.desk');  
// ['price' => 100]
```

array_get 函数还接收一个默认值，如果指定键不存在的话则返回该默认值：

```
$value = array_get($array, 'names.john', 'default');
```

array_has()

`array_has` 函数使用“.”检查给定数据项是否在数组中存在：

```
$array = ['product' => ['name' => 'desk', 'price' => 100]];  
  
$hasItem = array_has($array, 'product.name');  
  
// true  
  
$hasItems = array_has($array, ['product.price', 'product.discount']);  
  
// false
```

array_last()

`array_last` 函数通过测试数组的最后一个元素：

```
$array = [100, 200, 300, 110];  
  
$value = array_last($array, function ($value, $key) {  
    return $value >= 150;  
});  
  
// 300
```

array_only()

`array_only` 方法只从给定数组中返回指定键值对：

```
$array = ['name' => 'Desk', 'price' => 100, 'orders' => 10];  
  
$array = array_only($array, ['name', 'price']);  
// ['name' => 'Desk', 'price' => 100]
```

array_pluck()

`array_pluck` 方法从数组中返回给定键对应的键值对列表：

```
$array = [  
    ['developer' => ['name' => 'Taylor']],  
    ['developer' => ['name' => 'Abigail']]  
];  
  
$array = array_pluck($array, 'developer.name');  
// ['Taylor', 'Abigail'];
```

你还可以指定返回结果的键：

```
$array = array_pluck($array, 'developer.name', 'developer.id');  
// [1 => 'Taylor', 2 => 'Abigail'];
```

arrayprepend()

`arrayprepend` 函数将数据项推入数组开头：

```
$array = ['one', 'two', 'three', 'four'];  
  
$array = arrayprepend($array, 'zero');  
// $array: ['zero', 'one', 'two', 'three', 'four']
```

arraypull()

`arraypull` 方法从数组中返回并移除键值对：

```
$array = ['name' => 'Desk', 'price' => 100];  
  
$name = array_pull($array, 'name');  
// $name: Desk  
  
// $array: ['price' => 100]
```

array_set()

array_set 方法在嵌套数组中使用“.”号设置值：

```
$array = ['products' => ['desk' => ['price' => 100]]];  
  
array_set($array, 'products.desk.price', 200);  
// ['products' => ['desk' => ['price' => 200]]]
```

array_sort()

array_sort 方法通过给定闭包的结果对数组进行排序：

```
$array = [  
    ['name' => 'Desk'],  
    ['name' => 'Chair'],  
];  
  
$array = array_values(array_sort($array, function ($value) {  
    return $value['name'];  
}));  
  
/*
```

```
[  
    ['name' => 'Chair'],  
    ['name' => 'Desk'],  
]  
*/
```

array_sort_recursive()

`array_sort_recursive` 函数使用 `sort` 函数对数组进行递归排序：

```
$array = [  
    [  
        'Roman',  
        'Taylor',  
        'Li',  
    ],  
    [  
        'PHP',  
        'Ruby',  
        'JavaScript',  
    ],  
];  
  
$array = array_sort_recursive($array);  
  
/*  
[  
    [  
        'Li',  
    ],  
    [  
        'Taylor',  
        'Roman',  
    ],  
    [  
        'JavaScript',  
        'Ruby',  
        'PHP',  
    ],  
]
```

```
'Roman',
'Taylor',
],
[
'JavaScript',
'PHP',
'Ruby',
]
];
*/

```

array_where()

`array_where` 函数使用给定闭包对数组进行过滤：

```
$array = [100, '200', 300, '400', 500];

$array = array_where($array, function ($key, $value) {
    return is_string($value);
});

// [1 => 200, 3 => 400]
```

head()

`head` 函数只是简单返回给定数组的第一个元素：

```
$array = [100, 200, 300];

$first = head($array);

// 100
```

last()

`last` 函数返回给定数组的最后一个元素：

```
$array = [100, 200, 300];

$last = last($array);

// 300
```

路径函数

app_path()

`app_path` 函数返回 `app 目录` 的绝对路径，你还可以使用 `app_path` 函数为相对于 `app` 目录的给定

文件生成绝对路径：

```
$path = app_path();

$path = app_path('Http/Controllers/Controller.php');
```

base_path()

`base_path` 函数返回项目根目录的绝对路径，你还可以使用 `base_path` 函数为相对于应用目录的

给定文件生成绝对路径：

```
$path = base_path();

$path = base_path('vendor/bin');
```

config_path()

`config_path` 函数返回应用配置目录的绝对路径：

```
$path = config_path();
```

database_path()

`database_path` 函数返回应用数据库目录的绝对路径：

```
$path = database_path();
```

elixir()

`elixir` 函数返回版本控制的 Elixir 文件所在路径：

```
elixir($file);
```

public_path()

`public_path` 函数返回 `public` 目录的绝对路径：

```
$path = public_path();
```

resource_path()

`resource_path` 函数返回 `resources` 目录的绝对路径，还可以使用 `resources` 函数生成给定相

对 `resources` 目录文件的绝对路径：

```
$path = resource_path();
```

```
$path = resource_path('assets/sass/app.scss');
```

storage_path()

`storage_path` 函数返回 `storage` 目录的绝对路径，还可以使用 `storage_path` 函数生成相对

于 `storage` 目录的给定文件的绝对路径：

```
$path = storage_path();  
$path = storage_path('app/file.txt');
```

字符串函数

camel_case()

`camel_case` 函数将给定字符串转化为按驼峰式命名规则的字符串：

```
$camel = camel_case('foo_bar');  
// fooBar
```

class_basename()

`class_basename` 返回给定类移除命名空间后的类名：

```
$class = class_basename('Foo\Bar\Baz');  
// Baz
```

e()

`e` 函数在给定字符串上运行 `htmlentities`：

```
echo e('<html>foo</html>');  
// &lt;html&gt;foo&lt;/html&gt;
```

ends_with()

`ends_with` 函数判断给定字符串是否以给定值结尾：

```
$value = ends_with('This is my name', 'name');  
// true
```

snake_case()

`snake_case` 函数将给定字符串转化为下划线分隔的字符串：

```
$snake = snake_case('fooBar');  
// foo_bar
```

str_limit()

`str_limit` 函数限制输出字符串的数目，该方法接收一个字符串作为第一个参数以及该字符串最大输出字符数作为第二个参数：

```
$value = str_limit('The PHP framework for web artisans.', 7);  
// The PHP...
```

starts_with()

`starts_with` 函数判断给定字符串是否以给定值开头：

```
$value = starts_with('This is my name', 'This');  
// true
```

str_contains()

`str_contains` 函数判断给定字符串是否包含给定值：

```
$value = str_contains('This is my name', 'my');  
// true
```

你还可以传递数组来判断给定字符串是否包含数组中的值：

```
$value = str_contains('This is my name', ['my', 'foo']);  
// true
```

str_finish()

`str_finish` 函数添加字符到字符串结尾：

```
$string = str_finish('this/string', '/');  
// this/string/
```

str_is()

`str_is` 函数判断给定字符串是否与给定模式匹配，星号可用于表示通配符：

```
$value = str_is('foo*', 'foobar');  
// true  
  
$value = str_is('baz*', 'foobar');  
// false
```

str_plural()

`str_plural` 函数将字符串转化为复数形式，该函数当前只支持英文：

```
$plural = str_plural('car');  
// cars  
  
$plural = str_plural('child');  
// children
```

还可以传递整型数据作为第二个参数到该函数以获取字符串的单数或复数形式：

```
$plural = str_plural('child', 2);  
// children
```

```
$plural = str_plural('child', 1);  
// child
```

str_random()

`str_random` 函数通过指定长度生成随机字符串，该函数使用了 PHP 的 `random_bytes` 函数：

```
$string = str_random(40);
```

str_singular()

`str_singular` 函数将字符串转化为单数形式，该函数目前只支持英文：

```
$singular = str_singular('cars');  
// car
```

str_slug()

`str_slug` 函数将给定字符串生成 URL 友好的格式：

```
$title = str_slug("Laravel 5 Framework", "-");  
// laravel-5-framework
```

studly_case()

`studly_case` 函数将给定字符串转化为单词开头字母大写的格式：

```
$value = studly_case('foo_bar');  
// FooBar
```

title_case()

`title_case` 函数将字符串转化为 Title 形式：

```
$title = title_case('a nice title uses the correct case');

// A Nice Title Uses The Correct Case
```

trans()

`trans` 函数使用本地文件翻译给定语言行：

```
echo trans('validation.required');
```

trans_choice()

`trans_choice` 函数翻译带拐点的给定语言行：

```
$value = trans_choice('foo.bar', $count);
```

URL 函数

action()

`action` 函数为给定控制器动作生成 URL，你不需要传递完整的命名空间到该控制器，传递相对于命名空间 `App\Http\Controllers` 的类名即可：

```
$url = action('HomeController@getIndex');
```

如果该方法接收路由参数，你可以将其作为第二个参数传递进来：

```
$url = action('UserController@profile', ['id' => 1]);
```

asset()

使用当前请求的 scheme (HTTP 或 HTTPS) 为前端资源生成一个 URL：

```
$url = asset('img/photo.jpg');
```

secure_asset()

使用 HTTPS 为前端资源生成一个 URL :

```
echo secure_asset('foo/bar.zip', $title, $attributes = []);
```

route()

`route` 函数为给定命名路由生成一个 URL :

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

如果该路由接收参数，你可以将其作为第二个参数传递进来：

```
$url = route('routeName', ['id' => 1]);
```

url()

`url` 函数为给定路径生成绝对路径：

```
echo url('user/profile');
echo url('user/profile', [1]);
```

如果没有提供路径，将会返回 `Illuminate\Routing\UrlGenerator` 实例：

```
echo url()->current();
echo url()->full();
```

```
echo url()->previous();
```

其它函数

abort()

`abort` 函数会抛出一个被异常处理器渲染的 HTTP 异常：

```
abort(401);
```

还可以提供异常响应文本：

```
abort(401, 'Unauthorized.');
```

abort_if()

`abort_if` 函数在给定布尔表达式为 `true` 时抛出 HTTP 异常：

```
abort_if(! Auth::user()->isAdmin(), 403);
```

abort_unless()

`abort_unless` 函数在给定布尔表达式为 `false` 时抛出 HTTP 异常：

```
abort_unless(Auth::user()->isAdmin(), 403);
```

auth()

`auth` 函数返回一个认证器实例，为方便起见你可以用其取代 `Auth` 门面：

```
$user = auth()->user();
```

back()

`back` 函数生成重定向响应到用户前一个位置：

```
return back();
```

bcrypt()

`bcrypt` 函数使用 Bcrypt 对给定值进行哈希，你可以用其替代 `Hash` 门面：

```
$password = bcrypt('my-secret-password');
```

cache()

`cache` 函数可以用于从缓存中获取值，如果给定 `key` 在缓存中不存在，可选的默认值会被返回：

```
$value = cache('key');  
$value = cache('key', 'default');
```

你可以通过传递数组键值对到函数来添加数据项到缓存。还需要传递缓存有效期（分钟数）：

```
cache(['key' => 'value'], 5);  
cache(['key' => 'value'], Carbon::now()->addSeconds(10));
```

collect()

`collect` 函数会根据提供的数据项创建一个集合：

```
$collection = collect(['taylor', 'abigail']);
```

config()

`config` 函数获取配置变量的值，配置值可以通过使用””号访问，包含文件名以及你想要访问的选项。如果配置选项不存在的话默认值将会被指定并返回：

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

辅助函数 `config` 还可以用于在运行时通过传递键值对数组设置配置变量值：

```
config(['app.debug' => true]);
```

csrf_field()

`csrf_field` 函数生成一个包含 CSRF 令牌值的 HTML 隐藏域，例如，使用 [Blade 语法](#)：

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

csrf_token()

`csrf_token` 函数获取当前 CSRF 令牌的值：

```
$token = csrf_token();
```

dd()

`dd` 函数输出给定变量值并终止脚本执行：

```
dd($value);  
dd($value1, $value2, $value3, ...);
```

如果你不想停止脚本的运行，可以使用 `dump` 函数：

```
dump($value);
```

dispatch()

`dispatch` 函数推送一个新的任务到 Laravel 任务队列：

```
dispatch(new App\Jobs\SendEmails);
```

env()

`env` 函数获取环境变量值或返回默认值：

```
$env = env('APP_ENV');  
// Return a default value if the variable doesn't exist...  
$env = env('APP_ENV', 'production');
```

event()

`event` 函数分发给定事件到对应监听器：

```
event(new UserRegistered($user));
```

factory()

`factory` 函数为给定类、名称和数量创建模型工厂构建器，可用于测试或数据填充：

```
$user = factory(App\User::class)->make();
```

info()

`info` 函数会记录日志信息：

```
info('Some helpful information!');
```

还可以传递上下文数据数组到该函数：

```
info('User login attempt failed.', ['id' => $user->id]);
```

logger()

`logger` 函数可以用于记录 `debug` 级别的日志消息：

```
logger('Debug message');
```

同样，也可以传递上下文数据数组到该函数：

```
logger('User has logged in.', ['id' => $user->id]);
```

如果没有值传入该函数的话会返回 `logger` 实例：

```
logger()->error('You are not allowed here.');
```

method_field()

`method_field` 函数生成包含 HTTP 请求方法的 HTML `hidden` 输入字段，例如：

```
<form method="POST">  
    {!! method_field('DELETE') !!}  
</form>
```

old()

`old` 函数获取一次性存放在 Session 中的值：

```
$value = old('value');  
$value = old('value', 'default');
```

redirect()

`redirect` 函数返回重定向器实例进行重定向：

```
return redirect('/home');  
return redirect()->route('route.name');
```

request()

`request` 函数返回当前请求实例或者获取一个输入项：

```
$request = request();  
$value = request('key', $default = null)
```

response()

`response` 函数创建一个响应实例或者获取响应工厂实例：

```
return response('Hello World', 200, $headers);  
return response()->json(['foo' => 'bar'], 200, $headers)
```

session()

`session` 函数可以用于获取/设置 Session 值：

```
$value = session('key');
```

可以通过传递键值对数组到该函数的方式设置 Session 值：

```
session(['chairs' => 7, 'instruments' => 3]);
```

如果没有传入参数到 `session` 函数则返回 Session 存储器对象实例：

```
$value = session()->get('key');
```

```
session()->put('key', $value);
```

value()

`value` 函数返回给定的值，然而，如果你传递一个闭包到该函数，该闭包将被执行并返回执行

结果：

```
$value = value(function() { return 'bar'; });
```

view()

`view` 函数获取一个[视图](#)实例：

```
return view('auth.login');
```

with()

`with` 函数返回给定的值，该函数在方法链中特别有用，别的地方就没什么用了：

```
$value = with(new Foo)->work();
```

15.3 包开发

1、简介

包是添加功能到 [Laravel](#) 的主要方式。包可以提供任何功能，小到处理日期如 [Carbon](#)，大到整个

BDD 测试框架如 [Behat](#)。

当然，有很多不同类型的包。有些包是独立的，意味着可以在任何框架中使用，而不仅是 Laravel。

比如 Carbon 和 Behat 都是独立的包。所有这些包都可以通过在 `composer.json` 文件中请求以便被 Laravel 使用。

另一方面，其它包只能特定和 Laravel 一起使用，这些包可能有路由，控制器、[视图](#)和[配置](#)用于

加强 Laravel 应用的功能，本章主要讨论只能在 Laravel 中使用的包。

关于门面的注意点

编写 Laravel 应用时，不管你使用契约还是门面，通常并没有什么关系，因为两者都提供了基本同等级别的可测试性。不过，编写[扩展包](#)时，最好使用契约而不是门面，由于你的扩展包不能访问所有的 Laravel 测试辅助函数，所以模拟或存根契约往往比模拟门面来得更容易些。

2、服务提供者

[服务提供者](#)是包和 Laravel 之间的连接点。服务提供者负责绑定对象到 Laravel 的[服务容器](#)并告知 Laravel 从哪里加载包资源如视图、配置和本地化文件。

服务提供者继承自 `Illuminate\Support\ServiceProvider` 类并包含两个方法：`register` 和 `boot`。

`ServiceProvider` 基类位于 Composer 包 `illuminate/support`。

要了解更多关于服务提供者的内容，查看其[文档](#)。

3、路由

要定义包的路由，只需要在包服务提供者中的 `boot` 方法中引入路由文件。在路由文件中，可以使
用 `Illuminate\Support\Facades\Route` 门面[注册路由](#)，和 Laravel 应用中注册路由一样：

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot(){
    if (! $this->app->routessAreCached()) {
        require __DIR__ . '/../routes.php';
    }
}
```

```
    }  
}
```

4、资源

配置

通常，需要发布包配置文件到应用根目录下的 `config` 目录，这将允许包用户轻松覆盖默认配置选项，要发布一个配置文件，只需在服务提供者的 `boot` 方法中使用 `publishes` 方法即可：

```
/**  
 * Perform post-registration booting of services.  
 *  
 * @return void  
 */  
public function boot(){  
    $this->publishes([  
        __DIR__.'/path/to/config/courier.php' => config_path('courier.php'),  
    ]);  
}
```

现在，当包用户执行 Laravel 的 Artisan 命令 `vendor:publish` 时，你的文件将会被拷贝到指定位置，当然，配置被发布后，可以通过和其它配置选项一样的方式进行访问：

```
$value = config('courier.option');
```

默认包配置

你还可以选择将自己的包配置文件合并到应用的拷贝版本，这允许用户只引入他们在应用配置文件中实际想要覆盖的配置选项。要合并两个配置，在服务提供者的 `register` 方法中使用 `mergeConfigFrom` 方法即可：

```
/**  
 * 在容器中注册绑定  
 *  
 * @return void  
 */  
  
public function register(){  
    $this->mergeConfigFrom(  
        __DIR___. '/path/to/config/courier.php', 'courier'  
    );  
}
```

迁移

如果你的包包含数据库迁移，可以使用 `loadMigrationsFrom` 方法告知 Laravel 如何加载它们。`loadMigrationsFrom` 方法接收扩展包迁移的路径作为其唯一参数：

```
/**  
 * Perform post-registration booting of services.  
 *  
 * @return void  
 */  
  
public function boot()  
{  
    $this->loadMigrationsFrom(__DIR___. '/path/to/migrations');  
}
```

扩展包迁移注册好之后，会在执行 `php artisan migrate` 时自动运行。不需要将它们导出到应用

的 `database/migrations` 目录。

翻译

如果你的包包含[翻译文件](#)，你可以使用 `loadTranslationsFrom` 方法告诉 Laravel 如何加载它们，

例如，如果你的包命名为“courier”，你应该添加如下代码到服务提供者的 `boot` 方法：

```
/**  
 * Perform post-registration booting of services.  
 *  
 * @return void  
 */  
public function boot()  
{  
    $this->loadTranslationsFrom(__DIR__ . '/path/to/translations', 'c  
ourier');  
}
```

扩展包翻译使用形如 `package::file.line` 的语法进行引用。所以，你可以使用如下方式从

`messages` 文件中加载 `courier` 包的 `welcome` 行：

```
echo trans('courier::messages.welcome');
```

发布翻译文件

如果你想要发布包翻译到应用的 `resources/lang/vendor` 目录，你可以使用服务提供者的

`publishes` 方法，该方法接收一个包路径和相应发布路径数组参数，例如，要发布 `courier` 包的翻

译文件，可以这么做：

```
/**  
 * Perform post-registration booting of services.  
 *  
 * @return void
```

```
/*
public function boot(){

    $this->loadTranslationsFrom(__DIR__().'/path/to/translations', 'courier');

    $this->publishes([
        __DIR__.'/path/to/translations' => resource_path('lang/vendor/courier'),
    ]);
}
```

这样，包用户可以执行 Artisan 命令 `vendor:publish` 将包翻译文件发布到应用的指定目录。

视图

要在 Laravel 中注册包[视图](#)，需要告诉 Laravel 视图在哪，可以使用服务提供者的 `loadViewsFrom` 方法来实现。`loadViewsFrom` 方法接收两个参数：视图模板的路径和包名称。例如，如果你的包名称是“courier”，添加如下代码到服务提供者的 `boot` 方法：

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot(){
    $this->loadViewsFrom(__DIR__.'/path/to/views', 'courier');
}
```

包视图通过使用类似的 `package::view` 语法来引用。所以，你可以通过如下方式加载 `courier` 包上的 `admin` 视图：

```
Route::get('admin', function () {
    return view('courier::admin');
});
```

覆盖包视图

当你使用 `loadViewsFrom` 方法的时候， Laravel 实际上为视图注册了两个存放位置：一个是 `resources/views/vendor` 目录，另一个是你指定的目录。所以，以 `courier` 为例：当请求一个包视图时， Laravel 首先检查开发者是否在 `resources/views/vendor/courier` 提供了自定义版本的视图，如果该视图不存在， Laravel 才会搜索你调用 `loadViewsFrom` 方法时指定的目录。这种机制使得终端用户可以轻松地自定义/覆盖包视图。

发布视图

如果你想要视图能够发布到应用的 `resources/views/vendor` 目录，可以使用服务提供者的 `publishes` 方法。该方法接收包视图路径及其相应的发布路径数组作为参数：

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot(){
    $this->loadViewsFrom(__DIR__.'/path/to/views', 'courier');

    $this->publishes([
        __DIR__.'/path/to/views' => base_path('resources/views/vendor/courier'),
    ]);
}
```

现在，当包用户执行 Laravel 的 Artisan 命令 `vendor:publish` 时，你的视图包将会被拷贝到指定路径。

5、命令

要通过 Laravel 注册扩展包的 Artisan 命令，可以使用 `commands` 方法。该方法需要传入命令名称

数组，注册号命令后，可以使用 [Artisan CLI](#) 执行它们：

```
/**
 * Bootstrap the application services.
 *
 * @return void
 */
public function boot()
{
    if ($this->app->runningInConsole()) {
        $this->commands([
            FooCommand::class,
            BarCommand::class,
        ]);
    }
}
```

6、前端资源

你的包可能包含 JavaScript、CSS 和图片，要发布这些前端资源到应用根目录下的 `public` 目录，

使用服务提供者的 `publishes` 方法。在本例中，我们添加一个前端资源组标签 `public`，用于发布相关的前端资源组：

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot(){
```

```
$this->publishes([
    __DIR__.'/path/to/assets' => public_path('vendor/courier'),
], 'public');
}
```

现在，当包用户执行 `vendor:publish` 命令时，前端资源将会被拷贝到指定位置，由于你需要在每次包更新时重写前端资源，可以使用 `--force` 标识：

```
php artisan vendor:publish --tag=public --force
```

7、发布文件组

有时候你可能想要分开发布包前端资源组和资源，例如，你可能想要用户发布包配置的同时不發布包前端资源，可以通过在扩展包的服务提供者中调用 `publishes` 方法时给它们打上“标签”来实现分离。下面我们在扩展包服务提供者的 `boot` 方法中定义两个发布组：

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot(){
    $this->publishes([
        __DIR__.'/../config/package.php' => config_path('package.php')
    ], 'config');

    $this->publishes([
        __DIR__.'/../database/migrations/' => database_path('migrations')
    ], 'migrations');
}
```

现在用户可以在使用 Artisan 命令 `vendor:publish` 时通过引用标签名来分开发布这两个组：

```
php artisan vendor:publish --tag="config"
```