服务端\_PHP\_框架\_Laravel\_Laravel开发文档(6.x)

--王尔贝

服务端\_PHP\_框架\_Laravel\_Laravel开发文档(6.x)

--

2workroom3工作室

date 2020-04-22

--

//// laravel/6.x

// 中文

https://learnku.com/docs/laravel/6.x

// 英文

https://laravel.com/docs/6.x

//

//// 文档项目 project-laravel-laravelv6x

//0/ 环境

MacBookPro(php7.3.11) + PhpStorm (+ Laragon)

//

//1/ 项目创建

composer create-project --prefer-dist laravel/laravel project-laravel-laravelv6x "6.\*"

//

//2/ 仓库创建

https://gitee.com/cyb\_43/project-laravel-laravelv6x.git

//

//3/ 项目仓库添加

cd project-laravel-laravelv6x

git init

git remote add origin-gitee https://gitee.com/cyb\_43/project-laravel-laravelv6x.git

git add .

git commit -m '初始化仓库'

git push origin-gitee master

--

--

第一卷 前言

--

--

一、发行说明

https://learnku.com/docs/laravel/6.x/releases/5121

//

本文档最新版为 7.x( https://learnku.com/docs/laravel/7.x )，旧版本可能放弃维护，推荐阅读最新版！

--

1、语义化版本

Laravel和官方扩展包将正式遵循 语义化版本( https://semver.org/lang/zh-CN/ )。每六个月会发布一个大版本（如：6.0、7.0），每周可能会有小版本和补丁版本发布。小版本和补丁 绝不 包含破坏性变更。

当你在项目或者扩展包依赖于 Laravel 框架时，因大版本会包含破坏性变更，故必须使用大版本限制，如：^6.0。即便如此，我们仍然会努力减低你升级时的耗时，使其控制在一天或者更少。

--

2、支持策略

对于 LTS( 长期支持 Long Term Support ) 版本，例如 Laravel 6.0，将会提供为期两年的错误修复和三年的安全修复。这些版本提供了最长时间的支持和维护。对于一般版本，只会提供六个月的错误修复和一年的安全修复。对于所有其他库，包括 Lumen ，只有最新版本会得到错误修复。

//

版本 发布时间 Bug 修复截止时间 安全修复截止时间

5.5 (LTS) 2017 年 8 月 30 日 2019 年 8 月 30 日 2020 年 8 月 30 日

5.6 2018 年 2 月 7 日 2018 年 8 月 7 日 2019 年 2 月 7 日

5.7 2018 年 9 月 4 日 2019 年 3 月 4 日 2019 年 9 月 4 日

5.8 2019 年 2 月 26 日 2019 年 8 月 26 日 2020 年 2 月 26 日

6.0 (LTS) 2019 年 9 月 3 日 2021 年 9 月 3 日 2022 年 9 月 3 日

--

3、Laravel 6.0

Laravel 6.0（LTS）通过引入语义化版本( https://semver.org/lang/zh-CN/ )、与 Laravel Vapor( https://vapor.laravel.com/ ) 的兼容性、优化了授权响应、任务中间件、懒集合、子查询优化、并将前端脚手架单独分离到 laravel/ui Composer 软件包，以及其他各种错误修复和可用性改进。

(1)、语义版本控制

laravel 框架扩展包现在开始使用语义化版本( https://semver.org/lang/zh-CN/ )。这使得框架与已经遵循此版本控制标准的其他第一方 Laravel 包保持一致。Laravel 发布周期将保持不变。

(2)、Laravel Vapor 兼容性

Laravel Vapor 由 Taylor Otwell( https://github.com/taylorotwell ) 开发。

Laravel 6.0 提供了与 Laravel Vapor( https://vapor.laravel.com/ ) 的兼容性，这是一个用于 Laravel 的自动伸缩无服务器部署平台。vapor 抽象了管理 aws lambda 上的 laravel 应用程序的复杂性，以及将这些应用程序与 sqs 队列、数据库、redis 集群、网络、 CloudFront CDN 等接口的复杂性。

(3)、改进的异常：Ignition

Laravel 自带了 Ignition( https://github.com/facade/ignition )，这是一个由 Freek Van der Herten 与 Marcel Pociot 创建的关于异常详情页面的新的开源项目。相较之前的版本，Ignition 具有许多优势，比如改进的错误页面 Blade 文件与行号处理、对常见问题的运行时解决、代码编辑、异常共享以及改进的用户体验。

(4)、改进的授权响应

改进的授权响应由 Gary Green（ https://github.com/garygreen ） 实现。

在之前的 Laravel 版本中，很难提取并向最终的用户展示自定义的授权消息，这就导致了很难跟最终用户解释某个特定请求为什么会被拒绝。在 Laravel 6.0 中，使用授权响应消息和新方法 Gate::inspect 将会让事情变得更加容易。比如，给定以下授权策略（Policy）方法。

/\*\*

\* 判断用户能否查看指定的航班

\*

\* @param \App\User $user

\* @param \App\Flight $flight

\* @return mixed

\*/

public function view(User $user, Flight $flight)

{

return $this->deny('Explanation of denial.');

}

使用 Gate::inspect 方法可以很容易地提取出授权策略的响应和消息。

$response = Gate::inspect('view', $flight);

//

if ($response->allowed()) {

// 用户被允许查看该航班……

}

//

if ($response->denied()) {

echo $response->message();

}

除此之外，当你在路由或者控制器中使用了诸如 $this->authorize 或 Gate::authorize 的辅助方法后，这些自定义信息将能够自动地返回给你的前端。

(5)、任务中间件

任务中间件是由 Taylor Otwell（ https://github.com/taylorotwell ） 实现的。

任务中间件允许你围绕任务队列的执行封装自定义逻辑，从而减少任务本身的引用。例如：在 Laravel 的上一个发行版本里，你可能封装了一个有关速率限制的回调逻辑在任务的 handle 方法中。

/\*\*

\* 执行任务

\*

\* @return void

\*/

public function handle()

{

Redis::throttle('key')->block(0)->allow(1)->every(5)->then(function () {

info('获取锁...');

// 处理任务...

}, function () {

// 无法获取锁...

return $this->release(5);

});

}

在 Laravel 6.0 里，这个逻辑可以被提取到一个任务中间件中，从而让你任务的 handle 方法不用承担任何速率限制的职责。

<?php

namespace App\Jobs\Middleware;

use Illuminate\Support\Facades\Redis;

class RateLimited

{

/\*\*

\* 处理任务队列

\*

\* @param mixed $job

\* @param callable $next

\* @return mixed

\*/

public function handle($job, $next)

{

Redis::throttle('key')

->block(0)->allow(1)->every(5)

->then(function () use ($job, $next) {

// 获取锁...

$next($job);

}, function () use ($job) {

// 无法获取锁...

$job->release(5);

});

}

}

?>

创建中间件之后，它们可以通过返回任务的 middleware 方法连接到任务。

use App\Jobs\Middleware\RateLimited;

/\*\*

\* 获得任务的中间件

\*

\* @return array

\*/

public function middleware()

{

return [new RateLimited];

}

(6)、懒集合

懒集合 由 Joseph Silber（ https://github.com/JosephSilber ） 开发 。

许多开发者已经感受到了 Laravel集合 的强大。为了补充已经很强大的 集合( https://learnku.com/docs/laravel/6.x/collections ) 类， Laravel 6.0 引入了 一个，利用 PHP 的 生成器( https://www.php.net/manual/zh/language.generators.overview.php ) 来实现的 懒集合，允许你使用非常大的数据集合的同时保持较低的内存使用率。

例如，想像你的应用程序需要处理 1 个多 GB 大小的日志文件，同时利用 Laravel 的集合来解析日志。通过懒集合可以在给定时间内仅将文件的一部分保留在内存中，而不是一次将整个文件都读入内存。

use App\LogEntry;

use Illuminate\Support\LazyCollection;

LazyCollection::make(function () {

$handle = fopen('log.txt', 'r');

while (($line = fgets($handle)) !== false) {

yield $line;

}

})

->chunk(4)

->map(function ($lines) {

return LogEntry::fromLines($lines);

})

->each(function (LogEntry $logEntry) {

// Process the log entry...

});

或者，想像你需要迭代 10,000 个 Eloquent 模型，使用传统 Laravel 集合时，所有 10,000 个 Eloquent 模型必须同时加载到内存中。

$users = App\User::all()->filter(function ($user) {

return $user->id > 500;

});

然而，从 Laravel 6.0 开始，查询构建器的 cursor 方法已经更新为返回 懒集合 实例。虽然你仍然只能运行单个查询，但是一次只会在内存中保留一个 Eloquent 模型。在整个例子中，直到我们真正迭代完每个用户之后，filter 回调才会执行，从而大大减少了内存的使用。

$users = App\User::cursor()->filter(function ($user) {

return $user->id > 500;

});

foreach ($users as $user) {

echo $user->id;

}

(7)、Eloquent 子查询增强功能

Eloquent 子查询增强功能是由 Jonathan Reinink（ https://github.com/reinink ） 实现的。

Laravel 6.0 引入了几个对数据库子查询支持的新增强和改进。例如，假设我们有一个航班 destinations 表和一个飞往目的地的 flights 表。flights 表包含一个 arrived\_at 列，表示航班何时到达目的地。

使用 Laravel 6.0 中新的子查询选择功能，我们可以使用一个查询选择所有的 destinations 和最近到达该目的地的航班的名称。

return Destination::addSelect(['last\_flight' => Flight::select('name')

->whereColumn('destination\_id', 'destinations.id')

->orderBy('arrived\_at', 'desc')

->limit(1)

])->get();

并且我们可以使用添加到查询生成器的 orderBy 函数中的新子查询功能，根据最后一次航班到达目的地的时间对所有目的地进行排序。同样，这可以在对数据库执行单个查询时完成。

return Destination::orderByDesc(

Flight::select('arrived\_at')

->whereColumn('destination\_id', 'destinations.id')

->orderBy('arrived\_at', 'desc')

->limit(1)

)->get();

(8)、Laravel UI

通常在早期版本的 Laravel 中提供的前端脚手架已被提取到 laravel/ui Composer 包中。这将允许 UI 脚手架本身可以独立于主框架进行开发和版本化。经过此次更改，默认框架脚手架中已经不存在 Bootstrap 或 Vue 代码，并且还从框架中提取了 make:auth 命令。

你可以通过安装 laravel/ui 包并使用 ui Artisan 命令安装前端脚手架 ，来恢复前期 Laravel 版本中的传统 Vue / Bootstrap 脚手架。

composer require laravel/ui

php artisan ui vue --auth

--

--

二、 升级说明

https://learnku.com/docs/laravel/6.x/upgrade/5122

--

1、升级指南

(1)、从5.8升级到6.0版本

https://learnku.com/docs/laravel/6.x/upgrade/5122#upgrade-6.0

--

2、高影响变化

(1)、授权认证资源 & viewAny

https://learnku.com/docs/laravel/6.x/upgrade/5122#authorized-resources

(2)、字符串和数组辅助函数

https://learnku.com/docs/laravel/6.x/upgrade/5122#helpers

--

3、中影响变化

(1)、授权认证 RegisterController

https://learnku.com/docs/laravel/6.x/upgrade/5122#the-register-controller

(2)、Carbon 1.x 不再支持

https://learnku.com/docs/laravel/6.x/upgrade/5122#carbon-support

(3)、数据库 Capsule::table 方法

https://learnku.com/docs/laravel/6.x/upgrade/5122#capsule-table

(4)、Eloquent Arrayable & toArray

https://learnku.com/docs/laravel/6.x/upgrade/5122#eloquent-to-array

(5)、Eloquent BelongsTo::update 方法

https://learnku.com/docs/laravel/6.x/upgrade/5122#belongs-to-update

(6)、Eloquent 主键类型

https://learnku.com/docs/laravel/6.x/upgrade/5122#eloquent-primary-key-type

(7)、本地化 Lang::trans 和 Lang::transChoice 方法

https://learnku.com/docs/laravel/6.x/upgrade/5122#trans-and-trans-choice

(8)、本地化 Lang::getFromJson 方法

https://learnku.com/docs/laravel/6.x/upgrade/5122#get-from-json

(9)、队列重试限制

https://learnku.com/docs/laravel/6.x/upgrade/5122#queue-retry-limit

(10)、重发 Email 验证路由

https://learnku.com/docs/laravel/6.x/upgrade/5122#email-verification-route

(11)、Input Facade

https://learnku.com/docs/laravel/6.x/upgrade/5122#the-input-facade

--

4、从 5.8 升级到 6.0

(1)、预计升级时间：一小时

{注意} 我们尝试记录了每个可能的主要变化。由于这些主要变化中的一些是在框架的模糊部分，因此这些变化中的某些部分可能会对应用程序产生实际影响。

(2)、需要 PHP 7.2 版本

影响程度：中

PHP 7.1 版本从 2019 年十二月起将不再进行主动维护了。因此，Laravel 6.0 需要 PHP 7.2 或以上版本。

(3)、升级依赖

升级 laravel/framework 依赖于 composer.json 文件中指定的 ^6.0。

接下来，检查应用程序使用的任何第三方软件包，验证其是否适配 Laravel 6 支持的版本。

(4)、授权认证

///1/ 认证资源 & viewAny

影响程度：高；

使用 authorizeResource 方法附加到控制器中的授权策略现在应该要定义一个 viewAny 方法，当用户访问控制器中的 index 方法它就会被调用。换而言之，未经授权去访问控制器中的 index 方法将被拒绝。

//

///2/ RegisterController 控制器

影响程度：中；

如果你重写过 Laravel 框架中 RegisterController 的 register 或 registered 方法 ， 则应确保在对应方法中调用了 parent::register 和 parent::registered 方法，因为 Illuminate\Auth\Events\Registered 事件触发和新用户登录逻辑现在被移到 registered 方法中了，如果你重写了这些方法而没有调用对应的父级方法，则用户注册处理会失败。

//

///3/ 授权响应

影响程度：低；

Illuminate\Auth\Access\Response 类的构造函数参数已经改变。你应该更新相应的代码。 如果你没有手动构造过授权响应实例，只是在策略类中使用了 allow 和 deny 实例方法， 则不需要更新。

/\*\*

\* Create a new response.

\*

\* @param bool $allowed

\* @param string $message

\* @param mixed $code

\* @return void

\*/

public function \_\_construct($allowed, $message = '', $code = null)

//

///4/ Illuminate\Contracts\Auth\Access\Gate 契约

影响程度：低；

Illuminate\Contracts\Auth\Access\Gate 契约迎来了一个新的方法 inspect 。如果你正在手动实现这个接口，你应该将这个方法添加到你的实现中。

(5)、Carbon

//// Carbon 1.x 不再支持

影响程度：中；

Carbon 1.x 不再支持( https://github.com/laravel/framework/pull/28683 )，因为它的维护已经快要结束了。请将你的应用程序更新到 Carbon 2.0。

(6)、配置

//// AWS\_REGION 环境变量

影响程度：可选；

如果你打算使用 Laravel Vapor( https://vapor.laravel.com/ )，你应该将 config 目录中所有的 AWS\_REGION 更新为 AWS\_DEFAULT\_REGION 。另外，你应该在你的 .env 文件中，更新这个环境变量的名称。

(7)、Database

///1/ Capsule 类的 table 方法

影响程度：中等；

注意：此更改仅适用于使用 illuminate/database 作为依赖项的非 Laravel 应用程序。

Illuminate\Database\Capsule\Manager 类的 table 签名已更新为接受表别名作为其第二个参数。如果您在 Laravel 应用程序之外使用 illuminate/database ，则应相应地更新对此方法的任何调用。

/\*\*

\* 获得流畅的查询构建器实例

\*

\* @param \Closure|\Illuminate\Database\Query\Builder|string $table

\* @param string|null $as

\* @param string|null $connection

\* @return \Illuminate\Database\Query\Builder

\*/

public static function table($table, $as = null, $connection = null)

//

///2/ cursor 方法

影响程度：低；

cursor 方法现在返回一个 Illuminate\Support\LazyCollection 的实例，而不是一个 Generator ， LazyCollection 可以像生成器一样迭代。

$users = App\User::cursor();

foreach ($users as $user) {

//

}

(8)、Eloquent 集合

///1/ BelongsTo::update 方法

影响程度：中；

为了保持一致性，现在 BelongsTo 关系模型的 update 方法起到了临时更新查询的作用，这意味着它不提供批量赋值保护或触发 Eloquent 事件。这使得该关联关系与所有其它关联关系类型的 update 方法一致。

如果你通过 BelongsTo 更新关联关系连接的模型，并获取批量赋值更新保护和事件触发，则需要在模型自身上调用 update 方法。

// 临时查询…没有批量赋值保护或事件触发…

$post->user()->update(['foo' => 'bar']);

//

// 模型更新…提供批量赋值保护和事件触发…

$post->user->update(['foo' => 'bar']);

(9)、Arrayable 类 & toArray 方法

影响程度：中；

现在 Eloquent 集合的 toArray 方法实现了 Illuminate\Contracts\Support\Arrayable 类方法，将会把任何属性类型强制转换为数组。

(10)、主键类型的声明

影响程度：中；

Laravel 6.0 已经收到整数类型密钥的 性能优化( https://github.com/laravel/framework/pull/28153 ) 建议。如果你使用字符串作为模型的主键，那么你应该在模型属性的 $keyType 上声明密钥类型。

/\*\*

\* 主键 ID 的「类型」。

\*

\* @var string

\*/

protected $keyType = 'string';

(11)、Email 验证

///1/ 重发验证路由 HTTP 方法

影响程度：中；

为了防止可能的 CSRF 攻击，当你使用 Laravel 内置 email 验证时，系统注册的路由 email/resend 请求方式已经从 GET 变更为 POST。因此，请求此路由时，需要前端代码发送正确的请求方式。例如你可以使用如下内置的 email 验证模板。

{{ \_\_('Before proceeding, please check your email for a verification link.') }}

{{ \_\_('If you did not receive the email') }},

<form class="d-inline" method="POST" action="{{ route('verification.resend') }}">

@csrf

<button type="submit" class="btn btn-link p-0 m-0 align-baseline">

{{ \_\_('click here to request another') }}

</button>.

</form>

//

///2/ MustVerifyEmail 契约

影响程度：低；

一个新的方法 getEmailForVerification 加入到了 Illuminate\Contracts\Auth\MustVerifyEmail 契约。如果你正在手动实现这个契约，你应该实现这个方法。这个方法应该返回对象关联的电子邮箱地址。如果你的 App\User 模型正在使用 Illuminate\Auth\MustVerifyEmail 特性，不需要更改，因为这个特性会帮你实现这个方法。

(12)、辅助函数

///1/ String & Array 辅助函数包

影响程度：高；

所有的 str\_ 和 array\_ 辅助函数都已经从框架中移除，并移动到新的 laravel/helpers Composer 包中。如果需要，你可以使用 Illuminate\Support\Str 和 Illuminate\Support\Arr 类来更新这些辅助函数的所有调用。或者，你也可以在你的应用程序中添加新的 laravel/helpers 包来继续使用这些辅助函数。

composer require laravel/helpers

(13)、本地化

///1/ Lang::trans & Lang::transChoice 方法

影响程度：中；

翻译器的 Lang::trans 和 Lang::transChoice 方法已重命名为 Lang::get 和 Lang::choice 。

另外，如果你正在手动实现 Illuminate\Contracts\Translation\Translator 契约，你应该将你实现的 trans 和 transChoice 方法更新为 get 和 choice。

//

///2/ Lang::getFromJson 方法

影响程度：中；

Lang::get 和 Lang::getFromJson 方法已被合并。原先调用 Lang::getFromJson 方法的需要更改为调用 Lang::get 方法。

(14)、邮件

//// Mandrill 和 SparkPost 驱动已经被移除

影响程度：低；

mandrill 和 sparkpost 邮件驱动已经被移除，如果你想继续使用这两个驱动中的任意一个，我们鼓励你选择采用一个社区支持的包来提供驱动。

(15)、通知

//// Nexmo 路由已被移除

影响程度：低；

Nexmo 通知通道的延迟部分已经被从框架的核心中移除。如果你依赖于 Nexmo 通知路由，你应该在你的通知实体中手动的实现 routeNotificationForNexmo 方法。 详细描述请看这( https://learnku.com/docs/laravel/6.x/notifications#routing-sms-notifications )。

(16)、重置密码

//// 密码验证

影响程度：低；

PasswordBroker 不再限制或者验证密码。在 ResetPasswordController 类中密码验证已经被处理过了，这使得代理的验证显得冗余并且无法自定义。如果你在内置 ResetPasswordController 类中手动使用 PasswordBroker（或者 Password 门面），你应该在把所有密码传递给代理之前进行验证。

(17)、队列

//// 队列重试限制

影响程度：中；

在之前的 Laravel 版本 中，php artisan queue:work 命令会无限期重试队列任务，从 Laravel 6.0 开始，该命令默认只会重试队列任务一次，如果你想要强制任务无限期重试， 可以通过 --tries=0 指定。

php artisan queue:work --tries=0

另外，请确保你的数据库中包含 failed\_jobs 数据表。你可以通过 Artisan 命令 queue:failed-table 来生成这个迁移。

php artisan queue:failed-table

(17)、请求

//// Input 门面

影响程度：中；

Input 门面是 Request 门面的复制品，已经被移除。如果你在使用 Input::get 方法，应该将其替换成 Request::input 方法。所有其他调用 Input 门面的地方也同样需替换为 Request 门面。

(18)、任务调度

///1/ between 方法

影响程度：低；

在 Laravel 之前的版本中， 调度器的 between 方法出现了跨越日期边界的混乱行为。

$schedule->command('list')->between('23:00', '4:00');

对于大多数用户，上述调度预期行为是在 23:00 至 4:00 之间每分钟运行一次 list 命令。 然而， 在 Laravel 之前的版本中，调度器会在 4:00 至 23:00 之间每分钟运行一次 list 命令，刚好颠倒了时间阈值。 在 Laravel 6.0 中，这一行为已被纠正。

(19)、存储

///1/ Rackspace 存储驱动被移除

影响程度：低；

存储驱动 rackspace 已被移除。如果你想继续使用 Rackspace 作为存储驱动，我们建议您使用提供此驱动的社区所维护的扩展包。

(20)、URL 生成

///1/ 路由 URL 生成 & 提取参数

在之前版本的 Laravel 中，传递关联数组参数到 route 辅助函数或者 URL::route 方法生成指定路由（包含可选参数）对应 URL 时偶尔会出现将这些参数作为 URI 值的现象，即使传递参数值在路由路径中没有匹配键。从 Laravel 6.0 开始，这些值会被追加到查询字符串中，如下所示。

Route::get('/profile/{location?}', function ($location = null) {

//

})->name('profile');

//

// Laravel 5.8: http://example.com/profile/active

echo route('profile', ['status' => 'active']);

//

// Laravel 6.0: http://example.com/profile?status=active

echo route('profile', ['status' => 'active']);

(21)、其他

我们还鼓励您查看 laravel/laravel GitHub 仓库( https://github.com/laravel/laravel ) 代码更新日志。 尽管许多更新不是必须的，但您可能希望将这些文件与您的应用程序保持一致。其中的一些更新已经在这篇升级指南中覆盖到了，但是还有很多其他的小更新比如配置文件或注释的微调，就不会一一指出。你可以通过 GitHub comparison tool ( https://github.com/laravel/laravel/compare/5.8...master )轻松查看变更，以便选择那些对你而言更为重要的更新。

--

--

三、贡献导引

https://learnku.com/docs/laravel/6.x/contributions/5123

--

1、Bug 报告

为了鼓励积极协作，Laravel 官方强烈鼓励你拉取请求，而不仅仅是提交 Bug 报告。「Bug 报告」也可以以包含失败测试的拉取请求的形式发送。

然而，如果你提交了一个 bug 报告，你的议题（issue）应该包括关于这个议题一个标题和一个清晰的描述。你还应该包含尽可能多的相关信息以及演示该议题的代码示例。Bug 报告的目的就是让你自己和他人能够轻松地复现 bug 并修复它。

谨记，bug 报告的创建是希望和你有同样问题的他人能够与你协作解决问题，不要指望 bug 报告能够自动查看任何活动或者其他人跳转自此来修复它。创建 bug 报告有助于帮助自己和其他人开始着手解决问题。

//

//// Laravel 源码托管在 GitHub 上，每个项目都有一些仓库

// Laravel Application

https://github.com/laravel/laravel

//

// Laravel Art

https://github.com/laravel/art

//

// Laravel Documentation (开发文档项目)

https://github.com/laravel/docs

//

// Laravel Cashier

https://github.com/laravel/cashier

//

// Laravel Envoy

https://github.com/laravel/envoy

//

// Laravel Framework

https://github.com/laravel/framework

//

// Laravel Homestead

https://github.com/laravel/homestead

//

// Laravel Homestead Build Scripts

https://github.com/laravel/settler

//

// Laravel Horizon

https://github.com/laravel/horizon

//

// Laravel Passport

https://github.com/laravel/passport

//

// Laravel Scout

https://github.com/laravel/scout

//

// Laravel Socialite

https://github.com/laravel/socialite

//

// Laravel Telescope

https://github.com/laravel/telescope

//

// Laravel Website

https://github.com/laravel/laravel.com-next

--

2、核心开发讨论

你可以在 Laravel 创意 议题板( https://github.com/laravel/ideas/issues ) 中针对现有的 Laravel 行为提出新的功能或改进。如果你提出了一个新功能，我们希望你会乐于实现一些至少能够完成该功能所需的必要代码。

有关 bugs、新特性还有现存特性的实现的讨论在 Laravel Discord server( https://discordapp.com/invite/mPZNm7A ) 的 #internals 频道进行。Laravel 的维护者 Taylor Otwell 通常在工作日的上午 8 点到下午 5 点（UTC-06:00 或 America/Chicago）出现在该频道中，并在其他时间偶尔出现在该频道中。

--

3、选择哪个分支？

所有 bug 修复都应该发送到最新的稳定分支或 当前的 LTS 分支( https://learnku.com/docs/laravel/6.x/contributions/Prologue/Release-Notes?id=%E6%94%AF%E6%8C%81%E6%96%B9%E9%92%88 )。除非修复了仅在即将发布的版本中存在的功能，否则永远不应将 bug 修复发送到 master 分支。

完全向后兼容当前版本的次要功能可以发送到最新的稳定分支。

应始终将重要的新功能发送到主分支，其中包含即将发布的版本。

如果你不确定你的功能是重大功能还是次要功能，你可以在 Laravel Discord server( https://discordapp.com/invite/mPZNm7A ) 的 #internals 频道询问 Taylor Otwell。

--

4、编译资源

如果要提交的更改会影响已编译的文件，例如在 laravel/laravel 仓库中 resources/sass 或者 resources/js 目录下的大部分文件，请不要提交已编译的文件。由于这类文件尺寸过大，维护者审查他们不太实际。这会被利用来作为往 Laravel 里注入恶意代码的一种方式。为了预防性地阻止这种情况发生，Laravel 的维护者将生成并提交所有编译文件。

--

5、安全漏洞

如果你发现 Laravel 中存在安全漏洞，请发送电子邮件至 Taylor Otwell，电子邮件地址为 taylor@laravel.com，所有安全漏洞都将得到及时解决。

--

6、代码风格

Laravel 遵循 PSR-2( https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-2-coding-style-guide.md ) 编码规范和 PSR-4( https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-4-autoloader.md ) 自动加载规范。

(1)、PHPDoc

以下是正确写法的 Laravel 文档注释。请注意，@param 属性后跟两个空格、参数类型、两个空格，最后是变量名称。

/\*\*

\* 在容器中注册绑定。

\*

\* @param string|array $abstract

\* @param \Closure|string|null $concrete

\* @param bool $shared

\* @return void

\*

\* @throws \Exception

\*/

public function bind($abstract, $concrete = null, $shared = false)

{

//

}

(2)、StyleCI

别担心你的代码风格不够漂亮！在合并拉取请求后， StyleCI( https://styleci.io/ ) 将会自动把所有样式进行修正，再合并到 Laravel 存储库中。这使得我们更多的关注贡献的内容而不是代码风格。

--

--

第二卷 入门指南

--

--

一、安装

https://learnku.com/docs/laravel/6.x/installation/5124

--

1、安装

(1)、服务器要求

Laravel 对系统有一些要求。当然，所有这些要求 Laravel Homestead ( https://learnku.com/docs/laravel/6.x/homestead ) 虚拟机都能满足，因此强烈推荐你使用 Homestead 做为你的开发环境。

//

//// 当然，假如你不使用 Homestead，请确保你的服务器满足以下要求。

PHP >= 7.2.5

BCMath PHP 拓展

Ctype PHP 拓展

JSON PHP 拓展

Mbstring PHP 拓展

OpenSSL PHP 拓展

PDO PHP 拓展

Tokenizer PHP 拓展

XML PHP 拓展

(2)、安装 Laravel

Laravel 使用 Composer( https://getcomposer.org/ ) 来管理项目依赖。因此，在使用 Laravel 之前，请确保你的机器已经安装了 Composer。

//

a、通过 Laravel 安装器

///1/ 首先，通过使用 Composer 安装 Laravel 安装器

composer global require laravel/installer

确保将 Composer’s system-wide vendor 目录放置在你的系统环境变量 $PATH 中，以便系统可以找到 Laravel 的可执行文件。该目录根据你的操作系统存在不同的位置中；一些常见的配置包括 :- macOS and GNU / Linux 发行版: `$HOME/.config/composer/vendor/bin`；- Windows: `% USERPROFILE%\AppData\Roaming\Composer\vendor\bin`。

//

///2/ 再者，创建laravel项目

laravel new blog

安装完成后， laravel new 命令会在你指定的目录创建一个全新的 Laravel 项目。例如， laravel new blog 将会创建一个名为 blog 的目录，并已安装好 Laravel 所有的依赖项。

//

b、通过 Composer 创建项目

或者，你也可以在终端中运行 create-project 命令来安装 Laravel。

///1/ 创建项目

composer create-project --prefer-dist laravel/laravel blog "6.\*"

//

c、本地开发环境

如果你在本地安装了 PHP， 并且你想使用 PHP 内置的服务器来为你的应用程序提供服务，则可以使用 Artisan 命令 serve 。该命令会在 http://localhost:8000 上启动开发服务器。

//

///1/ 使用PHP内置服务器 http://localhost:8000

php artisan serve

//

当然，最好还是选择 Homestead( https://learnku.com/docs/laravel/6.x/homestead ) 和 Valet( https://learnku.com/docs/laravel/6.x/valet )。

(3)、配置

a、公共路径

安装完 Laravel 之后，你应该配置你的 web 服务的文档目录指向 public 路径。该路径下的 index.php 文件作为进入应用的所有 HTTP 请求的前端控制器。

b、配置文件

Laravel 框架的所有配置文件存放在 config 目录下。每个选项都有文档标注，便于通过文件查看并熟悉对你有用的选项。

c、目录权限

在安装 Laravel 后，你可能需要配置一些权限。 storage 和 bootstrap/cache 目录在你的 web 服务下应该是可写的权限，否则 Laravel 将无法运行。如果你用的是 Homestead( https://learnku.com/docs/laravel/6.x/homestead ) 虚拟机，这些权限应该已经设置好了。

d、应用密钥

安装好 Laravel 之后的下一步是设置你的应用密钥为随机字符串。如果你通过 Composer 或者 Laravel 安装器安装的，这个密钥已经通过 php artisan key:generate 命令为你设置好了。

通常，这个字符串应该是 32 个字符长度。这个密钥将会设置在环境变量文件 .env 中。如果你还没有将 .env.example 文件重命名为 .env 那么你现在应当把此文件重命名。 如果应用密钥还没有设置，你的用户会话和其他的加密数据将会不安全。

e、其他配置

Laravel 几乎不需要除上面所说的其他什么配置了。你可以随心所欲的开始开发了！然而，你可能会想要再次查看 config/app.php 文件和它的注释说明。它包含一些你希望根据你应用来更改的选项，诸如： timezone 和 locale。

你还可能想要配置 Laravel 的其他的一些组件，例如：- [缓存](/docs/laravel/6.x/cache#configuration)；- [数据库](/docs/laravel/6.x/database#configuration)；- [会话控制](/docs/laravel/6.x/session#configuration)；

--

2、Web 服务器配置

(1)、优雅链接

a、Apache

Laravel 中包含了一个 public/.htaccess 文件，通常用于在资源路径中隐藏 index.php 的前端控制器。在用 Apache 为 Laravel 提供服务之前，确保启用了 mod\_rewrite 模块，这样 .htaccess 文件才能被服务器解析。

//

// 如果 Laravel 附带的 .htaccess 文件不起作用，尝试下面的方法替代：

Options +FollowSymLinks -Indexes

RewriteEngine On

RewriteCond %{HTTP:Authorization} .

RewriteRule .\* - [E=HTTP\_AUTHORIZATION:%{HTTP:Authorization}]

RewriteCond %{REQUEST\_FILENAME} !-d

RewriteCond %{REQUEST\_FILENAME} !-f

RewriteRule ^ index.php [L]

//

b、Nginx

如果你使用 Nginx ，在你的站点配置中加入以下配置，所有的请求将会引导至 index.php 前端控制器。

location / {

try\_files $uri $uri/ /index.php?$query\_string;

}

当你使用 Homestead 或 Valet 时，优雅链接将会自动配置好。

--

--

二、配置信息 ( Configuration )

https://learnku.com/docs/laravel/6.x/configuration/5125

--

1、介绍

Laravel 框架的所有配置文件都保存在 config 目录中。每个选项都有说明，你可随时查看这些文件并熟悉都有哪些配置选项可供你使用。

--

2、环境配置

对于应用程序运行的环境来说，不同的环境有不同的配置通常是很有用的。 例如，你可能希望在本地使用的缓存驱动不同于生产服务器所使用的缓存驱动。

Laravel 利用 Vance Lucas 的 PHP 库 DotEnv( https://github.com/vlucas/phpdotenv ) 使得此项功能的实现变得非常简单。在新安装好的 Laravel 应用程序中，其根目录会包含一个 .env.example 文件。如果是通过 Composer 安装的 Laravel，该文件会自动更名为 .env。否则，需要你手动更改一下文件名。

你的 .env 文件不应该提交到应用程序的源代码控制系统中，因为每个使用你的应用程序的开发人员 / 服务器可能需要有一个不同的环境配置。此外，在入侵者获得你的源代码控制仓库的访问权的情况下，这会成为一个安全隐患，因为任何敏感的凭据都被暴露了。

如果是团队开发，则可能希望应用程序中仍包含 .env.example 文件。因为通过在示例配置文件中放置占位值，团队中的其他开发人员可以清楚地看到哪些环境变量是运行应用程序所必需的。你也可以创建一个 .env.testing 文件，当运行 PHPUnit 测试或以 --env=testing 为选项执行 Artisan 命令时，该文件将覆盖 .env 文件中的值。

Tip：.env 文件中的所有变量都可被外部环境变量（比如服务器级或系统级环境变量）所覆盖。

(1)、环境变量类型

.env 文件中的所有变量都被解析为字符串，因此创建了一些保留值以允许你从 env() 函数中返回更多类型的变量。

.env 值 env()值

true (bool) true

(true) (bool) true

false (bool) false

(false) (bool) false

empty (string) ''

(empty) (string) ''

null (null) null

(null) (null) null

//

如果你需要使用包含空格的值定义环境变量，可以通过将值括在双引号中来实现。

APP\_NAME="我的 应用"

(2)、检索环境配置

当应用程序收到请求时，.env 文件中列出的所有变量将被加载到 PHP 的超级全局变量 $\_ENV 中。你可以使用 env 函数检索这些变量的值。事实上，如果你查看 Laravel 的配置文件，你就能注意到有数个选项已经使用了这个函数。

'debug' => env('APP\_DEBUG', false),

传递给 env 函数的第二个值是「默认值」。如果给定的键不存在环境变量，则会使用该值。

(3)、确定当前环境

应用程序当前所处环境是通过 .env 文件中的 APP\_ENV 变量确定的。你可以通过 App facade 中的 environment 方法来访问此值。

$environment = App::environment();

//

你还可以传递参数给 environment 方法，以检查当前的环境配置是否与给定值匹配。 如果与给定的任意值匹配，该方法将返回 true。

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

// 当前环境是 local

}

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

// 当前的环境是 local 或 staging...

}

Tip：应用程序当前所处环境检测可以被服务器级的 APP\_ENV 环境变量覆盖。这在为相同的应用程序配置不同的环境时是非常有用的，这样你可以在你的服务器配置中为给定的主机设置与其匹配的给定的环境。

(4)、在调试页面隐藏环境变量

当一个异常未被捕获并且 APP\_DEBUG 环境变量为 true 时，调试页面会显示所有的环境变量和内容。在某些情况下你可能想隐藏某些变量。你可以通过设置 config/app.php 配置文件中的 debug\_blacklist 选项来完成这个操作。

环境变量、服务器或者请求数据中都有一些变量是可用的。因此，你可能需要将 $\_ENV 和 $\_SERVER 的变量加入到黑名单中。

return [

// ...

'debug\_blacklist' => [

'\_ENV' => [

'APP\_KEY',

'DB\_PASSWORD',

],

'\_SERVER' => [

'APP\_KEY',

'DB\_PASSWORD',

],

'\_POST' => [

'password',

],

],

];

--

3、访问配置值

你可以轻松地在应用程序的任何位置使用全局 config 函数来访问配置值。配置值的访问可以使用「点」语法，这其中包含了要访问的文件和选项的名称。还可以指定默认值，如果配置选项不存在，则返回默认值。

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

//

要在运行时设置配置值，传递一个数组给 config 函数。

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

--

4、配置缓存

为了给你的应用程序提升速度，你应该使用 Artisan 命令 config:cache 将所有的配置文件缓存到单个文件中。这会把你的应用程序中所有的配置选项合并成一个单一的文件，然后框架会快速加载这个文件。

通常来说，你应该把运行 php artisan config:cache 命令作为生产环境部署常规工作的一部分。这个命令不应在本地开发环境下运行，因为配置选项在应用程序开发过程中是经常需要被更改的。

注意：如果在部署过程中执行 config:cache 命令，那你应该确保只从配置文件内部调用 env 函数。一旦配置被缓存，.env 文件将不再被加载，所有对 env 函数的调用都将返回 null。

当应用程序处于维护模式时，所有对应用程序的请求都显示为一个自定义视图。这样可以在更新或执行维护时轻松地「关闭」你的应用程序。 维护模式检查包含在应用程序的默认中间件栈中。如果应用程序处于维护模式，则将抛出一个状态码为 503 的 MaintenanceModeException 异常。

//

// 要启用维护模式，只需执行下面的 Artisan 的 down 命令

php artisan down

//

你还可以向 down 命令提供 message 和 retry 选项。其中 message 选项的值可用于显示或记录自定义消息，而 retry 值可用于设置 HTTP 请求头中 Retry-After 的值。

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

//

即使在维护模式下，也可以使用命令 allow 选项允许特定的 IP 地址或网络访问应用程序。

php artisan down --allow=127.0.0.1 --allow=192.168.0.0/16

//

要关闭维护模式，请使用 up 命令。

php artisan up

Tip：你可以通过修改 resources/views/errors/503.blade.php 模板文件来自定义默认维护模式模板。

--

5、维护模式

(1)、维护模式 & 队列

当应用程序处于维护模式时，不会处理 队列任务( https://learnku.com/docs/laravel/6.x/queues )。而这些任务会在应用程序退出维护模式后再继续处理。

(2)、维护模式的替代方案

维护模式会导致应用程序有数秒的停机（不响应）时间，因此你可以考虑使用像 Envoyer( https://envoyer.io/ ) 这样的替代方案，以便与 Laravel 完成零停机时间部署。

--

--

三、文件夹结构

https://learnku.com/docs/laravel/6.x/structure/5126

--

1、简介

默认的 Laravel 应用结构旨在为不同大小的应用提供一个很好的起点。当然，您可以随意组织您的应用程序。Laravel 对任何给定类的位置几乎没有任何限制，只要它们能被 Composer 自动加载。

(1)、为什么没有模型目录？

当开始使用 Laravel 时，许多开发人员都因缺少 models 目录而感到困惑。然而，缺少这样的目录是故意的。我们发现「模型」含糊不清，因为不同的人对「模型」有不同的理解。一些开发者把应用的「模型」称为其所有业务逻辑的总体，而另一些人将「模型」称为与关系数据库交互的类。

出于这个原因，我们默认把 Eloquent 的模型放在 app 目录下，并允许开发人员将它们放在其他地方。

--

2、根目录

(1)、app目录

app 目录包含应用程序的核心代码。你应用中几乎所有的类都应该放在这里。稍后我们会更深入地了解这个目录的细节。

(2)、bootstrap目录

bootstrap 目录包含引导框架的 app.php 文件。该目录还包含了一个 cache 目录， cache 目录下存放着框架生成的用来提升性能的文件，比如路由和服务缓存文件。

(3)、config 目录

config 目录，顾名思义，包含应用程序所有的配置文件。我们鼓励你通读这些文件，以便帮助你熟悉所有可用的选项。

(4)、database目录

database 目录包含数据填充和迁移文件以及模型工厂类。你还可以把它作为 SQLite 数据库存放目录。

(5)、public目录

public 目录包含了入口文件 index.php ，它是进入应用程序的所有请求的入口点。此目录还包含了一些你的资源文件（如图片、JavaScript 和 CSS）。

(6)、resources目录

resources 目录包含了视图和未编译的资源文件（如 LESS、SASS 或 JavaScript）。此目录还包含你所有的语言文件。

(7)、routes目录

routes 目录包含了应用的所有路由定义，Laravel 默认包含了几个路由文件：web.php、api.php、 console.php 和 channels.php。

web.php 文件包含 RouteServiceProvider 放置在 web 中间件组中的路由，它提供会话状态、CSRF 防护和 cookie 加密。如果你的应用不提供无状态的、RESTful 风格的 API，则所有的路由都应该在 web.php 文件中定义。

api.php 文件包含 RouteServiceProvider 放置在 api 中间件组中的路由，它提供了频率限制。这些路由都是无状态的，所以通过这些路由进入应用请求旨在通过令牌进行身份认证，并且不能访问会话状态。

console.php 文件是定义所有基于控制台命令闭包函数的地方。每个闭包函数都被绑定到一个命令实例并且允许和命令行 IO 方法进行简单的交互。尽管这些文件没有定义 HTTP 路由，但它也将基于控制台的入口点（路由）定义到应用程序中。

channels.php 用来注册你的应用支持的所有的事件广播渠道的地方。

(8)、storage目录

storage 目录包含编译后的 Blade 模板、session 会话生成的文件、缓存文件以及框架生成的其他文件。这个目录被细分成 app 、 framework 和 logs 三个子目录。app 目录可以用来存储应用生成的任何文件。 framework 目录用来存储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 可以用来存储用户生成的文件，比如需要公开访问的用户头像。你应该创建一个 public/storage 的软链接指向这个目录。你可以直接通过 php artisan storage:link 命令来创建此链接。

(9)、tests目录

tests 目录包含自动化测试文件。在 PHPUnit( https://phpunit.de/ ) 有现成的范例供你参考。每个测试类都应该以 Test 作为后缀。你可以使用 phpunit 或者 php vendor/bin/phpunit 命令来运行测试。

(10)、vendor目录

vendor 目录包含你所有的 Composer( https://getcomposer.org/ ) 依赖包。

3、App 目录

你的大部分应用程序都位于 app 目录中。默认情况下，此目录的命名空间为 App, 并通过 Composer 使用 PSR-4 自动加载标准( https://www.php-fig.org/psr/psr-4/ ) 自动加载。

app 目录包含额外的各种目录，比如：Console, Http, 和 Providers。将 Console 和 Http 目录视为向应用程序的核心提供 API。HTTP 协议和 CLI 都是与应用程序交互的机制，但实际上并不包含应用程序逻辑。换句话说，它们是向你的应用程序发出命令的两种方式。Console 包含所有的 Artisan 命令，而 Http 目录包含你的控制器，中间件和请求。

当你使用 make Artisan 命令生成类时，将在 app 目录下生成各种其它目录。因此，例如，app/Jobs 目录直到你执行 make:job Artisan 命令去生成一个作业类之前将不存在。

Tip: 许多类通过 Artisan 命令生成在 app 目录中。为了查看可用的命令，在你的终端运行 php artisan list make 命令。

(1)、Broadcasting目录

此目录默认是不存在的，但是当你创建第一个通道时它将被创建。要了解更多的关于频道的信息，查看有关文档 事件广播( https://learnku.com/docs/laravel/6.x/broadcasting )。

(2)、Console目录

Console 目录包含应用程序的所有自定义 Artisan 命令。这些命令可以使用 make:command 命令生成。此目录也安置了控制台内核，在其中你可以注册自定义的 Artisan 命令，并定义 计划任务( https://learnku.com/docs/laravel/6.x/scheduling )。

(3)、Events目录

默认情况下这个目录是不存在的，但你可以通过 event:generate 和 make:event Artisan 命令去创建。Events 目录安置 事件类( https://learnku.com/docs/laravel/6.x/events )。事件可用于提醒您的应用程序其它部分已发生给定操作，提供了极大的灵活性和解耦。

(4)、Exceptions目录

Exceptions 目录包含应用程序的异常处理，并且也是一个放置应用程序抛出任何异常的好地方。如果你想自定义异常的记录和渲染方式，你应该修改此目录中的 Handler 类。

(5)、Http目录

Http 目录包含你的控制器，中间件和表单请求。处理进入应用程序请求的所有逻辑几乎都放置在此目录。

(6)、Jobs目录

默认情况下这个目录是不存在的，但如果你执行 make:job Artisan 命令时，它将被创建出来。Jobs 目录安置应用程序的 队列任务( https://learnku.com/docs/laravel/6.x/queues )。Jobs 可由应用程序排队作业，也可以在当前请求的生命周期内同步运行。在当前请求期间同步运行的 Jobs 有时会称为『命令』，因为它们是 命令模式( https://en.wikipedia.org/wiki/Command\_pattern ) 的一个实现。

(7)、Listeners目录

默认情况下这个目录是不存在的，但如果你执行了 event:generate 或者 make:listener Artisan 命令时，它将会被创建出来。Listeners 目录包含 事件( https://learnku.com/docs/laravel/6.x/events ) 的处理类。事件侦听器接收一个事件实例并执行逻辑以响应被触发的事件。例如，一个 UserRegistered 事件可能被 SendWelcomeEmail 侦听器处理。

(8)、Mail目录

默认情况下这个目录是不存在的，但如果你执行 make:mail Artisan 命令，它将被创建出来。Mail 目录包含应用程序发送邮件的所有类。邮件对象允许你去构建一个封装所有逻辑的邮件类，在这个类中可以使用 Mail::send 方法发送邮件。

(9)、Notifications目录

默认情况下这个目录是不存在的，但如果你执行 make:notification Artisan 命令，它将被创建出来。Notifications 目录包含应用程序的发送的所有『事务』通知，比如关于应用程序中发生的事件的简单通知。Laravel 的通知功能抽象了通过各种驱动（如：电子邮件，Slack，SMS 或者存储在数据库中）去发送通知。

(10)、Policies目录

默认情况下这个目录是不存在的，但如果你执行 make:policy Artisan 命令，它将被创建出来。Policies 目录包含应用程序的授权策略类。策略用于确定一个用户是否对一个资源能否执行一个给定的操作。有关更多信息，查看 授权文档( https://learnku.com/docs/laravel/6.x/authorization )。

(11)、Providers目录

Providers 目录包含应用程序的所有 服务提供者( https://learnku.com/docs/laravel/6.x/providers )。服务提供者通过在服务容器中绑定服务引导应用程序，注册事件或者准备为应用程序即将到来的请求执行其它任何任务。

在一个新的 Laravel 应用中，此目录已经包含一些提供者。你可以根据需要随意将你自己的提供者添加到此目录中。

(12)、Rules目录

默认情况下这个目录是不存在的，但如果你执行 make:rule Artisan 命令，它将被创建出来。Rules 目录包含应用程序的自定义验证规则对象。规则用于将复杂的验证逻辑封装在一个简单对象中。关于更多信息，查看 验证文档( https://learnku.com/docs/laravel/6.x/validation )。

--

--

四、Homestead

https://learnku.com/docs/laravel/6.x/homestead/5127

--

1、介绍

Laravel 致力于让整个 PHP 开发体验变的更愉快，包括你的本地开发环境。Vagrant( https://www.vagrantup.com/ ) 提供了一种简单、优雅的方式来管理和配置虚拟主机。

Laravel Homestead 是一个官方预封装的 Vagrant box ，它为你提供了一个完美的开发环境，你不需要在本地机器安装 PHP、web 服务器和其他的服务器软件。你再也不用担心会弄乱你的操作系统了！Vagrant boxes 是一次性的。 如果出现问题， 你可以在几分钟内销毁并重新创建 box!

Homestead 可以运行在任何 Windows、Mac、或 Linux 系统， 它包括了 Nginx web 服务器，PHP 7.3、PHP 7.2、PHP 7.1、MySQL、PostgreSQL、Redis、Memcached、Node 以及你开发 Laravel 所需要的东西。

注意：如果你使用的是 Windows, 你可能需要通过 BIOS 来启用硬件虚拟化 (VT-x)。如果你在 Hyper-V 系统上使用 UEFI 你可能还需要禁用 Hyper-V 以便访问 VT-x。

(1)、内置软件

- Ubuntu 18.04 - Git - PHP 7.3 - PHP 7.2 - PHP 7.1 - PHP 7.0 - PHP 5.6 - Nginx - MySQL - lmm for MySQL or MariaDB database snapshots - Sqlite3 - PostgreSQL - Composer - Node (With Yarn, Bower, Grunt, and Gulp) - Redis - Memcached - Beanstalkd - Mailhog - avahi - ngrok - Xdebug - XHProf / Tideways / XHGui - wp-cli。

(2)、可选软件

- Apache - Blackfire - Cassandra - Chronograf - CouchDB - Crystal & Lucky Framework - Docker - Elasticsearch - Gearman - Go - Grafana - InfluxDB - MariaDB - MinIO - MongoDB - MySQL 8 - Neo4j - Oh My Zsh - Open Resty - PM2 - Python - RabbitMQ - Solr - Webdriver & Laravel Dusk Utilities。

--

2、安装与设置

(1)、第一步

在你启动 Homestead 环境之前，你必须安装 VirtualBox 6.x( https://www.virtualbox.org/wiki/Downloads )，VMWare( https://www.vmware.com/ )，Parallels( https://www.parallels.com/products/desktop/ ) 或 Hyper-V( https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/quick-start/enable-hyper-v ) 以及 Vagrant( https://www.vagrantup.com/downloads.html )。以上所有软件均针对不同的操作系统提供了易于使用的可视化安装包。

若要使用 VMware 提供器，你需要购买 VMware Fusion / Workstation 和 VMware Vagrant plug-in( https://www.vagrantup.com/vmware )。 虽然它不是免费的，但是 VMware 可以提供更快的共享文件夹性能。

若要使用 Parallels 提供器，你需要安装 Parallels Vagrant plug-in( https://github.com/Parallels/vagrant-parallels )。它是免费的。

由于受到 Vagrant 限制( https://www.vagrantup.com/docs/hyperv/limitations.html )，Hyper-V 提供程序会忽略所有的网络设置。

//

///1/ 安装 Homestead Vagrant Box

一旦将 VirtualBox / VMware 和 Vagrant 都安装好之后，你可以在终端执行下面的命令将 laravel/homestead box 添加到 Vagrant 中安装。你可能需要几分钟的时间来下载 box，因为它取决于你的网络连接速度。

vagrant box add laravel/homestead

如果这个命令失败了，请确保你安装的是最新版的 Vagrant。

注意：Homestead 会定期发出「alpha」/「beta」box 用于测试，这可能会影响 vagrant box add 命令。如果你在运行 vagrant box add 时遇到问题，可以运行 vagrant up 命令，当 Vagrant 尝试启动虚拟机时，将下载正确的 box。

//

///2/ 安装 Homestead

你可以通过克隆代码来安装 Homestead。建议将代码克隆到你的「home」目录下的 Homestead 文件夹中，这样 Homestead box 就可以作为你的所有 Laravel 项目的主机.

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

因为 Homestead 的 master 分支并不是稳定的，你应该使用打过标签的稳定版本。您可以在 GitHub Release Page( https://github.com/laravel/homestead/releases ) 上找到最新的稳定版。或者，你可以查看包含最新稳定版本的 release 分支。

cd ~/Homestead

git checkout release

一旦克隆 Homestead 代码完成以后，在 Homestead 目录中使用 bash init.sh 命令来创建 Homestead.yaml 配置文件。Homestead.yaml 文件将被放在 Homestead 目录中。

// Mac / Linux...

bash init.sh

// Windows...

init.bat

(2)、配置 Homestead

///1/ 设置你的提供器

Homestead.yaml 文件中的 provider 参数决定了你用的是哪一个 Vagrant 提供器 : virtualbox, vmware\_fusion, vmware\_workstation, parallels 以及 hyperv。你可以根据你的喜好来设置他们。

provider: virtualbox

//

///2/ 配置共享文件夹

Homestead.yaml 文件的 folders 属性里列出了所有与 Homestead 环境共享的文件夹。这些文件夹中的文件如果发生变更，它们会保持本地机器与 Homestead 环境之间同步。你可以根据需要配置多个共享文件夹。

folders:

- map: ~/code/project1

to: /home/vagrant/project1

注意：Windows 用户不应使用 〜/ 路径语法，而应使用其项目的完整路径，例如 C:\Users\user\Code\project1。

你应始终将各个项目映射到自己的文件夹映射，而不是映射整个 〜/ code 文件夹。映射文件夹时，虚拟机必须跟踪文件夹中 每个 文件的所有磁盘 IO。如果文件夹中有大量文件，则会导致性能问题。

folders:

- map: ~/code/project1

to: /home/vagrant/project1

- map: ~/code/project2

to: /home/vagrant/project2

注意：使用 Homestead 时，不应该挂载到 .（当前目录）。这会导致 Vagrant 不将当前文件夹映射到 /vagrant，并且会破坏可选功能并在配置时导致意外结果。

若要开启 NFS( https://www.vagrantup.com/docs/synced-folders/nfs.html )，你只需要在共享的文件夹配置中添加一个简单的标志。

folders:

- map: ~/code/project1

to: /home/vagrant/project1

type: "nfs"

注意：当使用 NFS 时，你最好使用 vagrant-winnfsd( https://github.com/winnfsd/vagrant-winnfsd ) 扩展插件。这个插件会替你处理 Homestead box 中的文件或目录权限的问题。

// 你也可以通过在 options 中列出 Vagrant 的 同步文件夹 支持的任何选项。

folders:

- map: ~/code/project1

to: /home/vagrant/project1

type: "rsync"

options:

rsync\_\_args: ["--verbose", "--archive", "--delete", "-zz"]

rsync\_\_exclude: ["node\_modules"]

//

///3/ 配置 Nginx 站点

对 Nginx 不熟悉？没关系。 sites 功能可以让你在 Homestead 上轻松的映射一个” 域名” 到一个文件夹。在 Homestead.yaml 文件中包含了一个简单的站点配置示例。同样，您可以根据需要为您的 Homestead 环境添加很多的站点。Homestead 可以为你正在开发的每个 Laravel 项目提供一个方便的虚拟化环境。

sites:

- map: homestead.test

to: /home/vagrant/project1/public

如果你在 Homestead 虚拟机启动后更改了 sites 选项，你需要再次运行 vagrant reload --provision 命令去更新虚拟机上的 Nginx 配置。

注意：Homestead 脚本被设计为尽可能保证操作的幂等，不过，如果你在 provison 过程中还是出现问题，则需要通过 vagrant destroy && vagrant up 命令来销毁并重构虚拟机。

//

///4/ 主机名解析

Homestead 通过 mDNS 发布主机名来自动解析主机。如果你在 Homestead.yaml 中设置了 hostname: homestead，则该主机会以 homestead.local 的形式生效。MacOS、iOS 和 Linux 桌面系统默认提供了对 mDNS 的支持，Windows 需要安装 Bonjour Print Services for Windows( https://support.apple.com/kb/DL999?viewlocale=zh\_CN&locale=en\_US ) 使其生效。

自动主机名在每个项目独立安装的 Homestead 中工作最好，如果你在一个 Homestead 实例上托管了多个站点，可以在宿主机上将站点「域名」添加到 hosts 文件中做域名映射，hosts 文件会将针对 Homestead 站点的请求重定向到 Homestead 虚拟机。在 Mac 和 Linux 系统中，该文件位于 /etc/hosts，在 Windows 系统中，该文件位于 C:\Windows\System32\drivers\etc\hosts。我们以 homestead.test 域名映射为例，添加到 hosts 文件的记录如下所示：192.168.10.10 homestead.test。

确保 IP 地址和你的 Homestead.yaml 文件中的配置项一致，一旦你将域名添加到 hosts 文件中并启动 Homestead，就可以在浏览器中通过该域名访问站点了：http://homestead.test。

(3)、启动 Vagrant Box

根据你的需求编辑完成 Homestead.yaml，在你的 Homestead 文件夹中运行 vagrant up 命令。Vagrant 将启动虚拟机并自动配置你的共享文件夹和 Nginx 站点。

若要删除虚拟机，只需要运行 vagrant destroy --force 命令。

(4)、根据项目安装

除了全局安装 Homestead 并且在所有项目共享相同的 Homestead box 之外， 你可以为每个项目配置 Homestead 实例。 通过在项目下创建 Vagrantfile，其他的项目成员只需运行 vagrant up 就能拥有相同的开发环境。

要将 Homestead 直接安装到项目中，需要使用 Composer 命令：composer require laravel/homestead --dev。

Homestead 安装之后， 使用 make 命令在项目根目录中生成 Vagrantfile 和 Homestead.yaml 文件。 make 命令会自动配置 Homestead.yaml 文件中 sites 和 folders 指令。

//

// Mac / Linux

php vendor/bin/homestead make

//

// Windows

vendor\\bin\\homestead make

接下来，在命令行中运行 vagrant up 命令，然后在浏览器中访问 http://homestead.test 。 记住，如果你没有使用自动 主机名解析( https://learnku.com/docs/laravel/6.x/homestead/5127#hostname-resolution )，在访问之前你仍然需要在 /etc/hosts 文件中添加你的 homestead.test 记录或者其他的域名。

(5)、安装可选功能

安装可选功能可以在 Homestead 配置文件 Homestead.yaml 中使用 features 设置来实现，大部分功能只需要通过布尔值设置启用或者禁用来选择安装或者不安装，有些功能还允许配置多个选项。

features:

- blackfire:

server\_id: "server\_id"

server\_token: "server\_value"

client\_id: "client\_id"

client\_token: "client\_value"

- cassandra: true

- chronograf: true

- couchdb: true

- crystal: true

- docker: true

- elasticsearch:

version: 7

- gearman: true

- golang: true

- grafana: true

- influxdb: true

- mariadb: true

- minio: true

- mongodb: true

- mysql8: true

- neo4j: true

- ohmyzsh: true

- openresty: true

- pm2: true

- python: true

- rabbitmq: true

- solr: true

- webdriver: true

//

//// MariaDB

启用 MariaDB 将会移除 MySQL 并安装 MariaDB。MariaDB 是 MySQL 的替代品，完全兼容 MySQL，所以在应用的数据库配置中你仍然可以使用 mysql 驱动。

//

//// MongoDB

默认安装的 MongoDB 将会设置数据库用户名为 homestead 及对应的密码为 secret。

//

//// 如果你要安装 Elasticsearch，你可以在 Homestead.yaml 文件中添加 elasticsearch 选项并指定支持的版本号。可以仅包含主版本，也可以是某个具体的版本号（major.minor.patch）。默认安装会创建一个名为「homestead」的集群。 注意永远不要赋予 Elasticsearch 超过操作系统一半的内存，因此请保证你的 Homestead 至少分配了两倍于 Elasticsearch 的内存。

提示：你可以查看 Elasticsearch documentation( https://www.elastic.co/guide/en/elasticsearch/reference/current ) 文档学习如何自定义你的配置。

//

//// Neo4j

默认安装的 Neo4j 会设置数据库用户名为 homestead 及对应的密码 secret。要查看 Neo4j，可以在浏览器中访问 http://homestead.test:7474，Neo4j 对外提供了三个端口：7687（Bolt）、7474（HTTP）、7473（HTTPS） 用于从客户端访问。

(6)、别名

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

alias c='clear'

alias ..='cd ..'

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

3、日常使用

(1)、全局可用的 Homestead

有时你可能想在文件系统的任意路径都能运行 vagrant up 命令启动 Homestead 虚拟机。在 Mac / Linux 系统上可以在 Bash 配置文件（ ~/.bash\_profile ）中添加 Bash 函数 。在 Windows 上， 你可以添加「批处理」文件到 PATH 。这些脚本允许你在系统的任何位置运行 Vagrant 命令并自动将该命令指向 Homestead 的安装路径。

//

//// Mac / Linux

function homestead() {

( cd ~/Homestead && vagrant $\* )

}

确保将该函数中的 ~/Homestead 路径调整为实际的 Homestead 安装路径。这样你就可以在系统的任意位置运行 homestead up 或 homestead ssh 等命令。

//

//// Windows

在系统的任意位置创建一个批处理文件 homestead.bat ，内容如下。

～～start～～～

@echo off

set cwd=%cd%

set homesteadVagrant=C:\Homestead

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

cd /d %cwd%

set cwd=

set homesteadVagrant=

～～end～～～

一定要将脚本中 C:\Homestead 路径调整为 Homestead 的实际安装路径。创建文件后，将文件路径添加到 PATH 。然后你就可以在系统任意位置运行 homestead up 或 homestead ssh 等命令。

(2)、通过 SSH 连接

你可以通过在 Homestead 目录下运行 vagrant ssh 终端命令以 SSH 的方式连接到虚拟机。

但是，你可能需要频繁连接 Homestead 虚拟机，因此请考虑将上面的「函数」添加到主机以便快速地以 SSH 方式连接到 Homestead 虚拟机。

(3)、连接数据库

homestead 默认已经在虚拟机中为 MySQL 和 PostgreSQL 数据库做好了配置。要从主机的数据库客户端连接到 MySQL 或 PostgreSQL ，应该连接到 127.0.0.1 ，端口 33060 （MySQL） 或 54320 （PostgreSQL）。用户名和密码分别是 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。而 Laravel 在虚拟机中 运行时还是应该使用默认的 3306 和 5432 端口进行数据库连接。

(4)、数据库备份

当你的 Vagrant 盒子销毁时，Homestead 能自动备份你的数据库。为了利用这个功能，你必须使用 Vagrant 2.1.0 或者更高的版本。或者，如果你使用一个旧版本的 Vagrant，你必须安装 vagrant-triggers 插件。要开启自动数据库备份，需添加如下的行到你的 Homestead.yaml 文件：backup: true。

一旦配置了，当 vagrant destroy 命令被执行时，Homestead 将导出你的数据库到 mysql\_backup 和 postgres\_backup 目录。如果你使用 每个项目安装( https://learnku.com/docs/laravel/6.x/homestead/5127#per-project-installation ) 方法，则可以在克隆的 Homestead 或者在你的项目根目录中找到这些目录。

(5)、数据库快照

Homestead 支持保存 MySQL 和 MariaDB 数据库的状态并通过 Logical MySQL Manager 在不同状态间进行切换。例如，假设你的站点数据库有几个 G 的数据量，你可以导入这个数据库并保存一份快照，在本地工作一段时间后有可能会创建一些新的测试内容，你可以通过快照快速恢复到最初的状态。

在底层，LMM 使用了 LVM 的支持写时复制的瘦快照功能，这意味着当修改表中某条记录时，只会将你所做的更改写入磁盘，从而在恢复期节省大量时间和磁盘空间。

由于 lmm 会与 LVM 进行交互，所以必须以 root 身份运行。要了解所有命令，可以在 Homestead 虚拟机中通过 sudo lmm 查看。常见的工作流会是这样：导入数据库到 lmm 默认的 master 分支；运行 sudo lmm branch prod-YYYY-MM-DD 保存尚未做任何修改的数据库快照；修改数据库记录；运行 sudo lmm merge prod-YYYY-MM-D 撤销所有修改；运行 sudo lmm delete <branch> 删除不需要的分支。

(6)、添加其他站点

一旦你的 Homestead 环境已配置并运行，你可以为你的 Laravel 应用添加额外的 Nginx 站点。你可能希望在单个 Homestead 环境运行多个 Laravel 安装。要添加额外的站点，到 Homestead.yaml 文件中添加站点。

sites:

- map: homestead.test

to: /home/vagrant/project1/public

- map: another.test

to: /home/vagrant/project2/public

如果 Vagrant 不能自动管理你的『hosts』文件，你可能还需要添加新的站点到该文件中。

192.168.10.10 homestead.test

192.168.10.10 another.test

一旦这个站点被添加，从你的 Homestead 目录运行 vagrant reload --provision 命令。

//

//// 站点类型

Homestead 支持多种类型的站点，允许你轻松运行不是基于 Laravel 的项目。例如，我们可以使用 symfony2 站点类型轻松地添加一个 Symfony 应用到 Homestead。

sites:

- map: symfony2.test

to: /home/vagrant/my-symfony-project/web

type: "symfony2"

可用的站点类型是：apache，apigility，expressive，laravel（默认），proxy，silverstripe，statamic，symfony2，symfony4，和 zf。

//

//// 站点参数

你可以通过 params 站点指令添加额外的 Nginx fastcgi\_param 值到你的站点。例如，我们可以添加一个 FOO 参数，对应参数值是 BAR。

sites:

- map: homestead.test

to: /home/vagrant/project1/public

params:

- key: FOO

value: BAR

(7)、环境变量

你能通过添加如下的值到 Homestead.yaml 文件来设置全局环境变量。

variables:

- key: APP\_ENV

value: local

- key: FOO

value: bar

更新 Homestead.yaml 文件之后，确保通过运行 vagrant reload --provision 命令重新配置机器。这个将更新所有安装 的 PHP 版本的 PHP-FPM 配置并为 vagrant 用户更新环境。

(8)、配置 Cron 调度器（配置 Cron 定时计划）

Laravel 提供了一种 定时计划作业（ https://learnku.com/docs/%7B%7Bversion%7D%7D/scheduling ） 的方式，只需每分钟运行一次 schedule:run Artisan 命令即可。schedule:run 命令将检查在你在 App\Console\Kernel 类中定义的计划来决定运行哪个作业。

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.test

to: /home/vagrant/project1/public

schedule: true

站点的 Cron 作业将被定义在虚拟机的 /etc/cron.d 目录中。

(9)、配置 Mailhog

Mailhog 允许你轻松地捕获外发的电子邮件并进行检查它，而实际上没有将发送邮件给收件人。开始使用时，使用以下的邮件设置更新你的.env 文件。

MAIL\_DRIVER=smtp

MAIL\_HOST=localhost

MAIL\_PORT=1025

MAIL\_USERNAME=null

MAIL\_PASSWORD=null

MAIL\_ENCRYPTION=null

一旦 Mailhog 被配置，你可以在 http://localhost:8025 上访问 Mailhog 控制面板。

(10)、配置 Minio

Minio 是一个开源的对象存储服务器，具有与 Amazon S3 兼容的 API。要安装 Minio，用以下的配置选项更新 Homestead.yaml 文件中的 features 项：minio: true。

默认情况下，Minio 在端口 9600 上是可用的。你可以通过访问 http://homestead:9600 进入 Minio 控制面板。默认的 access key 是 homestead，默认的 secret key 是 secretkey，访问 Minio 时，需要使用 us-east-1 区域。

为了使用 Minio，你将需要在你的 config/filesystems.php 配置文件中调整 S3 磁盘配置。你将需要添加 use\_path\_style\_endpoint 选项到磁盘配置，并将 url 键更改为 endpoint。

's3' => [

'driver' => 's3',

'key' => env('AWS\_ACCESS\_KEY\_ID'),

'secret' => env('AWS\_SECRET\_ACCESS\_KEY'),

'region' => env('AWS\_DEFAULT\_REGION'),

'bucket' => env('AWS\_BUCKET'),

'endpoint' => env('AWS\_URL'),

'use\_path\_style\_endpoint' => true

]

最后，确保你的 .env 文件有如下的选项。

AWS\_ACCESS\_KEY\_ID=homestead

AWS\_SECRET\_ACCESS\_KEY=secretkey

AWS\_DEFAULT\_REGION=us-east-1

AWS\_URL=http://homestead:9600

为了配置 buckets，要添加 buckets 指令到 Homestead 配置文件。

buckets:

- name: your-bucket

policy: public

- name: your-private-bucket

policy: none

支持的 policy 值包括：none，download，upload 和 public。

(11)、端口

默认情况下，如下的端口将转发到你的 Homestead 环境。

SSH：2222 -> 转发到 22

ngrok UI：4040 -> 转发到 4040

HTTP：8000 -> 转发到 80

HTTPS：44300 -> 转发到 443

MySQL：33060 -> 转发到 3306

PostgreSQL：54320 -> 转发到 5432

MongoDB：27017 -> 转发到 27017

Mailhog：8025 -> 转发到 8025

Minio：9600 -> 转发到 9600

//

// 转发额外的端口

如果你愿意，你可以转发额外的端口到 Vagrant 盒子，同时也指定他们的协议。

ports:

- send: 50000

to: 5000

- send: 7777

to: 777

protocol: udp

(12)、共享你的环境

有时你希望跟同事或客户共享你当前工作的内容。Vagrant 有内置的方法通过 vagrant share 去支持这个；然而，如果在 Homestead.yaml 文件中有多个站点配置，这个将不会工作。

为了解决这个问题，Homestead 包含它自己的 share 命令。开始时，SSH 通过 vagrant ssh 进入你的 Homestead 机器并运行 share homestead.test。这个将从你的 Homestead.yaml 配置文件中分享 homestead.test 站点。当然，你可以将任何其它配置的站点替换为 homestead.test：share homestead.test。

运行此命令之后，你将看到一个 Ngrok 屏幕出现，其中包含了活动日志和共享站点的可访问的 URLs。如果你想指定一个自定义的区域，子域，或者其它 Ngrok 运行时选项，你可以添加他们到你的 share 命令：share homestead.test -region=eu -subdomain=laravel。

注意：记住，Vagrant 本质上是不安全的，并且在运行 share 命令时将虚拟机显露到互联网。

(13)、多个 PHP 版本（多版本 PHP）

Homestead 6 在同一个虚拟机上引入了对多个 PHP 版本的支持。你可以在你的 Homestead.yaml 文件中指定要用于给定站点的 PHP 版本。可用的 PHP 版本是：『5.6』，『7.0』，『7.1』，『7.2』和『7.3』（默认）。

sites:

- map: homestead.test

to: /home/vagrant/code/my-project/public

php: "7.1"

另外，你可以通过 CLI 使用任何支持的 PHP 版本。

php5.6 artisan list

php7.0 artisan list

php7.1 artisan list

php7.2 artisan list

php7.3 artisan list

还可以在虚拟机中通过如下这些命令来更新默认的 CLI 版本：php56、php70、php71、php72、php73。

(14)、Web 服务器

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

(15)、邮件

Homestead 包含了 Postfix 邮件转发代理，默认监听 1025 端口，所以，你可以指定你的应用在 localhost 端口 1025 上使用 smtp 邮件驱动。这样，所有发送的邮件将由 Postfix 处理并由 Mailhog 捕获。要查看已发送的邮件，可以在浏览器打开 http://localhost:8025/。

--

4、调试与分析

(1)、使用 Xdebug 调试 Web 请求

Homestead 引入了 Xdebug（ https://xdebug.org/ ） 提供对逐步调试的支持。例如，你可以从浏览器加载某个页面，此时 PHP 会连接到你的 IDE 以便允许对运行代码进行检查和修改。

默认情况下，Xdebug 已经运行并且准备好接收连接，如果你需要在 CLI 中启用 Xdebug，在 Homestead 虚拟机中运行 sudo phpenmod xdebug 命令即可。接下来，按照 IDE 的引导来启用调试功能。最后，通过浏览器扩展或者 bookmarklet 配置浏览器触发 Xdebug（ https://www.jetbrains.com/phpstorm/marklets/ ）。

注意：Xdebug 会导致 PHP 运行变慢，要禁用 Xdebug，可以在虚拟机中运行 sudo phpdismod xdebug 并重启 PHP-FPM 服务。

(2)、调试 CLI 应用程序

要调试 PHP CLI 应用，可以在 Homestead 虚拟机中使用 xphp 命令：xphp path/to/script。

//

//// 自动启动 Xdebug

当发送请求到 Web 服务器调试功能测试时，自动启动调试比修改测试通过自定义请求头或者 Cookie 来触发调试要轻松许多。要设置 Xdebug 自动启动，可以在 Homestead 虚拟机中编辑 /etc/php/7.#/fpm/conf.d/20-xdebug.ini 添加如下配置来完成。

; If Homestead.yml contains a different subnet for the IP address, this address may be different...

xdebug.remote\_host = 192.168.10.1

xdebug.remote\_autostart = 1

(3)、使用 Blackfire 分析应用程序（ 使用 Blackfire 分析应用 ）

Blackfire（ https://blackfire.io/docs/introduction ） 是一个用于分析 Web 请求和 CLI 应用以及编写性能断言的 Saas 服务，它提供了一个交互式用户界面用来显示调用关系图和时间线中的分析数据，可以在开发、预发和生产环境中使用，对终端用户来说也没有额外开销，我们可以通过它实现对代码和 php.ini 配置的性能、质量和安全检查。

Blackfire Player（ https://blackfire.io/docs/player/index ） 是一个开源的功能强大的 Web 爬取、测试和数据提取应用，可以和 Blackfire 一起用于脚本分析场景。

要启用 Blackfire，需使用 Homestead 配置文件的 features 配置项来设置。

features:

- blackfire:

server\_id: "server\_id"

server\_token: "server\_value"

client\_id: "client\_id"

client\_token: "client\_value"

获取 Blackfire 服务器凭证和客户端凭证需要注册账号（ https://blackfire.io/signup ），Blackfire 提供了多个方案来分析应用，包括一个客户端工具和浏览器扩展，更多详细请参考 Blackfire 官方文档（ https://blackfire.io/docs/cookbooks/index ）。

//

//// 使用 XHGui 分析 PHP 性能

XHGui 是一个用于探索 PHP 应用性能的图形化界面，需要结合 XHProf 使用。要启用 XHGui，需在站点配置中新增 xhgui: 'true' 配置。

sites:

- map: your-site.test

to: /home/vagrant/your-site/public

type: "apache"

xhgui: 'true'

如果站点已经存在，更新配置之后需要运行 vagrant provision 命令使其生效。

要分析一个 Web 请求，需要添加 xhgui=on 到查询字符串作为请求参数，XHGui 会自动添加 Cookie 到响应，以便后续请求不再需要添加查询字符串。你可以访问 http://your-site.test/xhgui 来查看应用分析结果。

要使用 XHGui 分析一个 CLI 请求，需要在命令之前加上 XHGUI=on 前缀：XHGUI=on path/to/script。

CLI 分析结果可以通过和 Web 分析结果一样的方式查看。

注意以上分析行为会让脚本执行变慢，甚至会导致性能降低一半，因此，我们通常会按照百分比选取一小部分样本进行分析，并根据分析结果进行改进，此外，注意分析结果中的执行时间包含了调试器中花费的所有时间。

由于性能分析会耗费大量的磁盘空间，它们会在几天之后被自动删除。

--

5、网络接口

Homestead.yaml 中的 networks 属性为 Homestead 环境配置网络接口。你可以根据需要配置多个接口。

networks:

- type: "private\_network"

ip: "192.168.10.20"

要开启一个 桥接（ https://www.vagrantup.com/docs/networking/public\_network.html ） 接口，需配置 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（ https://www.vagrantup.com/docs/networking/public\_network.html ），只需要从配置中移除 ip 选项即可。

networks:

- type: "public\_network"

bridge: "en1: Wi-Fi (AirPort)"

--

6、扩充 Homestead（ 扩展 Homestead ）

你可以使用 Homestead 根目录下的 after.sh 脚本来扩展 Homestead 的功能，在这个文件中，你可以添加任意 shell 命令来配置和自定义你的虚拟机。

当自定义 Homestead 时，Ubuntu 可能会询问你是要保留程序包的原始配置还是用一个新的配置文件覆盖它。为了避免这种情况，你应该在安装软件包时使用以下的命令，以避免覆盖之前由 Homestead 编写的任何配置。

sudo apt-get -y \

-o Dpkg::Options::="--force-confdef" \

-o Dpkg::Options::="--force-confold" \

install your-package

//

//// 用户自定义

在团队环境中使用 Homestead 时，你可能想要调整 Homestead 以便更好地适应你自己喜好的个人开发风格，这可以通过在 Homestead 根目录下创建 user-customizations.sh 文件来完成，在这个文件中，你可以做一些自己喜欢的定制化工作；不过，注意不要把 user-customizations.sh 文件提交到代码仓库中。

--

7、更新 Homestead

更新 Homestead 之前确保你已经在 Homestead 目录下通过如下命令移除了当前的虚拟机：vagrant destroy。

接下来，需要更新 Homestead 源码，如果你已经克隆仓库到本地，可以在项目根目录下运行如下命令进行更新。

git fetch

git pull origin release

这些命令从 GitHub 仓库拉取最新的 Homestead 代码，获取最新的标记，然后检出最新的标记版本。你能在 GitHub 版本页面（ https://github.com/laravel/homestead/releases ） 找到最新稳定的版本。

如果你通过你的项目的 composer.json 安装 Homestead，你应当确保你的 composer.json 包含 "laravel/homestead": "^9" 并更新你的依赖项：composer update。

然后，需要通过 vagrant box update 命令更新 Vagrant 盒子：vagrant box update。

最后，你需要重新生成新的 Homestead 盒子来使用最新安装的 Vagrant：vagrant up。

--

8、提供器的特殊设置（ 提供特殊设置 ）

(1)、VirtualBox

//// natdnshostresolver

默认情况下，Homestead 将 natdnshostresolver 配置设置为 on。这允许 Homestead 去使用你的主机操作系统的 DNS 设置。如果你不想覆盖这个行为，添加如下的行到你的 Homestead.yaml 文件。

provider: virtualbox

natdnshostresolver: off

//

//// Windows 上的符号链接

如果符号链接在你的 Windows 机器上无法正常工作，你可能需要添加以下的块到 Vagrantfile 文件。

config.vm.provider "virtualbox" do |v|

v.customize ["setextradata", :id, "VBoxInternal2/SharedFoldersEnableSymlinksCreate/v-root", "1"]

end

--

--

五、Valet

https://learnku.com/docs/laravel/6.x/valet/5128

--

1、简介

Valet 是 Mac 极简主义者的 Laravel 开发环境。没有 Vagrant，不需要配置 /etc/hosts 文件。甚至可以使用本地隧道公开共享你的站点。是的，我们都喜欢 Valet。

Laravel Valet 为您的 Mac 设置了开机后始终在后台运行 Nginx 服务（ https://www.nginx.com/ ）。然后，Valet 使用 DnsMasq （ https://en.wikipedia.org/wiki/Dnsmasq ）将所有指向安装在本地的计算机站点请求代理到 \*.test 结尾的域名上。

换句话说，一个速度极快的 Laravel 开发环境只占用 7MB 内存。Valet 并不是想要完全替换 Vagrant 或 Homestead，只是提供另外一种使用起来更加灵活、方便、以及内存占用更小的选择。

//

//// Valet 支持但不局限于以下内容

// Laravel

https://laravel.com/

// Lumen

https://lumen.laravel.com/

// Bedrock

https://roots.io/bedrock/

// CakePHP 3

https://cakephp.org/

// Concrete5

https://www.concrete5.org/

// Contao

https://contao.org/en/

// Craft

https://craftcms.com/

// Drupal

https://www.drupal.org/

// Jigsaw

https://jigsaw.tighten.co/

// Joomla

https://www.joomla.org/

// Katana

https://github.com/themsaid/katana

// Kirby

https://getkirby.com/

// Magento

https://magento.com/

// OctoberCMS

https://octobercms.com/

// Sculpin

https://sculpin.io/

// Slim

https://www.slimframework.com/

// Statamic

https://statamic.com/

// Static HTML

// Symfony

https://symfony.com/

// WordPress

https://wordpress.org/

// Zend

https://framework.zend.com/

当然，您也可以使用 custom drivers( https://learnku.com/docs/laravel/6.x/valet/5128#custom-valet-drivers ) 来拓展您的 Valet。

(1)、Valet 还是 Homestead

您应该知道，Laravel 还提供了另外一种本地开发环境 Homestead( https://learnku.com/docs/laravel/6.0/homestead )。Homestead 和 Valet 的不同之处在于目标受众和本地开发的方式。Homestead 提供了一个完整的、具有自动化的 Nginx 配置的 Ubuntu 虚拟机。如果你想在 Windows／Linux 上实现完全虚拟化的 Linux 开发环境，Homestead 是一个不错的选择。

Valet 仅支持 Mac，并要求你将 PHP 和数据库服务器直接安装到本地机器上。这可以很容易地通过使用 Homebrew( https://brew.sh/ ) 命令来实现，比如 brew install php 和 brew install mysql。Valet 提供了一个极快的、资源消耗最少本地开发环境，非常适合只需要 PHP／MySQL 并且不需要虚拟开发环境的开发人员。

Valet 和 Homestead 都是配置 Laravel 开发环境的绝佳选择。选择哪一个仅仅取决于个人喜好和团队的需求。

--

2、安装

Valet 需要 macOS 系统和 Homebrew( https://brew.sh/ )。在安装之前，要确保没有其它程序（如 Apache 或 Nginx）占用了本地机器的 80 端口。

//a/ 使用 brew update 将 Homebrew( https://brew.sh/ ) 安装或更新到最新版本。

//b/ 使用 Homebrew 的 brew install php 命令安装 PHP 7.3。

//c/ 安装 Composer( https://getcomposer.org/ )。

//d/ 使用 Composer 的 composer global require laravel/valet 命令安装 Valet，并确保 ~/.composer/vendor/bin 目录在系统的 "PATH" 中。

//e/ 运行 valet install 命令来配置和安装 Valet 和 DnsMasq，并注册 Valet 后台服务，随系统运行自行启动。

安装完 Valet 后，可以尝试使用类似 ping foobar.test 的命令在终端上 ping 任何一个 \*.test 的域名。如果 Valet 安装正确，可以在终端上看到来自 127.0.0.1 的响应。

每次机器启动时，Valet 会自动启动其进程。所以只要完成了 Valet 的初始化，就无需再次运行 valet start 或 valet install。

//

///1/ 使用其它域名

默认情况下，Valet 使用 .test 顶级域名为你的项目提供服务。如果你想使用其它域名，可以使用 valet tld tld-name 命令。

例如，如果你想使用 .app 来代替 .test，运行 valet tld app，Valet 会自动将站点顶级域名改为 \*.app 。

//

///2/ 数据库

如果你要使用数据库，请在终端运行 brew install mysql@5.7 安装 MySQL。一旦安装完成，你可以使用 brew services start mysql@5.7 命令启动 MySQL。然后，你可以使用 root 用户名和空字符串密码连接到 127.0.0.1 的数据库。

//

///3/ Valet 允许你使用 valet use php@version 命令来切换 PHP 版本。如果指定版本尚未安装，Valet 将通过 Brew 来安装指定的 PHP 版本：

valet use php@7.2

valet use php

(1)、升级

你可以在终端使用 composer global update 命令来更新 Valet。升级后，如有需要，最好再次运行 valet install ，以便 Valet 对配置文件进行升级。

//

///1/ 升级到 Valet 2.0

Valet 2.0 将 Valet 底层的 Web 服务器从 Caddy 转移到 Nginx。升级到此版本之前，你应该运行以下命令停止并卸载现有的 Caddy 进程。

valet stop

valet uninstall

接下来，就根据你采用的安装方式来升级 Valet （通常是通过 Git 或 Composer ）。如果是通过 Composer 安装了 Valet ，则应使用以下命令更新到最新的主要版本。

composer global require laravel/valet

如果更新了 Valet 的源码，你应该执行 install 命令。

valet install

valet restart

升级过后，可能需要重新设置或重新链接你的站点。

--

3、服务站点

安装了 Valet 之后，你就可以开始设置站点。Valet 提供两个命令来为 Laravel 的站点提供服务：park 和 link。

(1)、Park 命令

运行 mkdir ~/Sites 命令在 Mac 上创建一个新的目录。接下来，运行 cd ~/Sites 和 valet park 将当前的工作目录作为 Valet 搜索站点的路径。

接下来，在这个目录中创建一个新的 Laravel 站点：laravel new blog。

在浏览器中打开 http://blog.test。

就这么多。 现在，你在 『parked』的目录中创建的任何 Laravel 项目都将自动使用 http://folder-name.test 这种方式访问。

(2)、Link 命令

如果要在目录中提供单个站点而不是整个目录，就使用 link 命令。

|1> 要使用该命令，先在终端里切换到你的某个项目并运行 valet link app-name。Valet 会在 ~/.config/valet/Sites 中创建一个符号链接指向当前的目录。

|2> 运行 link 命令后，你可以在浏览器通过 http://app-name.test 访问站点。

运行 valet links 命令可以查看所有目录链接的列表。你还可以使用 valet unlink app-name 来删除符号链接。

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

(3)、使用 TLS 构建安全站点

默认情况下，Valet 服务站点通过纯 HTTP 的方式。但是，如果你想一个站点使用 HTTP/2 通过加密 TLS 提供服务，使用 secure 命令。例如，如果你的站点通过 Valet 在 laravel.test 域上提供服务，你应当运行如下的命令去保护它：valet secure laravel。

要一个站点『解除保护』并恢复为通过纯 HTTP 提供服务它的流量，使用 unsecure 命令。与 secure 命令一样，该命令接受你希望去解除保护的主机名称：valet unsecure laravel。

--

4、共享站点

Valet 甚至包括与世界共享你的本地站点的命令。一旦安装 Valet 后无需安装其它软件。

要共享站点，在终端中导航你的站点目录并运行 valet share 命令。一个公开可访问的 URL 将被插入到剪贴板中，并可以直接粘贴到你的浏览器中。仅此而已。

要停止共享你的站点，请按 Control + C 去取消该过程。

--

5、网站特定环境变量

有些使用其他框架的应用程序可能需要依赖一些服务器环境变量但是没有在你的项目中提供配置。 Valet 可以通过添加.valet-env.php 文件来允许你针对单独的网站进行设置环境变量。这些环境变量将会添加到 $\_SERVER 的全局数组中。

<?php

return [

'WEBSITE\_NAME' => 'My Blog',

];

?>

--

6、自定义 Valet 驱动

你可以编写你自己的 Valet『驱动』来为 Valet 原本不支持的其它框架或 CMS 上运行的 PHP 应用程序提供服务。当你安装 Valet 时，会创建一个包含 SampleValetDriver.php 文件的 ~/.config/valet/Drivers 目录。该文件包含一个示例驱动程序实现，演示了如何编写一个自定义的驱动程序。编写驱动程序仅需要你去实现三个方法：serves，isStaticFile 和 frontControllerPath。

这三个方法都接受 $sitePath，$siteName 和 $uri 值作为参数。$sitePath 是你的机器上提供站点的完全限定路径。比如 /Users/Lisa/Sites/my-project。$siteName 是域（my-project）的『主机』/『站点名称』部分。$uri 是即将到来的请求 URL（/foo/bar）。

一旦你完成你的自定义 Valet 驱动，使用 FrameworkValetDriver.php 命名约定将它放置在 ~/.config/valet/Drivers 目录中。例如，如果你为 WordPress 编写了一个自定义的 valet 驱动，你的文件名称应该是 WordPressValetDriver.php。

我们来看看自定义的 Valet 驱动程序应该实现的每种方法的示例实现。

///1/ serves 方法

如果你的驱动程序应当处理即将到来的请求时， serves 方法应该返回 true。否则，此方法应当返回 false。因此，在此方法中，你应该你企图确定给定的 $sitePath 是否包含你尝试提供的类型的项目。

例如，让我们假装正在编写一个 WordPressValetDriver。我们的 serves 方法可能看起来如下所示。

/\*\*

\* 确定驱动程序是否满足请求。

\*

\* @param string $sitePath

\* @param string $siteName

\* @param string $uri

\* @return bool

\*/

public function serves($sitePath, $siteName, $uri)

{

return is\_dir($sitePath.'/wp-admin');

}

//

///2/ isStaticFile 方法

isStaticFile 应当确定即将到来的请求是否针对一个『静态』文件，比如：图片和样式表。如果文件是静态的，此方法应当返回静态文件在磁盘上的完全限定路径。如果即将到来的请求不是针对一个静态文件，这个方法应当返回 false。

/\*\*

\* 确定即将到来的请求是否针对静态文件。

\*

\* @param string $sitePath

\* @param string $siteName

\* @param string $uri

\* @return string|false

\*/

public function isStaticFile($sitePath, $siteName, $uri)

{

if (file\_exists($staticFilePath = $sitePath.'/public/'.$uri)) {

return $staticFilePath;

}

return false;

}

注意：如果 serves 方法对即将到来的请求且请求 URI 不是 / 返回 true 时，才会调用 isStaticFile 方法。

//

///3/ frontControllerPath 方法

frontControllerPath 方法应当返回你的应用程序的『前端控制器』完全限定的路径，它通常是你的 『index.php』文件或者等效的文件。

/\*\*

\* 获取对应用程序的前端控制器的完全解析路径。

\*

\* @param string $sitePath

\* @param string $siteName

\* @param string $uri

\* @return string

\*/

public function frontControllerPath($sitePath, $siteName, $uri)

{

return $sitePath.'/public/index.php';

}

(1)、本地驱动

如果你想为单个应用程序自定义 Valet 驱动程序，在应用程序的根目录下创建一个 LocalValetDriver.php 文件。你的自定义驱动程序可以扩展 ValetDriver 基类或者继承一个现有的应用程序的特定驱动程序，比如：LaravelValetDriver。

class LocalValetDriver extends LaravelValetDriver

{

/\*\*

\* 确定驱动程序是否满足请求。

\*

\* @param string $sitePath

\* @param string $siteName

\* @param string $uri

\* @return bool

\*/

public function serves($sitePath, $siteName, $uri)

{

return true;

}

/\*\*

\* 获取对应用程序的前端控制器的完全解析路径。

\*

\* @param string $sitePath

\* @param string $siteName

\* @param string $uri

\* @return string

\*/

public function frontControllerPath($sitePath, $siteName, $uri)

{

return $sitePath.'/public\_html/index.php';

}

}

--

7、其他 Valet 命令

//// 命令/描述

valet forget 从一个『驻留』目录运行此命令，从驻留目录列表将其它移除；

valet log 从 Valet 的服务中查看日志；

valet paths 查看所有『驻留』路径；

valet restart 重启 Valet 守护进程；

valet start 开启 Valet 守护进程；

valet stop 停止 Valet 守护进程；

valet trust 为 Brew 和 Valet 添加文件修改权限使 Valet 输入命令的时候不需要输入密码；

valet uninstall 完成卸载 Valet 守护进程；

--

--

六、部署

https://learnku.com/docs/laravel/6.x/deployment/5129

--

1、简介

当你准备部署你的 Laravel 应用到生产环境时，请确保几个重要的注意点以保证你的应用能尽可能高效的运行。本文中我们将会覆盖几个重点来确保你的 Laravel 应用部署得当。

--

2、服务器配置

(1)、Nginx

如果你想要部署你的应用到 Nginx 服务器上，你可能会用到下面这个配置文件作为一个开始来配置你的 Web 服务器。很有可能，这个文件需要根据你的服务器配置来做一些自定义的修改。如果你需要协助来管理你的服务器，可以考虑使用 Laravel Forge( https://forge.laravel.com/ )。

server {

listen 80;

server\_name example.com;

root /example.com/public;

add\_header X-Frame-Options "SAMEORIGIN";

add\_header X-XSS-Protection "1; mode=block";

add\_header X-Content-Type-Options "nosniff";

index index.html index.htm index.php;

charset utf-8;

location / {

try\_files $uri $uri/ /index.php?$query\_string;

}

location = /favicon.ico { access\_log off; log\_not\_found off; }

location = /robots.txt { access\_log off; log\_not\_found off; }

error\_page 404 /index.php;

location ~ \.php$ {

fastcgi\_pass unix:/var/run/php/php7.2-fpm.sock;

fastcgi\_index index.php;

fastcgi\_param SCRIPT\_FILENAME $realpath\_root$fastcgi\_script\_name;

include fastcgi\_params;

}

location ~ /\.(?!well-known).\* {

deny all;

}

}

--

3、改进

(1)、改进自动加载器 ( 自动加载器改进 )

当你准备往生产环境部署应用时，确保你优化了你的 Composer 类的自动加载映射，这样可以使 Composer 可以很快的找到正确的加载文件去加载给定的类。

composer install --optimize-autoloader --no-dev

小提示：除了优化自动加载器，你还应该确保在你的项目代码仓库中包含了 composer.lock 这个文件。当你的项目代码中有 composer.lock 这个文件时，便可以更快的安装项目中需要的依赖项。

(2)、改进配置加载 ( 优化配置加载 )

当你将应用程序部署到生产环境时，你应当确保在你部署过程中运行 config:cache Artisan 命令。

php artisan config:cache

此命令将所有 Laravel 的配置文件合并到一个缓存文件，这将极大地减少框架在加载配置值时必须对文件系统进行访问的次数。

注意：如果在你部署过程中执行 config:cache 命令，你应当确保你仅从你的配置文件中调用 env 函数。一旦配置被缓存，.env 文件将不被加载并且对 env 函数的所有调用将返回 null。

(3)、改进路由加载 ( 优化路由加载 )

如果你想构建具有许多路由的大型应用程序，你应当确保在你部署的过程中运行 route:cache Artisan 命令：php artisan route:cache。

此命令将为所有路由注册缩减到一个缓存文件中的单个方法调用，从而在注册数百个路由时提高了路由注册的性能。

注意：由于此功能使用 PHP 序列化，你仅能缓存专门使用基于控制器路由的应用程序路由。PHP 不能序列化闭包路由。

--

4、使用 Forge 部署

如果你还没有准备好管理自己的服务器配置，或者不熟悉配置对运行强大的 Laravel 应用程序所需的各种服务，Laravel Forge（ https://forge.laravel.com/ ） 是一个不错的选择。

Laravel Forge 能在各种基础设施提供商（如：DigitalOcean，Linode，AWS 等等）上创建服务器。另外，Forge 安装和管理构建强大 Laravel 应用程序所需的所有工具，比如：Nginx，MySQL，Redis，Memcached，Beanstalk 等等。

--

--

第三卷 核心架构

--

--

一、请求周期 ( 请求生命周期 )

https://learnku.com/docs/laravel/6.x/lifecycle/5130

--

1、介绍

在「日常生活」中 使用任何工具时，如果理解了该工具的工作原理。那么用起来就会更加得心应手。应用开发也是如此，当你能真正懂得一个功能背后实现原理时，用起来会更加顺手，方便。

文档存在目的是为了让你更加清晰地了解 Laravel 框架是如何工作。通过框架进行全面了解，让一切都不再感觉很「神奇」。相信我，这有助于你更加清楚自己在做什么，对自己想做的事情更加胸有成竹。就算你不明白所有的术语，也不用因此失去信心！只要多一点尝试、学着如何运用，随着你浏览文档的其他部分，你的知识一定会因此增长。

--

2、生命周期概述

(1)、首先

Laravel 应用的所有请求入口都是 public/index.php 文件。而所有的请求都是经由你的 Web 服务器（Apache/Nginx）通过配置引导到这个文件。 index.php 文件代码并不多，但是，这里是加载框架其它部分的起点。

index.php 文件加载 Composer 生成的自动加载设置，然后从 bootstrap/app.php 脚本中检索 Laravel 应用程序的实例。 Laravel 本身采取的第一个动作是创建一个应用程序 / 服务容器( https://learnku.com/docs/laravel/6.x/container )。

(2)、HTTP / Console 内核

接下来， 根据进入应用程序的请求类型来将传入的请求发送到 HTTP 内核或控制台内核。而这两个内核是用来作为所有请求都要通过的中心位置。 现在，我们先看看位于 app/Http/Kernel.php 中的 HTTP 内核。

HTTP 内核继承了 Illuminate\Foundation\Http\Kernel 类，该类定义了一个 bootstrappers 数组。这个数组中的类在请求被执行前运行，这些 bootstrappers 配置了错误处理，日志，检测应用环境( https://learnku.com/docs/laravel/6.x/configuration#environment-configuration )，以及其它在请求被处理前需要执行的任务。

HTTP 内核还定义了所有请求被应用程序处理之前必须经过的 HTTP 中间件( https://learnku.com/docs/laravel/6.x/middleware )，这些中间件处理 HTTP 会话 读写、判断应用是否处于维护模式、 验证 CSRF 令牌( https://learnku.com/docs/laravel/6.x/csrf ) 等等。

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

(3)、服务提供者

内核启动操作中最重要的便是你应用的 服务提供者( https://learnku.com/docs/laravel/6.x/providers ) 了。所有应用下的服务提供者均配置到了 config/app.php 配置文件中的 providers 数组中。 第一步，所有服务提供者的 register 方法会被调用，然后一旦所有服务提供者均注册后， boot 方法才被调用。

服务提供者给予框架开启多种多样的组件，像数据库，队列，验证器，以及路由组件。只要被启动服务提供者就可支配框架的所有功能，所以服务提供者也是 Laravel 整个引导周期最重要组成部分。

(4)、请求调度

一旦启动且所有服务提供者被注册，Request 会被递送给路由。路由将会调度请求，交给绑定的路由或控制器，也当然包括路由绑定的中间件。

--

3、聚焦服务提供者

服务提供者是 Laravel 真正意义的生命周期中的关键。应用实例一旦创建，服务提供者就被注册，然后请求被启动的应用接管。简单吧！

牢牢掌握服务提供者的构建和其对 Laravel 应用处理机制的原理是非常有价值的。当然，你的应用默认的服务提供会存放在 app/Providers 下面。

默认的， AppServiceProvider 是空白的。这个提供者是一个不错的位置，用于你添加应用自身的引导处理和服务容器绑定。当然，大型项目中，你可能希望创建数个粒度更精细的服务提供者。

--

--

二、服务容器

https://learnku.com/docs/laravel/6.x/container/5131

--

1、简介

Laravel 服务容器是一个用于管理类的依赖和执行依赖注入的强大工具。依赖注入这个花哨名词实质上是指：类的依赖通过构造函数，或者某些情况下通过 「setter」 方法 「注入」到类中。

//

//// 来看一个简单的例子

<?php

namespace App\Http\Controllers;

use App\User;

use App\Repositories\UserRepository;

use App\Http\Controllers\Controller;

class UserController extends Controller

{

/\*\*

\* 用户存储库的实现。

\*

\* @var UserRepository

\*/

protected $users;

/\*\*

\* 创建新的控制器实例。

\*

\* @param UserRepository $users

\* @return void

\*/

public function \_\_construct(UserRepository $users)

{

$this->users = $users;

}

/\*\*

\*显示指定用户的 profile。

\*

\* @param int $id

\* @return Response

\*/

public function show($id)

{

$user = $this->users->find($id);

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

}

}

?>

在这个例子中，控制器 UserController 需要从数据源获取 users。因此，我们要 注入 一个能够获取 users 的服务。在当前上下文中，我们的 UserRepository 很可能是使用 Eloquent 从数据库中获取 user 信息。 然而，由于 repository 是被注入的，所以我们可以轻易地将其切换为另一个的实现。这种注入方式的便利之处还体现在当我们为应用编写测试时，我们还可以轻松地 “模拟” 或创建 UserRepository 的虚拟实现。

想要构建强大的大型应用，至关重要的一件事是：要深刻地理解 Laravel 服务容器。当然，为 Laravel 的核心代码做出贡献也一样。

--

2、绑定 (服务绑定)

(1)、基本绑定 (基础绑定)

几乎所有的服务容器绑定都会在 服务提供者( https://learnku.com/docs/%7B%7Bversion%7D%7D/providers ) 中注册，下面示例中的大多数将演示如何在该上下文（服务提供者）中使用容器。

Tip：如果某个容器不依赖于任何接口就没必要去绑定类在这个容器里。容器不需要指定如何构建这些对象，因为它可以使用反射来自动解析这些对象。

//

///1/ 简单绑定

在服务提供器中，你总是可以通过 $this->app 属性访问容器。我们可以通过容器的 bind 方法注册绑定，bind 方法的第一个参数为要绑定的类 / 接口名，第二个参数是一个返回类实例的 Closure。

$this->app->bind('HelpSpot\API', function ($app) {

return new HelpSpot\API($app->make('HttpClient'));

});

注意，我们接受容器本身作为解析器的参数。然后，我们可以使用容器来解析正在构建的对象的子依赖。

//

///2/ 绑定一个单例

singleton 方法将类或接口绑定到只解析一次的容器中。一旦单例绑定被解析，相同的对象实例会在随后的调用中返回到容器中。

$this->app->singleton('HelpSpot\API', function ($app) {

return new HelpSpot\API($app->make('HttpClient'));

});

//

///3/ 绑定实例

你也可以使用 instance 方法将现有对象实例绑定到容器中。给定的实例会始终在随后的调用中返回到容器中。

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

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

//

///4/ 绑定基本值

当你有一个类不仅需要接受一个注入类，还需要注入一个基本值（比如整数）。你可以使用上下文绑定来轻松注入你的类需要的任何值。

$this->app->when('App\Http\Controllers\UserController')

->needs('$variableName')

->give($value);

(2)、绑定接口到实现

服务容器有一个很强大的功能，就是支持绑定接口到给定的实现。例如，如果我们有个 EventPusher 接口 和一个 RedisEventPusher 实现。一旦我们写完了 EventPusher 接口的 RedisEventPusher 实现，我们就可以在服务容器中注册它，像这样下面。

$this->app->bind(

'App\Contracts\EventPusher',

'App\Services\RedisEventPusher'

);

这么做相当于告诉容器：当一个类需要实现 EventPusher 时，应该注入 RedisEventPusher。现在我们就可以在构造函数或者任何其他通过服务容器注入依赖项的地方使用类型提示注入 EventPusher 接口。

use App\Contracts\EventPusher;

/\*\*

\* Create a new class instance.

\*

\* @param EventPusher $pusher

\* @return void

\*/

public function \_\_construct(EventPusher $pusher)

{

$this->pusher = $pusher;

}

(3)、上下文绑定

有时你可能有两个类使用了相同的接口，但你希望各自注入不同的实现。例如， 有两个控制器可能依赖了 Illuminate\Contracts\Filesystem\Filesystem 契约. Laravel 提供了一个简单的，优雅的接口来定义这个行为。

～～～

use Illuminate\Support\Facades\Storage;

use App\Http\Controllers\PhotoController;

use App\Http\Controllers\VideoController;

use Illuminate\Contracts\Filesystem\Filesystem;

$this->app->when(PhotoController::class)

->needs(Filesystem::class)

->give(function () {

return Storage::disk('local');

});

$this->app->when([VideoController::class, UploadController::class])

->needs(Filesystem::class)

->give(function () {

return Storage::disk('s3');

});

～～～

(4)、标记

有时候，你可能需要解析某个「分类」下的所有绑定。 比如， 你可能正在构建一个报表的聚合器，它接收一个包含不同 Report 接口实现的数组。注册 Report 实现之后，你可以使用 tag 方法给他们分配一个标签。

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

//

});

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

//

});

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

一旦服务被标记，你就可以通过 tagged 方法轻松地解析它们。

$this->app->bind('ReportAggregator', function ($app) {

return new ReportAggregator($app->tagged('reports'));

});

(5)、扩展绑定

extend 方法可以修改已解析的服务。比如，当一个服务被解析后，你可以添加额外的代码来修饰或者配置它。 extend 方法接受一个闭包，该闭包唯一的参数就是这个服务， 并返回修改过的服务。

$this->app->extend(Service::class, function ($service) {

return new DecoratedService($service);

});

--

3、解析 ( 解析实例 )

(1)、Make 方法

你可以使用 make 方法从容器中解析出类实例。 make 方法接收你想要解析的类或接口的名字。

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

//

如果你的代码处于无法访问 $app 变量的位置，则可用全局辅助函数 resolve 来解析。

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

//

如果类依赖不能通过容器解析，你可以通过将它们作为关联数组作为 makeWith 方法的参数注入。

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

(2)、自动注入

另外，并且更重要的是，你可以简单地使用「类型提示」 的方式在类的构造函数中注入那些需要容器解析的依赖项，包括 控制器( https://learnku.com/docs/%7B%7Bversion%7D%7D/controllers )，事件监听器( https://learnku.com/docs/%7B%7Bversion%7D%7D/events )， 队列任务( https://learnku.com/docs/%7B%7Bversion%7D%7D/queues )，中间件( https://learnku.com/docs/%7B%7Bversion%7D%7D/middleware )等 。实际上，这才是大多数对象应该被容器解析的方式。

例如，你可以在控制器的构造函数中添加一个 repository 的类型提示，然后这个 repository 将会被自动解析并注入类中。

<?php

namespace App\Http\Controllers;

use App\Users\Repository as UserRepository;

class UserController extends Controller

{

/\*\*

\* The user repository instance.

\*/

protected $users;

/\*\*

\* Create a new controller instance.

\*

\* @param UserRepository $users

\* @return void

\*/

public function \_\_construct(UserRepository $users)

{

$this->users = $users;

}

/\*\*

\* Show the user with the given ID.

\*

\* @param int $id

\* @return Response

\*/

public function show($id)

{

//

}

}

?>

--

4、容器事件

服务容器每次解析对象会触发一个事件，你可以使用 resolving 方法监听这个事件。

$this->app->resolving(function ($object, $app) {

// Called when container resolves object of any type...

});

$this->app->resolving(HelpSpot\API::class, function ($api, $app) {

// Called when container resolves objects of type "HelpSpot\API"...

});

正如你所看到的，被解析的对象将会被传入回调函数，这使得你能够在对象被传给调用者之前给它设置额外的属性。

--

5、PSR-11

Laravel 的服务容器实现了 PSR-11 接口( https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-11-container.md )。因此，你可以使用 PSR-11 容器 『接口类型提示』 来获取 Laravel 容器的实例。

use Psr\Container\ContainerInterface;

Route::get('/', function (ContainerInterface $container) {

$service = $container->get('Service');

//

});

如果无法解析给定的标识符，则将会引发异常。未绑定标识符时，会抛出 Psr\Container\NotFoundExceptionInterface 异常。如果标识符已绑定但无法解析，会抛出 Psr\Container\ContainerExceptionInterface 异常。

--

--

三、服务提供者

https://learnku.com/docs/laravel/6.x/providers/5132

--

1、简介

服务提供者是所有 Laravel 应用程序的引导中心。你的应用程序，以及 通过服务器引导的 Laravel 核心服务都是通过服务提供器引导。

但是，「引导」是什么意思呢？ 通常，我们可以理解为注册，比如注册服务容器绑定，事件监听器，中间件，甚至是路由。服务提供者是配置应用程序的中心。

当你打开 Laravel 的 config/app.php 文件时，你会看到 providers 数组。数组中的内容是应用程序要加载的所有服务提供者的类。当然，其中有很多 「延迟」 提供者，他们并不会在每次请求的时候都加载，只有他们的服务实际被需要时才会加载。

本篇你将会学到如何编写自己的服务提供者，并将其注册到你的 Laravel 应用程序中。

--

2、编写服务提供者

所有的服务提供者都会继承 Illuminate\Support\ServiceProvider 类。 大多服务提供者都包含一个 register 和一个 boot 方法。在 register 方法中， 你只需要将服务绑定到服务容器( https://learnku.com/docs/%7B%7Bversion%7D%7D/container )。而不要尝试在 register 方法中注册任何监听器，路由，或者其他任何功能。

//

使用 Artisan 命令行工具，通过 make:provider 命令可以生成一个新的提供者。

php artisan make:provider RiakServiceProvider

(1)、注册方法

如上所述，在 register 方法中，你只需要将服务绑定到服务容器( https://learnku.com/docs/%7B%7Bversion%7D%7D/container )中。而不要尝试在 register 方法中注册任何监听器，路由，或者其他任何功能。否则，你可能会意外地使用到尚未加载的服务提供者提供的服务。

//

//// 让我们来看一个基础的服务提供者。在任何服务提供者方法中，你总是通过 $app 属性来访问服务容器。

<?php

namespace App\Providers;

use Riak\Connection;

use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider

{

/\*\*

\* 在服务容器里注册

\*

\* @return void

\*/

public function register()

{

// 服务容器\_绑定服务

$this->app->singleton(Connection::class, function ($app) {

return new Connection(config('riak'));

});

}

}

?>

这个服务提供者只是定义了一个 register 方法，并且使用这个方法在服务容器中定义了一个 Riak\Connection 接口。如果你不理解服务容器的工作原理，请查看其文档( https://learnku.com/docs/%7B%7Bversion%7D%7D/container )。

//

//// bindings 和 singletons 的特性

如果你的服务提供器注册了许多简单的绑定，你可能想用 bindings 和 singletons 属性替代手动注册每个容器绑定。当服务提供器被框架加载时，将自动检查这些属性并注册相应的绑定。

<?php

namespace App\Providers;

use App\Contracts\ServerProvider;

use App\Contracts\DowntimeNotifier;

use Illuminate\Support\ServiceProvider;

use App\Services\PingdomDowntimeNotifier;

use App\Services\DigitalOceanServerProvider;

class AppServiceProvider extends ServiceProvider

{

/\*\*

\* 设定所有的容器绑定的对应关系

\*

\* @var array

\*/

public $bindings = [

ServerProvider::class => DigitalOceanServerProvider::class,

];

/\*\*

\* 设定所有的单例模式容器绑定的对应关系

\*

\* @var array

\*/

public $singletons = [

DowntimeNotifier::class => PingdomDowntimeNotifier::class,

ServerToolsProvider::class => ServerToolsProvider::class,

];

}

?>

(2)、引导方法

如果我们要在服务提供者中注册一个 视图合成器( https://learnku.com/docs/%7B%7Bversion%7D%7D/views#view-composers ) 该怎么做？ 这就需要用到 boot 方法了。 该方法在所有服务提供者被注册以后才会被调用， 这就是说我们可以在其中访问框架已注册的所有其它服务。

<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class ComposerServiceProvider extends ServiceProvider

{

/\*\*

\* 启动所有的应用服务。

\*

\* @return void

\*/

public function boot()

{

view()->composer('view', function () {

//

});

}

}

?>

//

//// 启动方法的依赖注入

你可以为服务提供者的 boot 方法设置类型提示。 服务容器( https://learnku.com/docs/%7B%7Bversion%7D%7D/container ) 会自动注入你所需要的依赖。

use Illuminate\Contracts\Routing\ResponseFactory;

public function boot(ResponseFactory $response)

{

$response->macro('caps', function ($value) {

//

});

}

--

3、注册提供者

所有服务提供者都是通过配置文件 config/app.php 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。

//

//// 要注册提供器，只需要将其添加到数组。

'providers' => [

// 其他服务提供者

// ...

App\Providers\ComposerServiceProvider::class,

],

--

4、延迟提供者

如果你的服务提供者 只 在 服务容器( https://learnku.com/docs/%7B%7Bversion%7D%7D/container ) 中注册，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 编译并保存延迟服务提供者提供的所有服务的列表，以及其服务提供者类的名称。因此，只有当你在尝试解析其中一项服务时，Laravel 才会加载服务提供者。

要延迟加载提供者，需要实现 \Illuminate\Contracts\Support\DeferrableProvider 接口并置一个 provides 方法。这个 provides 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;

use Illuminate\Contracts\Support\DeferrableProvider;

class RiakServiceProvider extends ServiceProvider implements DeferrableProvider

{

/\*\*

\* 注册服务提供者。

\*

\* @return void

\*/

public function register()

{

$this->app->singleton(Connection::class, function ($app) {

return new Connection($app['config']['riak']);

});

}

/\*\*

\* 获取由提供者提供的服务。

\*

\* @return array

\*/

public function provides()

{

return [Connection::class];

}

}

?>

--

--

四、Facades

https://learnku.com/docs/laravel/6.x/facades/5133

--

1、简介

Facades 为应用的 服务容器( https://learnku.com/docs/laravel/6.x/container ) 提供了一个「静态」 接口。Laravel 自带了很多 Facades，可以访问绝大部分功能。Laravel Facades 实际是服务容器中底层类的 「静态代理」，相对于传统静态方法，在使用时能够提供更加灵活、更加易于测试、更加优雅的语法。

所有的 Laravel Facades 都定义在 Illuminate\Support\Facades 命名空间下。所以，我们可以轻松的使用 Facade。

use Illuminate\Support\Facades\Cache;

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

return Cache::get('key');

});

在 Laravel 文档中，有很多示例代码都会使用 Facades 来演示框架的各种功能。

--

2、何时使用 Facades

Facades 有很多优点，它提供了简单，易记的语法，从而无需手动注入或配置长长的类名。此外，由于他们对 PHP 静态方法的独特调用，使得测试起来非常容易。

然而，在使用 Facades 时，有些地方需要特别注意。使用 Facades 时最主要的危险就是会引起类作用范围的膨胀。由于 Facades 使用起来非常简单并且不需要注入，就会使得我们不经意间在单个类中使用许多 Facades ，从而导致类变得越来越大。然而使用依赖注入的时候，使用的类越多，构造方法就会越长，在视觉上注意到这个类有些庞大了。因此在使用 Facades 的时候，要特别注意控制类的大小，让类的作用范围保持短小。

Tip：在开发与 Laravel 进行交互的第三方扩展包时，最好选择注入 Laravel 契约( https://learnku.com/docs/laravel/6.x/contracts ) 而不使用 Facades 。因为扩展包是在 Laravel 之外构建，你无法使用 Laravel Facades 测试辅助函数。

(1)、Facades 相较于依赖注入

依赖注入的主要好处之一是能交换注入类的实现。在测试的时候非常有用，因为你可以注入一个 mock 或者 stub，并断言 stub 上的各种方法。

通常，真正的静态方法是不可能 mock 或 stub 的。但是 Facades 使用动态方法对服务容器中解析出来的对象方法的调用进行了代理，我们也可以像测试注入类实例一样测试 Facades。比如，像下面的路由。

use Illuminate\Support\Facades\Cache;

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

return Cache::get('key');

});

我们可以带上我们期望的参数编写下面的测试代码来验证 Cache::get 方法。

use Illuminate\Support\Facades\Cache;

/\*\*

\* 一个基础功能的测试用例。

\*

\* @return void

\*/

public function testBasicExample()

{

Cache::shouldReceive('get')

->with('key')

->andReturn('value');

$this->visit('/cache')

->see('value');

}

(2)、Facades 相较于辅助函数

除了 Facades，Laravel 还包含各种 『辅助函数』 来实现这些常用功能，比如生成视图、触发事件、任务调度或者发送 HTTP 响应。许多辅助函数都有与之对应的 Facades 。例如，下面这个 Facades 和辅助函数的作用是一样的。

return View::make('profile');

return view('profile');

//

Facade 和辅助函数之间没有实际的区别。当你使用辅助函数时，你可以像测试相应的 Facade 那样进行测试。例如，下面的路由。

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

return cache('key');

});

在底层实现，辅助函数 cache 实际是调用 Cache 这个 Facade 的 get 方法。因此，尽管我们使用的是辅助函数，我们依然可以带上我们期望的参数编写下面的测试代码来验证该方法。

use Illuminate\Support\Facades\Cache;

/\*\*

\* 一个基础功能的测试用例。

\*

\* @return void

\*/

public function testBasicExample()

{

Cache::shouldReceive('get')

->with('key')

->andReturn('value');

$this->visit('/cache')

->see('value');

}

--

3、Facades 工作原理

在 Laravel 应用中，Facade 就是一个可以从容器访问对象的类。其中核心的部件就是 Facade 类。不管是 Laravel 自带的 Facades，还是自定义的 Facades，都继承自 Illuminate\Support\Facades\Facade 类。

Facade 基类使用了\_\_callStatic() 魔术方法，直到对象从容器中被解析出来后，才会进行调用。在下面的例子中，调用了 Laravel 的缓存系统。通过浏览这段代码，可以假定在 Cache 类中调用了静态方法 get。

<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

use Illuminate\Support\Facades\Cache;

class UserController extends Controller

{

/\*\*

\* 显示给定用户的信息。

\*

\* @param int $id

\* @return Response

\*/

public function showProfile($id)

{

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

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

}

}

?>

注意在上面这段代码中，我们『导入』了 Cache Facade。这个 Facade 作为访问 Illuminate\Contracts\Cache\Factory 接口底层实现的代理。我们使用 Facade 进行的任何调用都将传递给 Laravel 缓存服务的底层实例。

//

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

class Cache extends Facade

{

/\*\*

\* 获取组件的注册名称。

\*

\* @return string

\*/

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

}

Cache Facade 继承了 Facade 类，并且定义了 getFacadeAccessor() 方法。这个方法的作用是返回服务容器绑定的名称。当用户调用 Cache Facade 中的任何静态方法时，Laravel 会从 服务容器( https://learnku.com/docs/laravel/6.x/container ) 中解析 cache 绑定以及该对象运行所请求的方法（在这个例子中就是 get 方法）。

--

4、实时 Facades

使用实时 Facades，你可以将应用程序中的任何类视为 Facade。为了说明这是如何使用的，我们来看看另一种方法。例如，假设我们的 Podcast 模型有一个 publish 方法。然而，为了发布 Podcast，我们需要注入一个 Publisher 实例。

<?php

namespace App;

use App\Contracts\Publisher;

use Illuminate\Database\Eloquent\Model;

class Podcast extends Model

{

/\*\*

\* 发布 Podcast。

\*

\* @param Publisher $publisher

\* @return void

\*/

public function publish(Publisher $publisher)

{

$this->update(['publishing' => now()]);

$publisher->publish($this);

}

}

?>

将发布者的实现注入到该方法中，我们可以轻松地测试这种方法，因为我们可以模拟注入的发布者。但是，它要求我们每次调用 publish 方法时都要传递一个发布者实例。使用实时的 Facades，我们可以保持同样的可测试性，而不需要显式地通过 Publisher 实例。要生成实时 Facade，请在导入类的名称空间中加上 Facades。

<?php

namespace App;

use Facades\App\Contracts\Publisher;

use Illuminate\Database\Eloquent\Model;

class Podcast extends Model

{

/\*\*

\* 发布 Podcast。

\*

\* @return void

\*/

public function publish()

{

$this->update(['publishing' => now()]);

Publisher::publish($this);

}

}

?>

当使用实时 Facade 时，发布者实现将通过使用 Facades 前缀后出现的接口或类名的部分来解决服务容器的问题。在测试时，我们可以使用 Laravel 的内置 facade 测试辅助函数来模拟这种方法调用。

<?php

namespace Tests\Feature;

use App\Podcast;

use Tests\TestCase;

use Facades\App\Contracts\Publisher;

use Illuminate\Foundation\Testing\RefreshDatabase;

class PodcastTest extends TestCase

{

use RefreshDatabase;

/\*\*

\* 一个测试演示。

\*

\* @return void

\*/

public function test\_podcast\_can\_be\_published()

{

$podcast = factory(Podcast::class)->create();

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

$podcast->publish();

}

}

?>

--

5、Facade 参考类

在下面你可以找到每个 Facade 类及其对应的底层类。这是一个查找给定 Facade 类 API 文档的工具。服务容器绑定( https://learnku.com/docs/laravel/6.x/container ) 的关键信息也包含在内。

Facade/Class/服务容器绑定

App Illuminate\Foundation\Application ( https://laravel.com/api/6.x/Illuminate/Foundation/Application.html ) app

Artisan Illuminate\Contracts\Console\Kernel ( https://laravel.com/api/6.x/Illuminate/Contracts/Console/Kernel.html ) artisan

Auth Illuminate\Auth\AuthManager ( https://laravel.com/api/6.x/Illuminate/Auth/AuthManager.html ) auth

Auth (Instance) Illuminate\Contracts\Auth\Guard ( https://laravel.com/api/6.x/Illuminate/Contracts/Auth/Guard.html ) auth.driver

Blade Illuminate\View\Compilers\BladeCompiler ( https://laravel.com/api/6.x/Illuminate/View/Compilers/BladeCompiler.html ) blade.compiler

Broadcast Illuminate\Contracts\Broadcasting\Factory ( https://laravel.com/api/6.x/Illuminate/Contracts/Broadcasting/Factory.html )

Broadcast (Instance) Illuminate\Contracts\Broadcasting\Broadcas... ( https://laravel.com/api/6.x/Illuminate/Contracts/Broadcasting/Broadcaster.html )

Bus Illuminate\Contracts\Bus\Dispatcher ( https://laravel.com/api/6.x/Illuminate/Contracts/Bus/Dispatcher.html )

Cache Illuminate\Cache\CacheManager ( https://laravel.com/api/6.x/Illuminate/Cache/CacheManager.html ) cache

Cache (Instance) Illuminate\Cache\Repository ( https://laravel.com/api/6.x/Illuminate/Cache/Repository.html ) cache.store

Config Illuminate\Config\Repository ( https://laravel.com/api/6.x/Illuminate/Config/Repository.html ) config

Cookie Illuminate\Cookie\CookieJar ( https://laravel.com/api/6.x/Illuminate/Cookie/CookieJar.html ) cookie

Crypt Illuminate\Encryption\Encrypter ( https://laravel.com/api/6.x/Illuminate/Encryption/Encrypter.html ) encrypter

DB Illuminate\Database\DatabaseManager ( https://laravel.com/api/6.x/Illuminate/Database/DatabaseManager.html ) db

DB (Instance) Illuminate\Database\Connection ( https://laravel.com/api/6.x/Illuminate/Database/Connection.html ) db.connection

Event Illuminate\Events\Dispatcher ( https://laravel.com/api/6.x/Illuminate/Events/Dispatcher.html ) events

File Illuminate\Filesystem\Filesystem ( https://laravel.com/api/6.x/Illuminate/Filesystem/Filesystem.html ) files

Gate Illuminate\Contracts\Auth\Access\Gate ( https://laravel.com/api/6.x/Illuminate/Contracts/Auth/Access/Gate.html )

Hash Illuminate\Contracts\Hashing\Hasher ( https://laravel.com/api/6.x/Illuminate/Contracts/Hashing/Hasher.html ) hash

Lang Illuminate\Translation\Translator ( https://laravel.com/api/6.x/Illuminate/Translation/Translator.html ) translator

Log Illuminate\Log\LogManager ( https://laravel.com/api/6.x/Illuminate/Log/LogManager.html ) log

Mail Illuminate\Mail\Mailer ( https://laravel.com/api/6.x/Illuminate/Mail/Mailer.html ) mailer

Notification Illuminate\Notifications\ChannelManager ( https://laravel.com/api/6.x/Illuminate/Notifications/ChannelManager.html )

Password Illuminate\Auth\Passwords\PasswordBrokerMa... ( https://laravel.com/api/6.x/Illuminate/Auth/Passwords/PasswordBrokerManager.html ) auth.password

Password (Instance) Illuminate\Auth\Passwords\PasswordBroker ( https://laravel.com/api/6.x/Illuminate/Auth/Passwords/PasswordBroker.html ) auth.password.broker

Queue Illuminate\Queue\QueueManager ( https://laravel.com/api/6.x/Illuminate/Queue/QueueManager.html ) queue

Queue (Instance) Illuminate\Contracts\Queue\Queue ( https://laravel.com/api/6.x/Illuminate/Contracts/Queue/Queue.html ) queue.connection

Queue (Base Class) Illuminate\Queue\Queue ( https://laravel.com/api/6.x/Illuminate/Queue/Queue.html )

Redirect Illuminate\Routing\Redirector ( https://laravel.com/api/6.x/Illuminate/Routing/Redirector.html ) redirect

Redis Illuminate\Redis\RedisManager ( https://laravel.com/api/6.x/Illuminate/Redis/RedisManager.html ) redis

Redis (Instance) Illuminate\Redis\Connections\Connection ( https://laravel.com/api/6.x/Illuminate/Redis/Connections/Connection.html ) redis.connection

Request Illuminate\Http\Request ( https://laravel.com/api/6.x/Illuminate/Http/Request.html ) request

Response Illuminate\Contracts\Routing\ResponseFacto... ( https://laravel.com/api/6.x/Illuminate/Contracts/Routing/ResponseFactory.html )

Response (Instance) Illuminate\Http\Response ( https://laravel.com/api/6.x/Illuminate/Http/Response.html )

Route Illuminate\Routing\Router ( https://laravel.com/api/6.x/Illuminate/Routing/Router.html ) router

Schema Illuminate\Database\Schema\Builder ( https://laravel.com/api/6.x/Illuminate/Database/Schema/Builder.html )

Session Illuminate\Session\SessionManager ( https://laravel.com/api/6.x/Illuminate/Session/SessionManager.html ) session

Session (Instance) Illuminate\Session\Store ( https://laravel.com/api/6.x/Illuminate/Session/Store.html ) session.store

Storage Illuminate\Filesystem\FilesystemManager ( https://laravel.com/api/6.x/Illuminate/Filesystem/FilesystemManager.html ) filesystem

Storage (Instance) Illuminate\Contracts\Filesystem\Filesystem ( https://laravel.com/api/6.x/Illuminate/Contracts/Filesystem/Filesystem.html ) filesystem.disk

URL Illuminate\Routing\UrlGenerator ( https://laravel.com/api/6.x/Illuminate/Routing/UrlGenerator.html ) url

Validator Illuminate\Validation\Factory ( https://laravel.com/api/6.x/Illuminate/Validation/Factory.html ) validator

Validator (Instance) Illuminate\Validation\Validator ( https://laravel.com/api/6.x/Illuminate/Validation/Validator.html )

View Illuminate\View\Factory ( https://laravel.com/api/6.x/Illuminate/View/Factory.html ) view

View (Instance) Illuminate\View\View ( https://laravel.com/api/6.x/Illuminate/View/View.html )

--

--

五、契约

https://learnku.com/docs/laravel/6.x/contracts/5134

--

1、介绍

Laravel 的契约是一组接口，它们由框架提供并定义了核心服务。 例如，Illuminate\Contracts\Queue\Queue 契约 定义了队列任务需要的方法，而 Illuminate\Contracts\Mail\Mailer 契约定义了发送邮件需要的方法。

每一个契约都有框架提供的相应的实现。例如， Laravel 提供了对多种驱动的队列实现，和一个由 SwiftMailer( https://swiftmailer.symfony.com/ ) 驱动的邮件实现。

所有 Laravel 契约都在 它们各自的 GitHub 仓库( https://github.com/illuminate/contracts )。这为所有可用的契约以及扩展包开发者们可能用到的单个、解耦的包，提供了一个快速参考入口。

(1)、契约 Vs. Facades

Laravel 的 Facades( https://learnku.com/docs/laravel/6.x/facades ) 和辅助函数提供了一种简便方式来使用 Laravel 服务而无需用到类型提示，也可在服务容器外部解析契约。多数情况下，每个 Facade 都有一个等效的契约。

和 Facades （不须要在你类中的构造函数去引用依赖）不同的是，契约允许你给自己的类定义明确的依赖。一些开发者更喜欢依赖被明确地定义出来，所以更倾向于使用契约，而其他开发者则享受于 Facades 带来的方便。

Tip：在大多数应用中，无论你更喜欢 Facades 还是契约，都是没问题的。然而如果你在搭建扩展包，那你应该强烈考虑使用契约，因为他们更便于在包的上下文中做测试。

--

2、何时使用契约

综上所述，使用契约还是 Facades 很大程度上取决于你个人或者团队的喜好。契约和 Facades 均可以用来构建健壮的、充分测试过的 Laravel 应用。只要你保持类的职责单一，你会发现使用契约和 Facades 的实际差别是非常小的。

然而，你也许仍有许多关于契约的问题。比方说，为啥都用接口 ？用接口不是更复杂吗？让我们在接下来的内容（「低耦合」与「简明性」）中，提炼出原因。

(1)、低耦合

首先，让我们来看一些缓存实现的高耦合代码。假设有下面代码。

<?php

namespace App\Orders;

class Repository

{

/\*\*

\* 缓存实例

\*/

protected $cache;

/\*\*

\* 创建一个新的仓库实例

\*

\* @param \SomePackage\Cache\Memcached $cache

\* @return void

\*/

public function \_\_construct(\SomePackage\Cache\Memcached $cache)

{

$this->cache = $cache;

}

/\*\*

\* 根据 ID 获取订单

\*

\* @param int $id

\* @return Order

\*/

public function find($id)

{

if ($this->cache->has($id)) {

//

}

}

}

?>

在这个类中，代码与给定的缓存实现形成高度耦合。它的高度耦合是因为我们依赖了一个扩展包中具体的缓存类。如果该扩展包的 API 变了，那么我们的代码也将必须做出修改。

同理，如果我们想要将底层的缓存技术 （ Memcached ） 替换成另一种缓存技术 （ Redis ），我们得再次修改我们的代码库。我们的代码库不应该对谁提供的数据或者数据是怎么提供的有太多了解。

我们可以通过依赖一个简单的与扩展包无关的接口来改进我们的代码，来替代之前的实现方式。

<?php

namespace App\Orders;

use Illuminate\Contracts\Cache\Repository as Cache;

class Repository

{

/\*\*

\* 缓存实例

\*/

protected $cache;

/\*\*

\* 创建一个新的仓库实例.

\*

\* @param Cache $cache

\* @return void

\*/

public function \_\_construct(Cache $cache)

{

$this->cache = $cache;

}

}

?>

现在的代码不与任何特定的扩展包耦合了，甚至与 Laravel 都是无关的。由于契约扩展包不包含任何实现和依赖，你可以轻松地为给定的契约编写替代实现的代码，从而可以在不修改任何缓存代码的情况下替换缓存的实现。

(2)、简明

当 Laravel 的所有服务都在简单的接口中简洁地定义时，很容易就可以确定给定服务提供的功能。契约是框架功能的简洁文档。

此外，当您依赖简单的接口时，您的代码更易于理解和维护。您可以参考一个简单，干净的接口，而不是在大型复杂的类中跟踪哪些方法可用。

--

3、如何使用契约

那么，你如何实现契约呢？它实际上非常简单。

Laravel 中的许多类都通过服务容器( https://learnku.com/docs/laravel/6.x/container )解析，包括控制器，事件监听器，中间件，队列任务，甚至路由闭包。因此，要获得契约的接口实现，你可以在正在解析的类的构造函数中「类型提示」接口。

//

//// 例如，看看这个事件监听器

<?php

namespace App\Listeners;

use App\User;

use App\Events\OrderWasPlaced;

use Illuminate\Contracts\Redis\Factory;

class CacheOrderInformation

{

/\*\*

\* Redis工厂实例

\*/

protected $redis;

/\*\*

\* 创建新的事件处理程序实例

\*

\* @param Factory $redis

\* @return void

\*/

public function \_\_construct(Factory $redis)

{

$this->redis = $redis;

}

/\*\*

\* 处理事件.

\*

\* @param OrderWasPlaced $event

\* @return void

\*/

public function handle(OrderWasPlaced $event)

{

//

}

}

?>

解析事件侦听器后，服务容器将读取类的构造函数上的类型提示，并注入适当的值。要了解有关在服务容器中注册内容的更多信息，请查看其文档( https://learnku.com/docs/laravel/6.x/container )。

--

4、契约参考

此表提供了所有 Laravel 契约及其等效 facades 的快速参考。

契约/参考Facade

Illuminate\Contracts\Auth\Access\Authoriza... ( https://github.com/illuminate/contracts/blob/laravel/6.x/Auth/Access/Authorizable.php )

Illuminate\Contracts\Auth\Access\Gate ( https://github.com/illuminate/contracts/blob/laravel/6.x/Auth/Access/Gate.php ) Gate

Illuminate\Contracts\Auth\Authenticatable ( https://github.com/illuminate/contracts/blob/laravel/6.x/Auth/Authenticatable.php )

Illuminate\Contracts\Auth\CanResetPassword ( https://github.com/illuminate/contracts/blob/laravel/6.x/Auth/CanResetPassword.php )

Illuminate\Contracts\Auth\Factory ( https://github.com/illuminate/contracts/blob/laravel/6.x/Auth/Factory.php ) Auth

Illuminate\Contracts\Auth\Guard ( https://github.com/illuminate/contracts/blob/laravel/6.x/Auth/Guard.php ) Auth::guard()

Illuminate\Contracts\Auth\PasswordBroker ( https://github.com/illuminate/contracts/blob/laravel/6.x/Auth/PasswordBroker.php ) Password::broker()

Illuminate\Contracts\Auth\PasswordBrokerFa... ( https://github.com/illuminate/contracts/blob/laravel/6.x/Auth/PasswordBrokerFactory.php ) Password

Illuminate\Contracts\Auth\StatefulGuard ( https://github.com/illuminate/contracts/blob/laravel/6.x/Auth/StatefulGuard.php )

Illuminate\Contracts\Auth\SupportsBasicAut... ( https://github.com/illuminate/contracts/blob/laravel/6.x/Auth/SupportsBasicAuth.php )

Illuminate\Contracts\Auth\UserProvider ( https://github.com/illuminate/contracts/blob/laravel/6.x/Auth/UserProvider.php )

Illuminate\Contracts\Bus\Dispatcher ( https://github.com/illuminate/contracts/blob/laravel/6.x/Bus/Dispatcher.php ) Bus

Illuminate\Contracts\Bus\QueueingDispatche... ( https://github.com/illuminate/contracts/blob/laravel/6.x/Bus/QueueingDispatcher.php ) Bus::dispatchToQueue()

Illuminate\Contracts\Broadcasting\Factory ( https://github.com/illuminate/contracts/blob/laravel/6.x/Broadcasting/Factory.php ) Broadcast

Illuminate\Contracts\Broadcasting\Broadcas... ( https://github.com/illuminate/contracts/blob/laravel/6.x/Broadcasting/Broadcaster.php ) Broadcast::connection()

Illuminate\Contracts\Broadcasting\ShouldBr... ( https://github.com/illuminate/contracts/blob/laravel/6.x/Broadcasting/ShouldBroadcast.php )

Illuminate\Contracts\Broadcasting\ShouldBr... ( https://github.com/illuminate/contracts/blob/laravel/6.x/Broadcasting/ShouldBroadcastNow.php )

Illuminate\Contracts\Cache\Factory ( https://github.com/illuminate/contracts/blob/laravel/6.x/Cache/Factory.php ) Cache

Illuminate\Contracts\Cache\Lock ( https://github.com/illuminate/contracts/blob/laravel/6.x/Cache/Lock.php )

Illuminate\Contracts\Cache\LockProvider ( https://github.com/illuminate/contracts/blob/laravel/6.x/Cache/LockProvider.php )

Illuminate\Contracts\Cache\Repository ( https://github.com/illuminate/contracts/blob/laravel/6.x/Cache/Repository.php ) Cache::driver()

Illuminate\Contracts\Cache\Store ( https://github.com/illuminate/contracts/blob/laravel/6.x/Cache/Store.php )

Illuminate\Contracts\Config\Repository ( https://github.com/illuminate/contracts/blob/laravel/6.x/Config/Repository.php ) Config

Illuminate\Contracts\Console\Application ( https://github.com/illuminate/contracts/blob/laravel/6.x/Console/Application.php )

Illuminate\Contracts\Console\Kernel ( https://github.com/illuminate/contracts/blob/laravel/6.x/Console/Kernel.php ) Artisan

Illuminate\Contracts\Container\Container ( https://github.com/illuminate/contracts/blob/laravel/6.x/Container/Container.php ) App

Illuminate\Contracts\Cookie\Factory ( https://github.com/illuminate/contracts/blob/laravel/6.x/Cookie/Factory.php ) Cookie

Illuminate\Contracts\Cookie\QueueingFactor... ( https://github.com/illuminate/contracts/blob/laravel/6.x/Cookie/QueueingFactory.php ) Cookie::queue()

Illuminate\Contracts\Database\ModelIdentif... ( https://github.com/illuminate/contracts/blob/laravel/6.x/Database/ModelIdentifier.php )

Illuminate\Contracts\Debug\ExceptionHandle... ( https://github.com/illuminate/contracts/blob/laravel/6.x/Debug/ExceptionHandler.php )

Illuminate\Contracts\Encryption\Encrypter ( https://github.com/illuminate/contracts/blob/laravel/6.x/Encryption/Encrypter.php ) Crypt

Illuminate\Contracts\Events\Dispatcher ( https://github.com/illuminate/contracts/blob/laravel/6.x/Events/Dispatcher.php ) Event

Illuminate\Contracts\Filesystem\Cloud ( https://github.com/illuminate/contracts/blob/laravel/6.x/Filesystem/Cloud.php ) Storage::cloud()

Illuminate\Contracts\Filesystem\Factory ( https://github.com/illuminate/contracts/blob/laravel/6.x/Filesystem/Factory.php ) Storage

Illuminate\Contracts\Filesystem\Filesystem ( https://github.com/illuminate/contracts/blob/laravel/6.x/Filesystem/Filesystem.php ) Storage::disk()

Illuminate\Contracts\Foundation\Applicatio... ( https://github.com/illuminate/contracts/blob/laravel/6.x/Foundation/Application.php ) App

Illuminate\Contracts\Hashing\Hasher ( https://github.com/illuminate/contracts/blob/laravel/6.x/Hashing/Hasher.php ) Hash

Illuminate\Contracts\Http\Kernel ( https://github.com/illuminate/contracts/blob/laravel/6.x/Http/Kernel.php )

Illuminate\Contracts\Mail\MailQueue ( https://github.com/illuminate/contracts/blob/laravel/6.x/Mail/MailQueue.php ) Mail::queue()

Illuminate\Contracts\Mail\Mailable ( https://github.com/illuminate/contracts/blob/laravel/6.x/Mail/Mailable.php )

Illuminate\Contracts\Mail\Mailer ( https://github.com/illuminate/contracts/blob/laravel/6.x/Mail/Mailer.php ) Mail

Illuminate\Contracts\Notifications\Dispatc... ( https://github.com/illuminate/contracts/blob/laravel/6.x/Notifications/Dispatcher.php ) Notification

Illuminate\Contracts\Notifications\Factory ( https://github.com/illuminate/contracts/blob/laravel/6.x/Notifications/Factory.php ) Notification

Illuminate\Contracts\Pagination\LengthAwar... ( https://github.com/illuminate/contracts/blob/laravel/6.x/Pagination/LengthAwarePaginator.php )

Illuminate\Contracts\Pagination\Paginator ( https://github.com/illuminate/contracts/blob/laravel/6.x/Pagination/Paginator.php )

Illuminate\Contracts\Pipeline\Hub ( https://github.com/illuminate/contracts/blob/laravel/6.x/Pipeline/Hub.php )

Illuminate\Contracts\Pipeline\Pipeline ( https://github.com/illuminate/contracts/blob/laravel/6.x/Pipeline/Pipeline.php )

Illuminate\Contracts\Queue\EntityResolver ( https://github.com/illuminate/contracts/blob/laravel/6.x/Queue/EntityResolver.php )

Illuminate\Contracts\Queue\Factory ( https://github.com/illuminate/contracts/blob/laravel/6.x/Queue/Factory.php ) Queue

Illuminate\Contracts\Queue\Job ( https://github.com/illuminate/contracts/blob/laravel/6.x/Queue/Job.php )

Illuminate\Contracts\Queue\Monitor ( https://github.com/illuminate/contracts/blob/laravel/6.x/Queue/Monitor.php ) Queue

Illuminate\Contracts\Queue\Queue ( https://github.com/illuminate/contracts/blob/laravel/6.x/Queue/Queue.php ) Queue::connection()

Illuminate\Contracts\Queue\QueueableCollec... ( https://github.com/illuminate/contracts/blob/laravel/6.x/Queue/QueueableCollection.php )

Illuminate\Contracts\Queue\QueueableEntity ( https://github.com/illuminate/contracts/blob/laravel/6.x/Queue/QueueableEntity.php )

Illuminate\Contracts\Queue\ShouldQueue ( https://github.com/illuminate/contracts/blob/laravel/6.x/Queue/ShouldQueue.php )

Illuminate\Contracts\Redis\Factory ( https://github.com/illuminate/contracts/blob/laravel/6.x/Redis/Factory.php ) Redis

Illuminate\Contracts\Routing\BindingRegist... ( https://github.com/illuminate/contracts/blob/laravel/6.x/Routing/BindingRegistrar.php ) Route

Illuminate\Contracts\Routing\Registrar ( https://github.com/illuminate/contracts/blob/laravel/6.x/Routing/Registrar.php ) Route

Illuminate\Contracts\Routing\ResponseFacto... ( https://github.com/illuminate/contracts/blob/laravel/6.x/Routing/ResponseFactory.php ) Response

Illuminate\Contracts\Routing\UrlGenerator ( https://github.com/illuminate/contracts/blob/laravel/6.x/Routing/UrlGenerator.php ) URL

Illuminate\Contracts\Routing\UrlRoutable ( https://github.com/illuminate/contracts/blob/laravel/6.x/Routing/UrlRoutable.php )

Illuminate\Contracts\Session\Session ( https://github.com/illuminate/contracts/blob/laravel/6.x/Session/Session.php ) Session::driver()

Illuminate\Contracts\Support\Arrayable ( https://github.com/illuminate/contracts/blob/laravel/6.x/Support/Arrayable.php )

Illuminate\Contracts\Support\Htmlable ( https://github.com/illuminate/contracts/blob/laravel/6.x/Support/Htmlable.php )

Illuminate\Contracts\Support\Jsonable ( https://github.com/illuminate/contracts/blob/laravel/6.x/Support/Jsonable.php )

Illuminate\Contracts\Support\MessageBag ( https://github.com/illuminate/contracts/blob/laravel/6.x/Support/MessageBag.php )

Illuminate\Contracts\Support\MessageProvid... ( https://github.com/illuminate/contracts/blob/laravel/6.x/Support/MessageProvider.php )

Illuminate\Contracts\Support\Renderable ( https://github.com/illuminate/contracts/blob/laravel/6.x/Support/Renderable.php )

Illuminate\Contracts\Support\Responsable ( https://github.com/illuminate/contracts/blob/laravel/6.x/Support/Responsable.php )

Illuminate\Contracts\Translation\Loader ( https://github.com/illuminate/contracts/blob/laravel/6.x/Translation/Loader.php )

Illuminate\Contracts\Translation\Translato... ( https://github.com/illuminate/contracts/blob/laravel/6.x/Translation/Translator.php ) Lang

Illuminate\Contracts\Validation\Factory ( https://github.com/illuminate/contracts/blob/laravel/6.x/Validation/Factory.php ) Validator

Illuminate\Contracts\Validation\ImplicitRu... ( https://github.com/illuminate/contracts/blob/laravel/6.x/Validation/ImplicitRule.php )

Illuminate\Contracts\Validation\Rule ( https://github.com/illuminate/contracts/blob/laravel/6.x/Validation/Rule.php )

Illuminate\Contracts\Validation\ValidatesW... ( https://github.com/illuminate/contracts/blob/laravel/6.x/Validation/ValidatesWhenResolved.php )

Illuminate\Contracts\Validation\Validator ( https://github.com/illuminate/contracts/blob/laravel/6.x/Validation/Validator.php ) Validator::make()

Illuminate\Contracts\View\Engine ( https://github.com/illuminate/contracts/blob/laravel/6.x/View/Engine.php )

Illuminate\Contracts\View\Factory ( https://github.com/illuminate/contracts/blob/laravel/6.x/View/Factory.php ) View

Illuminate\Contracts\View\View ( https://github.com/illuminate/contracts/blob/laravel/6.x/View/View.php ) View::make()

--

--

第四卷 基础功能

--

--

一、路由

https://learnku.com/docs/laravel/6.x/routing/5135

--

1、基本路由

构建基本的路由只需要一个 URI 与一个 闭包 ，这里提供了一个非常简单优雅定义路由的方法。

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

return 'Hello World';

});

(1)、默认路由文件

所有的 Laravel 路由都在 routes 目录中的路由文件中定义，这些文件都由框架自动加载。routes/web.php 文件用于定义 web 界面的路由。这里面的路由都会被分配给 web 中间件组，它提供了会话状态和 CSRF 保护等功能。定义在 routes/api.php 中的路由都是无状态的，并且被分配了 api 中间件组。

大多数的应用构建，都是以在 routes/web.php 文件定义路由开始的。可以通过在浏览器中输入定义的路由 URL 来访问 routes/web.php 中定义的路由。例如，你可以在浏览器中输入 http://your-app.dev/user 来访问以下路由。

//

// http://your-app.dev/user

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

//

routes/api.php 文件中定义的路由通过 RouteServiceProvider 被嵌套到一个路由组里面。在这个路由组中，会自动添加 URL 前缀 /api 到此文件中的每个路由，这样你就无需再手动添加了。你可以在 RouteServiceProvider 类中修改此前缀以及其他路由组选项。

(2)、可用的路由方法

路由器允许你注册能响应任何 HTTP 请求的路由。

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

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

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

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

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

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

有的时候你可能需要注册一个可响应多个 HTTP 请求的路由，这时你可以使用 match 方法，也可以使用 any 方法注册一个实现响应所有 HTTP 请求的路由。

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

//

});

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

//

});

(3)、CSRF 保护

指向 web 路由文件中定义的 POST、PUT 或 DELETE 路由的任何 HTML 表单都应该包含一个 CSRF 令牌字段，否则，这个请求将会被拒绝。可以在 CSRF 文档( https://learnku.com/docs/laravel/6.x/csrf ) 中阅读有关 CSRF 更多信息。

<form method="POST" action="/profile">

@csrf

...

</form>

(4)、路由重定向

如果要定义重定向到另一个 URI 的路由，可以使用 Route::redirect 方法。这个方法可以快速的实现重定向，而不再需要去定义完整的路由或者控制器。

// 重定向

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

//

// Route::redirect 默认会返回状态码 302 。 你可以通过第三个参数自定义返回码。

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

//

// 你也可以使用 Route::permanentRedirect 方法来返回 301 状态码。

Route::permanentRedirect('/here', '/there');

(5)、视图路由

如果你的路由只需要返回一个视图，可以使用 Route::view 方法。它和 redirect 一样方便，不需要定义完整的路由或控制器。view 方法有三个参数，其中第一个是必填参数，是包含视图名称的 URI 。第二个也是必填参数，是需要渲染的视图名称。此外，还可以通过数组传递一组数据给视图，作为可选的第三个参数。

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

Route::view('/welcome', 'welcome', ['name' => 'Taylor']);

--

2、路由参数

(1)、必选参数 ( 必填参数 )

当然，有时需要在路由中捕获一些 URL 片段。例如，从 URL 中捕获用户的 ID，可以通过定义路由参数来执行此操作。

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

return 'User '.$id;

});

也可以根据需要在路由中定义多个参数。

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

//

});

路由的参数通常都会被放在 {} 内，并且参数名只能为字母，同时路由参数不能包含 - 符号，如果有需要，可以用下划线 (\_) 代替。路由参数会按顺序依次被注入到路由回调或者控制器中，而不受回调或者控制器的参数名称的影响。

(2)、可选参数

有时，你可能需要指定一个路由参数，但你希望这个参数是可选的。你可以在参数后面加上 ? 标记来实现，但前提是要确保路由的相应变量有默认值。

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

return $name;

});

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

return $name;

});

(3)、正则表达式参数 (正则表达式约束)

你可以使用路由实例上的 where 方法约束路由参数的格式。where 方法接受参数名称和定义参数应如何约束的正则表达式。

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

//

})->where('name', '[A-Za-z]+');

//

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

//

})->where('id', '[0-9]+');

//

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

//

})->where(['id' => '[0-9]+', 'name' => '[a-z]+']);

(4)、全局约束

如果你希望某个具体的路由参数都遵循同一个正则表达式的约束，就使用 pattern 方法在 RouteServiceProvider 的 boot 方法中定义这些模式。

//

//// RouteServiceProvider.php

/\*\*

\* 定义你的路由模型绑定, pattern 过滤器等。

\*

\* @return void

\*/

public function boot()

{

Route::pattern('id', '[0-9]+');

parent::boot();

}

一旦定义好之后，便会自动应用这些规则到所有使用该参数名称的路由上。

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

// 只有在 id 为数字时才执行

});

(5)、Laravel 路由组件允许除 / 之外的所有字符。你必须使用 where 条件正则表达式显式地允许 / 成为占位符的一部分。

Route::get('search/{search}', function ($search) {

return $search;

})->where('search', '.\*');

{注意} 编码正斜杠字符仅在最后一个路由段中是支持的。

--

3、命名路由 (路由命名)

--路由命名可以方便地为指定路由生成 URL 或者重定向。通过在路由定义上链式调用 name 方法可以指定路由名称。

Route::get('user/profile', function () {

//

})->name('profile');

你还可以指定控制器行为的路由名称。

Route::get('user/profile', 'UserProfileController@show')->name('profile');

(1)、生成指定路由的 URL

为路由指定了名称后，就可以使用全局辅助函数 route 来生成链接或者重定向到该路由。

// 生成 URL...

$url = route('profile');

// 生成重定向...

return redirect()->route('profile');

//

如果是有定义参数的命名路由，可以把参数作为 route 函数的第二个参数传入，指定的参数将会自动插入到 URL 中对应的位置。

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

//

})->name('profile');

$url = route('profile', ['id' => 1]);

(2)、检查当前路由

如果你想判断当前请求是否指向了某个命名过的路由，你可以调用路由实例上的 named 方法。例如，你可以在路由中间件中检查当前路由名称。

/\*\*

\* 处理一个请求

\*

\* @param \Illuminate\Http\Request $request

\* @param \Closure $next

\* @return mixed

\*/

public function handle($request, Closure $next)

{

if ($request->route()->named('profile')) {

//

}

return $next($request);

}

--

4、路由组

路由组允许你在大量路由之间共享路由属性，例如中间件或命名空间，而不需要为每个路由单独定义这些属性。共享属性应该以数组的形式传入 Route::group 方法的第一个参数中。

嵌套的组尝试智能地「合并」其属性及其父组。中间件和 where 条件语句在附加名称、命名空间和前缀时被合并。在适当的情况下，命名空间的分隔符和斜线会被自动添加到 URI 前缀中。

(1)、中间件

要给路由组中所有的路由分配中间件，可以在 group 之前调用 middleware 方法，中间件会依照它们在数组中列出的顺序来运行。

Route::middleware(['first', 'second'])->group(function () {

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

// 使用 first 和 second 中间件

});

Route::get('user/profile', function () {

// 使用 first 和 second 中间件

});

});

(2)、命名空间

另一个常见用例是使用 namespace 方法将相同的 PHP 命名空间分配给路由组的中所有的控制器。

Route::namespace('Admin')->group(function () {

// 在 「App\Http\Controllers\Admin」 命名空间下的控制器

});

请记住，默认情况下，RouteServiceProvider 会在命名空间组中引入你的路由文件，让你不用指定完整的 App\Http\Controllers 命名空间前缀就能注册控制器路由。因此，你只需要指定命名空间 App\Http\Controllers 之后的部分。

(3)、子域名路由

路由组也可以用来处理子域名。子域名可以像路由 URI 一样被分配路由参数，允许你获取一部分子域名作为参数给路由或控制器使用。可以在定义 group 之前调用 domain 方法来指定子域名。

Route::domain('{account}.myapp.com')->group(function () {

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

//

});

});

{注意} 为了确保您的子域名路由是可访问的，您应该在注册根域名路由之前注册子域名路由。这将防止根域名路由覆盖具有相同 URI 路径的子域名路由。

(4)、路由前缀

可以用 prefix 方法为路由组中给定的 URL 增加前缀。例如，你可以为组中所有路由的 URI 加上 admin 前缀。

Route::prefix('admin')->group(function () {

Route::get('users', function () {

// 匹配包含 「/admin/users」 的 URL

});

});

(5)、路由命名前缀 (路由名称前缀)

name 方法可以用来给路由组中的每个路由名称添加一个给定的字符串。 例如，您可能希望以 admin 为所有分组路由的名称加前缀。 给定的字符串与指定的路由名称前缀完全相同，因此我们将确保在前缀中提供尾部的 “.” 字符。

Route::name('admin.')->group(function () {

Route::get('users', function () {

// 指定路由名为 「admin.users」...

})->name('users');

});

--

5、路由模型绑定

当向路由或控制器行为注入模型 ID 时，就需要查询这个 ID 对应的模型。Laravel 为路由模型绑定提供了一个直接自动将模型实例注入到路由中的方法。例如，你可以注入与给定 ID 匹配的整个 User 模型实例，而不是注入用户的 ID。

(1)、隐式绑定

Laravel 会自动处理定义在路由或控制器行为中，与类型提示的变量名相匹配的路由段名称的 Eloquent 模型，例如。

Route::get('api/users/{user}', function (App\User $user) {

return $user->email;

});

在这个例子中，由于 $user 变量被类型提示为 Eloquent 模型 App\User，变量名称又与 URI 中的 {user} 匹配，因此，Laravel 会自动注入与请求 URI 中传入的 ID 匹配的用户模型实例。如果在数据库中找不到对应的模型实例，将会自动生成 404 异常。

(2)、自定义键名

如果你想要模型绑定在检索给定的模型类时使用除 id 之外的数据库字段，你可以在 Eloquent 模型上重写 getRouteKeyName 方法。

/\*\*

\* 获取该模型的路由的自定义键名

\*

\* @return string

\*/

public function getRouteKeyName()

{

return 'slug';

}

(3)、显式绑定

要注册显式绑定，使用路由器的 model 方法来为给定参数指定类。在 RouteServiceProvider 类中的 boot 方法内定义这些显式模型绑定。

public function boot()

{

parent::boot();

Route::model('user', App\User::class);

}

然后，定义一个包含 {user} 参数的路由。

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

//

});

因为我们已经将所有 {user} 参数绑定至 App\User 模型，所以 User 实例将被注入该路由。所以例如，profile/1 的请求会注入数据库中 ID 为 1 的 User 实例。

如果在数据库中找不到匹配的模型实例，就会自动抛出一个 404 异常。

(4)、自定义逻辑解析

如果你想要使用自定义的解析逻辑，就使用 Route::bind 方法。传递到 bind 方法的 闭包 会接受 URI 中大括号对应的值，并且返回你想要在该路由中注入的类的实例。

/\*\*

\* 启动任意应用服务

\*

\* @return void

\*/

public function boot()

{

parent::boot();

Route::bind('user', function ($value) {

return App\User::where('name', $value)->first() ?? abort(404);

});

}

或者，您可以重写 Eloquent 模型上的 resolveRouteBinding 方法。 此方法会接受 URI 中大括号对应的值，并且返回你想要在该路由中注入的类的实例。

/\*\*

\* 检索绑定值的模型

\*

\* @param mixed $value

\* @return \Illuminate\Database\Eloquent\Model|null

\*/

public function resolveRouteBinding($value)

{

return $this->where('name', $value)->first() ?? abort(404);

}

--

6、回退路由

使用 Route::fallback 方法， 你可以定义一个在没有其他路由可匹配传入的请求时，才执行的路由。一般来说，未处理的请求会通过应用程序的异常处理程序自动呈现 「404」 页面。 但是，因为你可以在 routes/web.php 文件中定义 fallback 路由，web 中间件组中的所有中间件都将应用到路由中。 当然，你也可以根据需要向这条路由中添加额外的中间件。

Route::fallback(function () {

//

});

{注意} 回退路由应始终是你应用程序注册的最后一个路由。

--

7、限流

Laravel 包含了一个 中间件( https://learnku.com/docs/laravel/laravel/6.x/middleware ) 用于控制应用程序对路由的访问。 如果想要使用， 请将 throttle 中间件分配给一个路由或者一个路由组。throttle 中间件会接收两个参数，这两个参数决定了在给定的分钟数内可以进行的最大请求数。例如，让我们指定一个经过身份验证并且用户每分钟访问频率不超过 60 次的路由组。

//

//// 经过身份验证并且用户每分钟访问频率不超过 60 次的路由组。

// 通过一个 : 来隔开中间件名称和参数来指定中间件参数。

Route::middleware('auth:api', 'throttle:60,1')->group(function () {

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

//

});

});

(1)、动态限流

你可以根据已验证的 User 模型的属性，指定动态请求的最大值。例如，如果你的 User 模型包含 rate\_limit 属性，则可以将属性名称传递给 throttle 中间件，以便它用于计算最大请求数。

Route::middleware('auth:api', 'throttle:rate\_limit,1')->group(function () {

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

//

});

});

(2)、独立访客和认证用户的限流

您可以为访客和经过身份验证的用户指定不同的访问控制。例如，可以为访客指定每分钟最多 10 次请求，为认证用户设置每分钟最多 60 次请求。

Route::middleware('throttle:10|60,1')->group(function () {

//

});

您还可以将此功能与动态限流相结合。例如，如果 User 模型包含一个 rate\_limit 属性，您可以将该属性的名称传递给 throttle 中间件，以便用于计算认证用户的最大请求数。

Route::middleware('auth:api', 'throttle:10|rate\_limit,1')->group(function () {

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

//

});

});

--

8、表单方法伪造

HTML 表单不支持 PUT 、PATCH 或 DELETE 行为。所以当你要从 HTML 表单中调用定义了 PUT、 PATCH 或 DELETE 行为的路由时，你将需要在表单中增加一个隐藏的 \_method 输入标签。使用 \_method 字段的值作为 HTTP 的请求方法。

<form action="/foo/bar" method="POST">

<input type="hidden" name="\_method" value="PUT">

<input type="hidden" name="\_token" value="{{ csrf\_token() }}">

</form>

//

你也可以使用 @method 模板指令生成 \_method 输入。

<form action="/foo/bar" method="POST">

@method('PUT')

@csrf

</form>

--

9、访问当前路由

你可以使用 Route facade 上的 current 、 currentRouteName 和 currentRouteAction 方法来访问处理传入请求的路由的信息。

$route = Route::current();

$name = Route::currentRouteName();

$action = Route::currentRouteAction();

如果你想知道所有可访问的方法，可以查看 API 文档，了解 路由 facade 基础类( https://laravel.com/api/6.x/Illuminate/Routing/Router.html ) 和 Route 实例( https://laravel.com/api/6.x/Illuminate/Routing/Route.html ) 。

--

--

二、中间件

https://learnku.com/docs/laravel/6.x/middleware/5136

--

1、介绍

中间件提供了一种方便的机制过滤进入应用程序的 HTTP 请求。例如，Laravel 包含一个验证用户身份的中间件。如果用户未能通过认证，中间件会把用户重定向到登录页面。反之，用户如果通过验证，中间件将把请求进一步转发到应用程序中。

当然，除了验证身份外，还可以编写其他的中间件来执行各种任务。例如：CORS 中间件可以负责为所有的应用返回的 responses 添加合适的响应头。日志中间件可以记录所有传入应用的请求。

Laravel 自带了一些中间件，包括身份验证、CSRF 保护等。所有的这些中间件都位于 app/Http/Middleware 目录。

--

2、定义中间件 (定义一个中间件)

//// 使用 make:middleware 命令来创建新的中间件。

php artisan make:middleware CheckAge

该命令会在 app/Http/Middleware 目录下生成新的 CheckAge 类，在这个中间件中，我们仅允许 age 参数大于 200 的请求对路由进行访问，否则将重定向到 home 页面。

<?php

namespace App\Http\Middleware;

use Closure;

class CheckAge

{

/\*\*

\* Handle an incoming request.

\*

\* @param \Illuminate\Http\Request $request

\* @param \Closure $next

\* @return mixed

\*/

public function handle($request, Closure $next)

{

if ($request->age <= 200) {

return redirect('home');

}

return $next($request);

}

}

?>

正如你所见，如果给定的 age 参数小于或者等于 200，这个中间件将会返回一个 HTTP 重定向到客户端；否则这个请求将会通过，进一步传递到应用层中。要让请求继续传递到应用层中（即允许「通过」中间件验证），只需将 $request 作为参数来调用函数 $next 。

最好将中间件想象成一系列层次， HTTP 请求必须通过它们才能进入你的应用层。每一层都会检查请求（是否符合中间件要求），而后决定通过或拒绝访问应用。

{提示} 所有的中间件都是通过 服务容器( https://learnku.com/docs/%7B%7Bversion%7D%7D/container ) 解析的，因此，你可以在你的中间件构造函数中键入你需要的任何依赖。

(1)、前置 & 后置中间件

中间件是在请求之前或之后执行，取决于中间件本身。例如，下面的中间件将在应用处理请求 之前 执行一些任务。

//

//// 前置中间件

<?php

namespace App\Http\Middleware;

use Closure;

class BeforeMiddleware

{

public function handle($request, Closure $next)

{

// 执行一些任务

return $next($request);

}

}

?>

然后，下面中间件是在应用请求 之后 执行一些任务。

//

//// 后置中间件

<?php

namespace App\Http\Middleware;

use Closure;

class AfterMiddleware

{

public function handle($request, Closure $next)

{

$response = $next($request);

// 执行一些任务

return $response;

}

}

?>

--

3、注册中间件

(1)、全局中间件

如果你希望中间件在应用处理每个 HTTP 请求期间运行。只需要在 app/Http/Kernel.php 中的 $middleware 属性中列出这个中间件。

(2)、为路由分配中间件

假设你想为指定的路由分配中间件 ，首先应该在 app/Http/Kernel.php 文件内为该中间件分配一个键。默认情况下，该类中的 $routeMiddleware 属性下包含了 Laravel 内置的中间件。若要加入自定义的中间件，只需把它附加到列表后并为其分配一个自定义键。例如。

//

// 在 App\Http\Kernel 类中...

protected $routeMiddleware = [

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

'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,

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

'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,

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

'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,

'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,

'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,

'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,

];

一旦在 HTTP 内核中定义好了中间件，就可以通过 middleware 方法将为路由分配中间件。

Route::get('admin/profile', function () {

//

})->middleware('auth');

//

// 你也可以为路由分配多个中间件。

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

//

})->middleware('first', 'second');

//

// 分配中间件时，还可以传递完整的类名。

use App\Http\Middleware\CheckAge;

Route::get('admin/profile', function () {

//

})->middleware(CheckAge::class);

(3)、中间件群组

某些时候你可能希望使用一个键把多个中间件打包成一个组，以便更容易地分配给路由。你可以使用 Http 核心的 $middlewareGroups 属性。

Laravel 内置了开箱即用的 web 和 api 中间件组，其中包含你可能希望应用于 Web UI 和 API 路由的常用中间件。

/\*\*

\* 应用程序的路由中间件组

\*

\* @var array

\*/

protected $middlewareGroups = [

'web' => [

\App\Http\Middleware\EncryptCookies::class,

\Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,

\Illuminate\Session\Middleware\StartSession::class,

\Illuminate\View\Middleware\ShareErrorsFromSession::class,

\App\Http\Middleware\VerifyCsrfToken::class,

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

],

'api' => [

'throttle:60,1',

'auth:api',

],

];

中间件组可以使用与单个中间件相同的语法将自身分配给路由和控制器动作。同样，中间件组使得一次将多个中间件分配给一个路由更加方便。

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

//

})->middleware('web');

//

Route::group(['middleware' => ['web']], function () {

//

});

Tip：RouteServiceProvider 默认将 web 中间件组自动应用到 routes/web.php。

(4)、中间件的排序 (排序中间件)

很少情况下，你可能需要中间件以特定的顺序执行，但是当它们被分配到路由时，你无法控制它们的顺序。在这种情况下，可以使用 app/Http/Kernel.php 文件的 $middlewarePriority 属性指定中间件优先级。

/\*\*

\* 中间件的优先级排序列表

\*

\* 将会强制非全局中间件始终保持给定的顺序。

\*

\* @var array

\*/

protected $middlewarePriority = [

\Illuminate\Session\Middleware\StartSession::class,

\Illuminate\View\Middleware\ShareErrorsFromSession::class,

\App\Http\Middleware\Authenticate::class,

\Illuminate\Session\Middleware\AuthenticateSession::class,

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

\Illuminate\Auth\Middleware\Authorize::class,

];

--

4、中间件参数

中间件还可以接收其他参数。例如，如果你的应用程序需要在执行给定操作之前验证用户是否为给定的「角色」 ，你可以创建一个 CheckRole 中间件，由它来接收「角色」名称作为附加参数。

附加的中间参数应该在 $next 参数之后传递给中间件。

//

//// CheckRole中间件

// 在执行给定操作之前验证用户是否为给定的「角色」。

<?php

namespace App\Http\Middleware;

use Closure;

class CheckRole

{

/\*\*

\* 处理传入的参数

\*

\* @param \Illuminate\Http\Request $request

\* @param \Closure $next

\* @param string $role

\* @return mixed

\*/

public function handle($request, Closure $next, $role)

{

if (! $request->user()->hasRole($role)) {

// Redirect...

}

return $next($request);

}

}

?>

//

//// CheckRole使用

// 定义路由时通过一个 “:” 来隔开中间件名称和参数来指定中间件参数。多个参数就使用逗号分隔。

Route::put('post/{id}', function ($id) {

//

})->middleware('role:editor');

--

5、Terminable 中间件

有时，在准备好 HTTP 响应之后，中间件可能需要做一些工作。例如，Laravel 内置的「session」 中间件会在完全准备好响应后将会话数据写入存储。如果你在中间件上定义了一个 terminate 方法，并且你使用的是 FastCGI ，那么它将会在响应准备发送到浏览器之后自动调用。

<?php

namespace Illuminate\Session\Middleware;

use Closure;

class StartSession

{

public function handle($request, Closure $next)

{

return $next($request);

}

public function terminate($request, $response)

{

// Store the session data...

}

}

?>

terminate 方法应该同时接收请求和响应。定义了这个中间件之后，别忘了将它添加到路由列表或者 app/Http/Kernel.php 文件的全局中间件中。

当你在中间件上调用 terminate 方法的时候， Laravel 将从 服务容器( https://learnku.com/docs/%7B%7Bversion%7D%7D/container )中解析出一个新的中间件实例。如果在调用 handle 和 terminate 方法的同时使用相同的中间件实例，请使用容器的 singleton 方法在容器中注册中间件。

--

--

三、CSRF 保护

https://learnku.com/docs/laravel/6.x/csrf/5137

--

1、介绍

Laravel 可以轻松保护你的应用程序免受 跨站请求伪造 (CSRF https://en.wikipedia.org/wiki/Cross-site\_request\_forgery ) 攻击。 跨站点请求伪造是一种恶意攻击，它凭借已通过身份验证的用户身份来运行未经过授权的命令。

Laravel 会自动为每个活跃的用户的会话生成一个 CSRF「令牌」。该令牌用于验证经过身份验证的用户是否是向应用程序发出请求的用户。

无论何时，当您在应用程序中定义 HTML 表单时，都应该在表单中包含一个隐藏的 CSRF 标记字段，以便 CSRF 保护中间件可以验证该请求，你可以使用 @csrf Blade 指令来生成令牌字段，如下。

<form method="POST" action="/profile">

@csrf

...

</form>

包含在 web 中间件组里 VerifyCsrfToken 中间件( https://learnku.com/docs/laravel/6.x/middleware )会自动验证请求里的令牌是否与存储在会话中令牌匹配。

(1)、CSRF 令牌 & JavaScript

当构建由 JavaScript 驱动的应用时，可以方便的让 JavaScript HTTP

函数库发起每一个请求时自动附上 CSRF 令牌。默认情况下，resources/js/bootstrap.js 文件中提供的 Axios HTTP 库会使用 cookie 中加密的 XSRF-TOKEN 的值然后在请求时自动发送 X-XSRF-TOKEN 标头。 如果不使用此库，则需要为应用程序手动配置此行为。

--

2、CSRF - 白名单

有时候你可能希望设置一组不需要的 CSRF 保护的 URL 。例如，如果你正在使用 Stripe( https://stripe.com/ ) 处理付款并使用了他们的 webhook 系统，你会需要从 CSRF 的保护中排除 Stripe webhook 处理程序路由，因为 Stripe 并不会给你的路由发送 CSRF 令牌。

典型做法，你可以把这类路由放在 routes/web.php 外，因为 RouteServiceProvider 的 web 中间件适用于该文件中的所有路由。不过，你也可以通过将这类 URL 添加到 VerifyCsrfToken 中间件的 $except 属性来排除对这类路由的 CSRF 保护，如下所示。

<?php

namespace App\Http\Middleware;

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

class VerifyCsrfToken extends Middleware

{

/\*\*

\* 从CSRF验证中排除的URI

\*

\* @var array

\*/

protected $except = [

'stripe/\*',

'http://example.com/foo/bar',

'http://example.com/foo/\*',

];

}

?>

Tip：当 运行测试( https://learnku.com/docs/laravel/6.x/testing ) 时， CSRF 中间件会自动禁用。

--

3、X-CSRF-Token

除了检查 POST 参数中的 CSRF 令牌外， VerifyCsrfToken 中间件还会检查 X-CSRF-TOKEN 请求头。你应该将令牌保存在 HTML meta 标签中，如下。

<meta name="csrf-token" content="{{ csrf\_token() }}">

然后，一旦你创建了 meta 标签，就可以指示像 jQuery 这样的库自动将令牌添加到所有请求的头信息中。还可以为基于 AJAX 的应用提供简单，方便的 CSRF 保护。如下。

$.ajaxSetup({

headers: {

'X-CSRF-TOKEN': $('meta[name="csrf-token"]').attr('content')

}

});

--

4、X-XSRF-Token

Laravel 将当前的 CSRF 令牌存储在一个 XSRF-TOKEN cookie

中，该 cookie 包含在框架生成的每个响应中。你可以使用 cookie 值来设置 X-XSRF-TOKEN 请求头。

这个 cookie 主要是作为一种方便的方式发送的，因为一些 JavaScript 框架和库，例如 Angular 和 Axios ，会自动将它的值放入 X-XSRF-TOKEN 头中。

Tip：默认情况下，resources/js/bootstrap.js 文件包含的 Axios HTTP 库，会自动为你发送。

--

--

四、控制器

https://learnku.com/docs/laravel/6.x/controllers/5138

--

1、简介

为了替代在路由文件中以闭包形式定义的所有的请求处理逻辑， 你可能想要使用控制类来组织这些行为。控制器能将相关的请求处理逻辑组成一个单独的类。控制器被存放在 app/Http/Controllers 目录。

--

2、基础控制器

(1)、定义控制器

下面是一个基础控制器类的例子。需要注意的是，该控制器继承了 Laravel 的基类控制器。该基类控制器提供了一些便利的方法，比如 middleware 方法，该方法可以为控制器行为添加中间件。

//

///1/ 定义控制器

<?php

namespace App\Http\Controllers;

use App\User;

use App\Http\Controllers\Controller;

class UserController extends Controller

{

/\*\*

\* 显示给用户的概要文件.

\*

\* @param int $id

\* @return View

\*/

public function show($id)

{

return view('user.profile', ['user' => User::findOrFail($id)]);

}

}

?>

//

///2/ 定义路由：你可以这样定义一个指向控制器行为的路由。

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

现在，当一个请求与指定路由的 URI 匹配时，UserController 控制器中的 show 方法将被执行。路由参数也将会被传递给该方法。

Tip：控制器并不是 必需 继承基础类。然而，如果控制器没有继承基础类，你将无法使用一些便捷的功能，比如 middleware，validate 和 dispatch 方法。

(2)、控制器 & 命名空间

需要着重指出的是，在定义控制器路由时我们不需要指定完整的控制器命名空间。因为 RouteServiceProvider 会在一个包含命名空间的路由组中加载路由文件，我们只需要指令类名中 App\Http\Controllers 命名空间之后的部分就可以了。

如果你选择将控制器放在 App\Http\Controllers 更深层次的目录中，需要使用相对于 App\Http\Controllers 作为根命名空间的指定类名。因此，如果你完整的控制器类名为 App\Http\Controllers\Photos\AdminController ，你在路由中应该采用如下的形式注册。

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

(3)、单个行为控制器

如果你想要定义一个只处理单个行为的控制器，你可以在控制器中放置一个 \_\_invoke 方法。

//

///1/ 定义控制器

<?php

namespace App\Http\Controllers;

use App\User;

use App\Http\Controllers\Controller;

class ShowProfile extends Controller

{

/\*\*

\* 显示给定用户的资料.

\*

\* @param int $id

\* @return View

\*/

public function \_\_invoke($id)

{

return view('user.profile', ['user' => User::findOrFail($id)]);

}

}

?>

//

///2/ 定义路由：当注册单个行为控制器的路由时，你不需要指明方法。

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

//

你可以通过 Artisan 命令工具里的 make:controller 命令中的 --invokable 选项来生成一个可调用的控制器。

php artisan make:controller ShowProfile --invokable

--

3、控制器中间件

//

//// 分配路由

Middleware 可以在路由文件中分配给控制器的路由。

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

//

//// 分配控制器

然而，在控制器的构造函数中指定中间件更为方便。使用控制器构造函数中的 middleware 方法，可以轻松地将中间件分配给控制器的操作。你甚至可以将中间件限制为控制器类上的某些方法。

class UserController extends Controller

{

/\*\*

\* 实例化一个新的控制器实例.

\*

\* @return void

\*/

public function \_\_construct()

{

$this->middleware('auth');

$this->middleware('log')->only('index');

$this->middleware('subscribed')->except('store');

}

}

//

//// 闭包中间件

同时，控制器还允许你使用一个闭包来注册中间件。这为不定义整个中间件类的情况下为单个控制器定义中间件提供了一种方便的方法。

$this->middleware(function ($request, $next) {

// ...

return $next($request);

});

Tip：你可以将中间件分配给控制器操作的一个子集；然而，它可能表明你的控制器正在变的复杂。建议你将控制器拆分为多个较小的控制器。

--

4、资源控制器

Laravel 的资源路由将典型的「CURD (增删改查)」路由分配给具有单行代码的控制器。例如，你希望创建一个控制器来处理保存 "照片" 应用的所有 HTTP 请求。使用 Artisan 命令 make:controller ，我们可以快速创建这样一个控制器。

//

///1/ 创建资源控制器

php artisan make:controller PhotoController --resource

这个命令将会生成一个控制器 app/Http/Controllers/PhotoController.php 。其中包括每个可用资源操作的方法。

//

///2/ 注册单个资源路由

// 接下来，你可以给控制器注册一个资源路由。

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

这个单一的路由声明创建了多个路由来处理资源上的各种行为。生成的控制器为每个行为保留了方法，包括了关于处理 HTTP 动词和 URLs 的声明注释。

//

///3/ 注册多个自由路由

你可以通过将数组传参到 resources 方法中的方式来一次性的创建多个资源控制器。

Route::resources([

'photos' => 'PhotoController',

'posts' => 'PostController'

]);

(1)、资源控制器操作处理

HTTP方法 | URI | 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

(2)、指定资源模型

如果你使用了路由模型绑定，并且想在资源控制器的方法中使用类型提示，你可以在生成控制器的时候使用 --model 选项。

php artisan make:controller PhotoController --resource --model=Photo

(3)、伪造表单方法

因为 HTML 的表单不能生成 PUT，PATCH，和 DELETE 请求，所以你需要添加一个隐藏的 \_method 字段来伪造 HTTP 动作。这个 Blade 指令 @method 可以为你创造这个字段。

<form action="/foo/bar" method="POST">

@method('PUT')

</form>

(4)、部分资源路由

Route::resource('photos', 'PhotoController')->only([

'index', 'show'

]);

//

Route::resource('photos', 'PhotoController')->except([

'create', 'store', 'update', 'destroy'

]);

(5)、API 资源路由

///1/ 定义单个API资源路由

当声明用于 APIs 的资源路由时，通常需要排除显示 HTML 模板的路由（如 create 和 edit ）。为了方便起见，你可以使用 apiResource 方法自动排除这两个路由。

Route::apiResource('photos', 'PhotoController');

//

///2/ 定义多个API资源路由

你可以传递一个数组给 apiResources 方法来同时注册多个 API 资源控制器。

Route::apiResources([

'photos' => 'PhotoController',

'posts' => 'PostController'

]);

//

///3/ 定义API资源控制器

// 要快速生成不包含 create 或 edit 方法的用于开发接口的资源控制器，请在执行 make:controller 命令时使用 --api 开关。

php artisan make:controller API/PhotoController --api

(6)、命名资源路由

默认情况下，所有的资源控制器行为都有一个路由名称。你可以传入 names 数组来覆盖这些名称：

Route::resource('photos', 'PhotoController')->names([

'create' => 'photos.build'

]);

(7)、命名资源路由参数

默认情况下，Route::resource 会根据资源名称的「单数」形式创建资源路由的路由参数。你可以在选项数组中传入 parameters 参数来轻松地覆盖每个资源。parameters 数组应该是资源名称和参数名称的关联数组。

Route::resource('users', 'AdminUserController')->parameters([

'users' => 'admin\_user'

]);

上例将会为资源的 show 路由生成如下的 URI ：/users/{admin\_user}。

(8)、本地资源化 URIs (本地化资源 URI)

默认情况下，Route::resource 将会用英文动词创建资源 URI。如果需要本地化 create 和 edit 行为动作名，可以在 AppServiceProvider 的 boot 中使用 Route::resourceVerbs 方法实现。

use Illuminate\Support\Facades\Route;

/\*\*

\* 引导任何应用服务。

\*

\* @return void

\*/

public function boot()

{

Route::resourceVerbs([

'create' => 'crear',

'edit' => 'editar',

]);

}

动作被自定义后，像 Route::resource('fotos', 'PhotoController') 这样注册的资源路由将会产生如下的 URI。

/fotos/crear

/fotos/{foto}/editar

(9)、补充资源控制器

如果你想在默认的资源路由中增加额外的路由，你应该在 Route::resource 之前定义这些路由。否则由 resource 方法定义的路由可能会无意中优先于你补充的路由.

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

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

Tip：记住保持控制器的专一性。如果你需要典型的资源操作之外的方法，可以考虑将你的控制器分成两个更小的控制器。

--

5、依赖注入 & 控制器

(1)、构造函数注入

Laravel 使用 服务容器( https://learnku.com/docs/laravel/6.x/container ) 来解析所有的控制器。因此，你可以在控制器的构造函数中使用类型提示需要的依赖项，而声明的依赖项会自动解析并注入控制器实例中。

<?php

namespace App\Http\Controllers;

use App\Repositories\UserRepository;

class UserController extends Controller

{

/\*\*

\* 用户 repository 实例.

\*/

protected $users;

/\*\*

\* 创建一个新的控制器实例。

\*

\* @param UserRepository $users

\* @return void

\*/

public function \_\_construct(UserRepository $users)

{

$this->users = $users;

}

}

?>

当然，你也可以类型提示 Laravel 契约( https://learnku.com/docs/laravel/6.x/contracts )，只要它能被解析。根据你的应用，将你的依赖项注入控制器能提供更好的可测试性。

(2)、方法注入

除了构造函数注入之外，你还可以在控制器方法中类型提示依赖项。最常见的用法就是将 Illuminate\Http\Request 实例注入到控制器方法中。

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller

{

/\*\*

\* 保存一个新用户。

\*

\* @param Request $request

\* @return Response

\*/

public function store(Request $request)

{

$name = $request->name;

//

}

}

?>

如果控制器方法需要从路由参数中获取输入内容，只需要在其他依赖项后列出路由参数即可。比如，如果你的路由是这样定义的。

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

你仍然可以类型提示 Illuminate\Http\Request 并通过定义控制器方法获取 id 参数，如下所示。

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller

{

/\*\*

\* 更新给定用户的信息。

\*

\* @param Request $request

\* @param string $id

\* @return Response

\*/

public function update(Request $request, $id)

{

//

}

}

?>

--

6、路由缓存

注意：基于闭包的路由不能被缓存。如果要使用路由缓存，你必须将所有的闭包路由转换成控制器类路由。

类路由。

如果你的应用只使用了基于控制器的路由，那么你应该充分利用 Laravel 的路由缓存。使用路由缓存将极大地减少注册所有应用路由所需的时间。某些情况下，路由注册的速度甚至可以快一百倍。要生成路由缓存，只需执行 Artisan 命令 route:cache。

//

//// 路由缓存

php artisan route:cache

运行这个命令之后，每一次请求的时候都将会加载缓存的路由文件。如果你添加了新的路由，你需要生成 一个新的路由缓存。因此，你应该只在生产环境运行 route:cache 命令。

//

//// 路由缓存清除

你可以使用 route:clear 命令清除路由缓存。

php artisan route:clear

--

--

五、请求 (HTTP 请求)

https://learnku.com/docs/laravel/6.x/requests/5139

--

1、接收请求

要通过依赖注入获取当前 HTTP 请求实例，你应该在控制器上引入 Illuminate\Http\Request 类，传入的请求实例将会由 服务容器( https://learnku.com/docs/laravel/6.x/container ) 自动注入。

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller

{

/\*\*

\* 存储一个新用户。

\*

\* @param Request $request

\* @return Response

\*/

public function store(Request $request)

{

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

//

}

}

?>

(1)、依赖注入 & 路由参数

///1/ 路由定义

如果你的控制器需要从路由参数中获取数据，你应该在其他依赖项之后列入参数。举个例子，你的路由是这样定义的。

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

//

///2/ 控制器操作方法定义

你可以通过下面的方法来定义控制器，使用 Illuminate\Http\Request 类来获取你的路由参数 id。

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller

{

/\*\*

\* Update the specified user.

\*

\* @param Request $request

\* @param string $id

\* @return Response

\*/

public function update(Request $request, $id)

{

//

}

}

?>

(2)、通过闭包路由获取参数

你也可以在路由闭包中使用 Illuminate\Http\Request 类， 服务容器会自动的将请求参数注入到路由闭包中。

use Illuminate\Http\Request;

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

//

});

(3)、请求路径 & 方法

Illuminate\Http\Request 提供了各种方法来检查应用程序的 http 请求，并扩展了 Symfony\Component\HttpFoundation\Request 类。 我们将在下面讨论一些最重要的方法。

///1/ 检索请求路径

// path()方法

path 方法返回请求的路径信息。因此，如果接收到的请求目标是 http://domain.com/foo/bar，则 path 方法会返回 foo/bar。

$uri = $request->path();

//

// is()方法

is 方法验证请求的路径是否与给定的模式匹配。使用此方法时，可以将 \* 字符作为通配符。

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

//

}

//

///2/ 检索请求 URL

要获取请求的完整 url ，可以使用 url 或 fullUrl 方法。 url 方法返回不带查询条件的 url 字符串，而 fullUrl 方法包含查询字符串。

// 没有查询字符串...

$url = $request->url();

//

// 带查询字符串...

$url = $request->fullUrl();

//

///3/ 检索请求方法

检索请求方法

method 方法将会返回请求的 HTTP 动词。你也可以使用 isMethod 方法验证 HTTP 动词是否与给定的字符串匹配。

$method = $request->method();

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

//

}

(4)、PSR-7 请求 Requests

PSR-7 标准( https://www.php-fig.org/psr/psr-7/ ) 指定了 HTTP 的消息接口，包含请求和响应。如果你想要获取 PSR-7 请求实例而不是 Laravel 请求， 那么你首先需要安装几个库。 Laravel 使用 Symfony HTTP Message Bridge 组件将典型的 Laravel 请求和响应转换为 PSR-7 的兼容实现。

composer require symfony/psr-http-message-bridge

composer require zendframework/zend-diactoros

//

安装这些库后，可以通过路由闭包和控制器方法的请求接口类型提示来获取 PSR-7 请求。

use Psr\Http\Message\ServerRequestInterface;

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

//

});

{提示} 如果从路由或控制器返回 PSR-7 响应实例，框架将会自动将其转换为 Laravel 的响应实例并显示。

--

2、输入预处理 & 规范化 (输入过滤 & 规范化)

默认情况下， Laravel 应用程序的全局中间件堆栈中包含了 TrimStrings 和 ConvertEmptyStringsToNull 中间件。它们被放在 App\Http\Kernel 类的堆栈列表中。 这些中间件将自动过滤掉请求中的字符串字段，并将空字符串字段转换为 null。这样一来，你将不用担心路由和控制器的约束规范问题。

如果你想禁用这些行为， 你可以在应用程序中 App\Http\Kernel 类的 $middleware 属性中移除这两个中间件。

--

3、接收数据 (Retrieving Input)

(1)、获取所有输入数据

// 你可以用 all 方法获取请求数据数组。

$input = $request->all();

(2)、获取一个值

使用一些简单方法，你可以从 Illuminate\Http\Request 获取所有的用户输入数据，不用担心用户使用的是哪种 HTTP 动作。不管是什么 HTTP 动词，都可以使用 input 方法获取用户输入的数据。

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

你可以给 input 方法传第二个参数。如果请求中没有呈递该参数则会返回此值。

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

//

当有呈递数组输入的数据时，可以使用「点」 运算符访问数据。

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

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

//

你可以使用不带参数的 input 方法，它将返回一包含所有用户输入数据的数组。

$input = $request->input();

(3)、从查询字符串获取输入

input 方法从整个请求体获取值（包含字符串）， query 仅从查询字符串中获取输入值。

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

//

如果请求中不存在查询的字符串，则第二个参数将作为默认值返回。

$name = $request->query('name', 'Helen');

//

你可以使用不带任何参数的 query 方法来获取所有查询条件的关联数组。

$query = $request->query();

(4)、通过动态属性获取输入

可以通过 Illuminate\Http\Request 实例的动态属性访问用户输入。例如，如果应用表单包含 name 域，可以像下面这样访问该域的值。

$name = $request->name;

在使用动态属性时，Laravel 首先会在请求载体中查找参数的值。如果该值不存在，Lavarel 将在路由参数中搜索。

(5)、当向应用传递 JSON 请求时，可以通过 input 方法访问 JSON 数据，只要将请求的 Content-Type 头设置为 application/json。 同样可以使用 「点」语法访问 JSON 数组。

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

(6)、获取部分输入数据

如果需要获取输入数据的子集，可以使用 only 或 except 方法。它们接受单个 array 或者动态参数列表。

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

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

$input = $request->except(['credit\_card']);

$input = $request->except('credit\_card');

Tip：only 方法返回请求中的全部键值对；但是它不返回请求中不存在的键值对。

(7)、判断输入值是否存在

has 方法用于判定请求中是否存在指定的值。如果请求中存在该值，has 方法返回 true。

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

//

}

//

如果给出一个数组， has 方法将判断在请求中，指定的值是否全部存在。

if ($request->has(['name', 'email'])) {

//

}

//

如果想要判断一个值在请求中是否存在，并且不为空，需要使用 filled 方法。

if ($request->filled('name')) {

//

}

(8)、旧数据

Laravel 允许你在两次请求之间保持数据。这个特性在有效性校验出错后重新填充表单时非常有用。不过，如果你使用 Laravel 自带 验证特性( https://learnku.com/docs/laravel/6.x/validation ), 不需要自己手动调用这些方法因为一些 Laravel 内置的验证功能会自动调用它们。

//

///1/ 将输入数据传送到 Session

Illuminate\Http\Request 类的 flash 方法将把当前的输入传送到 session , 在用户向应用发出这一次请求时它们仍然可用。

$request->flash();

//

你也可以使用 flashOnly 或 flashExcept 方法将请求数据的子集传送给 session。这些方法常用于将密码之类的敏感数据排除在 session 保持之外。

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

$request->flashExcept('password');

//

///2/ 传送数据并跳转

当你经常需要将输入传送至 session 并紧接着跳转至之前的页面，可以通过在跳转函数后链接调用 withInput 方法轻易地实现。

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

return redirect('form')->withInput(

$request->except('password')

);

//

///3/ 获取旧数据

若要获取上一次请求中闪存的输入，则可以使用 Request 实例的 old 方法。 old 方法会从 session( https://learnku.com/docs/laravel/6.x/session ) 取出之前被闪存的输入数据。

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

//

Laravel 也提供了全局辅助函数 old 。如果你要在 Blade 模板( https://learnku.com/docs/laravel/6.x/blade ) 中显示旧的输入，使用 old 会更加方便。如果给定字段没有旧的输入，则返回 null。

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

(9)、Cookies

///1/ 从请求中获取 Cookies

Laravel 框架生成的全部 cookies 都是加密的，并且已经用授权码签名，这意味着如果它们被客户端改变就会失效。使用 Illuminate\Http\Request 实例的 cookie 方法可以从请求中获取 cookie 值。

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

//

或者也可以使用 Cookie facade 访问 cookie 值。

use Illuminate\Support\Facades\Cookie;

$value = Cookie::get('name');

//

///2/ 将 Cookies 附加到响应中

你可以使用 cookie 将 cookie 附加到传出的 Illuminate\Http\Response 实例。你需要传递名称、值、cookie 的过期时间（以分钟为单位）给该方法。

return response('Hello World')->cookie(

'name', 'value', $minutes

);

//

cookie 方法还接受一些不太频繁使用的参数。通常这些参数和 PHP 内置的 setcookie ( https://secure.php.net/manual/en/function.setcookie.php ) 方法的参数有着相同的作用和意义。

return response('Hello World')->cookie(

'name', 'value', $minutes, $path, $domain, $secure, $httpOnly

);

//

或者，你可以使用 Cookie facade 来「排列」 用于从应用中附加到输出响应的 cookies。queue 方法接受一个 Cookie 实例或者用于创建 Cookie 所需的参数列表。在发送到浏览器之前，这些 cookie 将被附加到输出响应。

Cookie::queue(Cookie::make('name', 'value', $minutes));

Cookie::queue('name', 'value', $minutes);

//

///3/ 生成 Cookie 实例

如果你想要在一段时间以后生成一个可以给定 Symfony\Component\HttpFoundation\Cookie 的响应实例，可以使用全局辅助函数 cookie 。除非此 cookie 被附加到响应实例，否则不会发送回客户端。

$cookie = cookie('name', 'value', $minutes);

return response('Hello World')->cookie($cookie);

--

4、文件

(1)、接收上传的文件

你可以使用 file 方法或使用动态属性从 Illuminate\Http\Request 实例中访问上传的文件。 该 file 方法返回 Illuminate\Http\UploadedFile 类的实例，该类继承了 PHP 的 SplFileInfo 类的同时也提供了各种与文件交互的方法。

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

$file = $request->photo;

//

当然你也可以使用 hasFile 方法判断请求中是否存在指定文件。

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

//

}

//

///1/ 验证成功上传

除了检查上传的文件是否存在外，你也可以通过 isValid 方法验证上传的文件是否有效。

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

//

}

//

///2/ 文件路径 & 扩展名

UploadedFile 类还包含访问文件的全路径和扩展名的方法。 extension 方法会根据文件内容判断文件的扩展名。该扩展名可能会和客户端提供的扩展名不同。

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

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

//

///3/ 其它的文件方法

UploadedFile 实例上还有许多可用的方法。可以查看该类的 API 文档( https://api.symfony.com/3.0/Symfony/Component/HttpFoundation/File/UploadedFile.html ) 了解这些方法的详细信息。

(2)、存储上传的文件

要存储上传的文件，先配置好 文件系统( https://learnku.com/docs/laravel/6.x/filesystem )。 你可以使用 UploadedFile 的 store 方法把上传文件移动到你的某个磁盘上，该文件可能是本地文件系统中的一个位置，甚至像 Amazon S3 这样的云存储位置。

store 方法接受相对于文件系统配置的存储文件根目录的路径。这个路径不能包含文件名，因为系统会自动生成唯一的 ID 作为文件名。

store 方法还接受可选的第二个参数，用于存储文件的磁盘名称。这个方法会返回相对于磁盘根目录的文件路径。

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

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

//

如果你不想自动生成文件名，那么可以使用 storeAs 方法，它接受路径、文件名和磁盘名作为其参数。

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

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

--

5、配置信任代理

如果你的应用程序运行在失效的 TLS / SSL 证书的负载均衡器后，你可能会注意到你的应用程序有时不能生成 HTTPS 链接。通常这是因为你的应用程序正在从端口 80 上的负载均衡器转发流量，却不知道是否应该生成安全链接。

解决这个问题需要在 Laravel 应用程序中包含 App\Http\Middleware\TrustProxies 中间件，这使得你可以快速自定义应用程序信任的负载均衡器或代理。你的可信代理应该作为这个中间件的 $proxies 属性的数组列出。除了配置受信任的代理之外，还可以配置应该信任的代理 $header。

<?php

namespace App\Http\Middleware;

use Illuminate\Http\Request;

use Fideloper\Proxy\TrustProxies as Middleware;

class TrustProxies extends Middleware

{

/\*\*

\* 应用程序的可信代理列表

\*

\* @var array

\*/

protected $proxies = [

'192.168.1.1',

'192.168.1.2',

];

/\*\*

\* 应该用来检测代理的头信息

\*

\* @var string

\*/

protected $headers = Request::HEADER\_X\_FORWARDED\_ALL;

}

?>

Tip：如果你使用 AWS 弹性负载平衡，你的 $header 值应该是 Request::HEADER\_X\_FORWARDED\_AWS\_ELB。如果您想查看更多可用于 $headers 的属性信息，请查阅 Symfony 的 文档 信任代理( https://symfony.com/doc/current/deployment/proxies.html )。

(1)、信任所有代理

如果你使用 Amazon AWS 或其他的「云」的负载均衡服务，你可能不知道负载均衡器的实际 IP 地址。在这种情况下，你可以使用 \* 来信任所有代理。

/\*\*

\* 应用程序的可信代理列表

\*

\* @var array

\*/

protected $proxies = '\*';

--

--

六、响应 (HTTP 响应)

https://learnku.com/docs/laravel/6.x/responses/5140

--

1、创建响应

(1)、字符串 & 数组

所有路由和控制器处理完业务逻辑之后都会返回一个发送到用户浏览器的响应，Laravel 提供了多种不同的方式来返回响应，最基本的响应就是从路由或控制器返回一个简单的字符串，框架会自动将这个字符串转化为一个完整的 HTTP 响应。

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

return 'Hello World';

});

//

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

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

return [1, 2, 3];

});

Tip：你知道还可以从路由或控制器返回 Eloquent 集合(https://learnku.com/docs/laravel/6.x/eloquent-collections) 吗？他们也会被自动转化为 JSON 响应。

(2)、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( https://learnku.com/docs/laravel/6.x/views )。

返回完整的 Response 实例允许你自定义响应的 HTTP 状态码和响应头信息。 Response 实例 继承自 Symfony\Component\HttpFoundation\Response 类， 该类提供了各种构建 HTTP 响应的方法。

Route::get('home', function () {

return response('Hello World', 200)

->header('Content-Type', 'text/plain');

});

(3)、添加响应头

大部分的响应方法都是可链式调用的，使得创建响应实例的过程更具可读性。例如，你可以在响应返回给用户前使用 header 方法为其添加一系列的头信息。

return response($content)

->header('Content-Type', $type)

->header('X-Header-One', 'Header Value')

->header('X-Header-Two', 'Header Value');

//

或者，你可以使用 withHeaders 方法来指定要添加到响应的头信息数组。

return response($content)

->withHeaders([

'Content-Type' => $type,

'X-Header-One' => 'Header Value',

'X-Header-Two' => 'Header Value',

]);

///1/ 缓存控制中间件

Laravel 内置了一个 cache.headers 中间件，可以用来快速地为路由组设置 Cache-Control 头信息。如果在指令集中声明了 etag，Laravel 会自动将 ETag 标识符设置为响应内容的 MD5 哈希值。

Route::middleware('cache.headers:public;max\_age=2628000;etag')->group(function() {

Route::get('privacy', function () {

// ...

});

Route::get('terms', function () {

// ...

});

});

(4)、添加 Cookies 到响应

你可以使用响应上的 cookie 方法轻松地将为响应增加 Cookies。例如，你可以像这样使用 cookie 方法生成一个 cookie 并轻松地将其附加到响应上。

return response($content)

->header('Content-Type', $type)

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

//

cookie 方法还接受一些不太频繁使用的参数。通常，这些参数与原生 PHP 的 setcookie 方法的参数有着相同的目的和含义。

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

或者，你可以使用 Cookie facade 「队列」， Cookie 以附加到应用程序的传出响应。 queue 方法接受一个 Cookie 实例或创建 Cookie 实例所需的参数。 这些 cookie 在发送到浏览器之前会附加到传出响应中。

Cookie::queue(Cookie::make('name', 'value', $minutes));

Cookie::queue('name', 'value', $minutes);

(5)、Cookies & 加密

默认情况下，Laravel 生成的所有 Cookie 都是经过加密和签名，因此不能被客户端修改或读取。 如果你想要应用程序生成的部分 Cookie 不被加密，那么可以使用在 app/Http/Middleware 目录中 App\Http\Middleware\EncryptCookies 中间件的 $except 属性。

/\*\*

\* The names of the cookies that should not be encrypted.

\*

\* @var array

\*/

protected $except = [

'cookie\_name',

];

--

2、重定向

重定向响应是 Illuminate\Http\RedirectResponse 类的实例，并且包含用户需要重定向至另一个 URL 所需的头信息。Laravel 提供了几种方法用于生成 RedirectResponse 实例。其中最简单的方法是使用全局辅助函数 redirect。

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

return redirect('home/dashboard');

});

有时候你可能希望将用户重定向到之前的位置，比如提交的表单无效时。这时你可以使用全局辅助函数 back 来执行此操作。由于这个功能利用了 会话控制，请确保调用 back 函数的路由使用 web 中间件组或所有 Session 中间件。

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

// 验证请求

return back()->withInput();

});

(1)、重定向到命名路由

如果调用不带参数的辅助函数 redirect 时，会返回 Illuminate\Routing\Redirector 实例。这个实例允许你调用 Redirector 上的任何方法。例如为命名路由生成 RedirectResponse，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]);

//

///1/ 通过 Eloquent 模型填充参数

如果你要重定向到使用从 Eloquent 模型填充「ID」参数的路由，可以简单地传递模型本身。ID 会被自动提取。

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

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

//

如果你想要自定义这个路由参数中的默认参数名，需要重写模型实例上的 getRouteKey 方法。

/\*\*

\* 获取模型的路由键

\*

\* @return mixed

\*/

public function getRouteKey()

{

return $this->slug;

}

(2)、重定向到控制器行为

还可以生成到 controller action（ https://learnku.com/docs/laravel/6.x/controllers ） 的重定向。要达到这个目的，只要把 控制器 和 action 的名称传递给 action 方法。记住，不需要传递控制器的全部命名空间，Laravel 的 RouteServiceProvider 会自动将其设置为基本控制器的命名空间。

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

//

如果控制器路由需要参数，可以将其作为 action 方法的第二个参数。

return redirect()->action(

'UserController@profile', ['id' => 1]

);

(3)、重定向到外部域名

有时候你需要重定向到应用外的域名。调用 away 方法可以达到此目的，它会创建一个不带有任何额外的 URL 编码、有效性校验和检查的 RedirectResponse 实例。

return redirect()->away('https://www.google.com');

(4)、重定向并使用闪存的 Session 数据

重定向到新的 URL 的同时 传送数据给 session( https://learnku.com/docs/laravel/6.x/session#flash-data ) 是很常见的。 通常会在成功执行一个动作并传送消息给 session 之后这样做。为了方便起见，你可以创建一个 RedirectResponse 实例并在链式方法调用中将数据传送给 session。

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

// Update the user's profile...

return redirect('dashboard')->with('status', 'Profile updated!');

});

//

在用户重定向后，可以显示 session( https://learnku.com/docs/laravel/6.x/session ) 中的传送数据。比如使用 Blade syntax ( https://learnku.com/docs/laravel/6.x/blade )。

@if (session('status'))

<div class="alert alert-success">

{{ session('status') }}

</div>

@endif

--

3、其他响应类型 (其它的响应类型)

response 助手可以用于生成其它类型的响应实例。当还带参数调用 response 助手时，返回 Illuminate\Contracts\Routing\ResponseFactory contract ( https://learnku.com/docs/laravel/6.x/contracts ) 的一个实现。这个契约提供了几个用于生成响应的方法。

(1)、视图响应

如果需要把 视图(https://learnku.com/docs/laravel/6.x/views) 作为响应内容返回的同时，控制响应状态和头信息，就需要调用 view 方法。

return response()

->view('hello', $data, 200)

->header('Content-Type', $type);

如果不需要传递自定义的 HTTP 状态码和自定义头信息，还可以使用全局的 view 辅助函数。

(2)、JSON 响应

json 自动将 Content-Type 头信息设置为 application/json，同时使用 PHP 的 json\_encode 函数将给定的数组转换为 JSON 。

return response()->json([

'name' => 'Abigail',

'state' => 'CA'

]);

//

如果想要创建 JSONP 响应，可以结合 withCallback 方法使用 json 方法。

return response()

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

->withCallback($request->input('callback'));

(3)、文件下载

download 方法可以用于生成强制用户浏览器下载给定路径文件的响应。 download 方法文件名作为其第二个参数，它将作为用户下载文件的文件名。最后，你可以传递 HTTP 头信息数组作为其第三个参数。

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

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

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

注意：用于管理文件下载的 Symfony HttpFoundation 要求下载的文件有一个 ASCII 文件名。

//

///1/ 流下载

有时，你可能希望将给定操作的字符串响应转换为下载响应，而不需要将其写入磁盘。此时可以使用 streamDownload 方法。这个方法接受回调、文件名和可选的头信息数组作为参数。

return response()->streamDownload(function () {

echo GitHub::api('repo')

->contents()

->readme('laravel', 'laravel')['contents'];

}, 'laravel-readme.md');

(4)、文件响应

file 方法用于直接在用户浏览器显示一个图片或 PDF 之类的文件，而不是下载。这个方法接受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 方法编写如下代码。

<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

use Illuminate\Support\Facades\Response;

class ResponseMacroServiceProvider extends ServiceProvider

{

/\*\*

\* 注册应用程序的响应宏

\*

\* @return void

\*/

public function boot()

{

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

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

});

}

}

?>

macro 方法接受一个名称作为第一个参数，闭包函数作为的第二个参数。响应宏的闭包在 ResponseFactory 实现类或辅助函数 response 中调用宏名称的时候被执行。

//

//// 响应宏调用

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

--

--

七、视图

https://learnku.com/docs/laravel/6.x/views/5141

--

1、创建视图

Tip：如果你想找到有关如何编写 Blade 模板的更多信息？从 查看完整的 Blade 文档( https://learnku.com/docs/laravel/6.x/blade ) 入手。

视图包含应用程序的 HTML，并且将控制器 / 应用程序逻辑与演示逻辑分开。视图文件存放于 resources/views 目录下。一个简单的视图如下所示。

<!-- 此视图文件位置 resources/views/greeting.blade.php -->

<html>

<body>

<h1>Hello, {{ $name }}</h1>

</body>

</html>

//

该视图文件位于 resources/views/greeting.blade.php， 可以使用全局辅助函数 view 将其返回。

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

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

});

如你所见， 传递给 view 辅助函数的第一个参数对应 resources/views 目录中视图文件的名称。第二个参数是应该可供视图使用的数据数组。在这种情况下，我们传递 name 变量，该变量将使用 Blade syntax( https://learnku.com/docs/laravel/laravel/6.x/blade ) 在视图中显示。

当然，视图文件也可以嵌套在 resources/views 目录的子目录中。「点」符号可以用来引用嵌套视图。例如，如果你的视图存储在 resources/views/admin/profile.blade.php，则可以这样引用它。

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

(1)、判断视图文件是否存在

如果需要判断视图文件是否存在，可以使用 View facade。如果存在，exists 方法会返回 true。

use Illuminate\Support\Facades\View;

if (View::exists('emails.customer')) {

//

}

(2)、创建第一个可用视图

使用 first 方法，你可以创建存在于给定数组视图中的第一个视图。 如果你的应用程序或开发的第三方包允许定制或覆盖视图，这非常有用。

return view()->first(['custom.admin', 'admin'], $data);

//

当然，也可以通过 View facade 调用这个方法。

use Illuminate\Support\Facades\View;

return View::first(['custom.admin', 'admin'], $data);

--

2、向视图传递数据 (向视图传递参数)

正如您在前面的示例中所看到的，您可以将一组数据传递给视图。

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

以这种方式传递信息时，数据应该是具有键 / 值对的数组。在视图中，您可以使用相应的键访问每个值，例如 <?php echo $key; ?>。 作为将完整的数据数组传递给 view 辅助函数的替代方法，您可以使用 with 方法将各个数据片段添加到视图中。

return view('greeting')->with('name', 'Victoria');

(1)、与视图共享数据 (与所有视图共享数据)

有时候，你可能需要共享一段数据给应用程序的所有视图。 你可以在服务提供器的 boot 方法中调用视图 Facade 的 share 方法。例如，可以将它们添加到 AppServiceProvider 或者为它们生成一个单独的服务提供器。

<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;

class AppServiceProvider extends ServiceProvider

{

/\*\*

\* 注册任何应用服务

\*

\* @return void

\*/

public function register()

{

//

}

/\*\*

\* 引导任何应用程序服务

\*

\* @return void

\*/

public function boot()

{

View::share('key', 'value');

}

}

?>

--

3、视图合成器

视图合成器是在呈现视图时调用的回调或类方法。 如果每次呈现视图时都希望将数据绑定到视图，则视图合成器可以帮助您将该逻辑组织到一个位置。

在下面这个例子中，我们会在一个 服务提供商( https://learnku.com/docs/laravel/6.x/providers ) 中注册视图合成器。 使用 View facade 来访问底层的 Illuminate\Contracts\View\Factory 契约实现。默认情况下，Laravel 没有存放视图合成器的目录，你需要根据需求来重新建立目录，例如： app/Http/View/Composers。

<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;

use Illuminate\Support\ServiceProvider;

class ViewServiceProvider extends ServiceProvider

{

/\*\*

\* 注册任何应用服务

\*

\* @return void

\*/

public function register()

{

//

}

/\*\*

\* 引导任何应用程序服务

\*

\* @return void

\*/

public function boot()

{

// 使用基于合成器的类...

View::composer(

'profile', 'App\Http\View\Composers\ProfileComposer'

);

// 使用基于合成器的闭包...

View::composer('dashboard', function ($view) {

//

});

}

}

?>

注意：记住，如果你创建了新的一个服务提供器来存放你注册视图合成器的代码，那么你需要将这个服务提供器添加到配置文件 config/app.php 的 providers 数组中。

到此我们已经注册了视图合成器，每次渲染 profile 视图时都会执行 ProfileComposer@compose 方法。那么下面我们来定义视图合成器的这个类吧。

<?php

namespace App\Http\View\Composers;

use Illuminate\View\View;

use App\Repositories\UserRepository;

class ProfileComposer

{

/\*\*

\* 实现 UserRepository

\*

\* @var UserRepository

\*/

protected $users;

/\*\*

\* 创建一个新的 profile 合成器.

\*

\* @param UserRepository $users

\* @return void

\*/

public function \_\_construct(UserRepository $users)

{

// Dependencies automatically resolved by service container...

$this->users = $users;

}

/\*\*

\* 将数据绑定到视图

\*

\* @param View $view

\* @return void

\*/

public function compose(View $view)

{

$view->with('count', $this->users->count());

}

}

?>

视图合成器的 compose 方法会在视图渲染之前被调用，并传入一个 Illuminate\View\View 实例。你可以使用 with 方法将数据绑定到视图。

提示： 所有的视图合成器都会通过 服务容器( https://learnku.com/docs/laravel/6.x/container ), 进行解析，所以你可以在视图合成器的构造函数中类型提示需要注入的依赖项。

(1)、将视图合成器添加到多个视图

通过将一组视图作为第一个参数传入 composer 方法，将一个视图合成器添加到多个视图。

View::composer(

['profile', 'dashboard'],

'App\Http\View\Composers\MyViewComposer'

);

//

composer 方法同时也接受通配符 \* ，表示将一个视图合成器添加到所有视图。

View::composer('\*', function ($view) {

//

});

(2)、视图构造器

视图 creators 和视图合成器非常相似。唯一不同之处在于视图构造器在视图实例化之后立即执行，而视图合成器在视图即将渲染时执行。使用 creator 方法注册视图构造器。

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

--

--

八、生成 URL

https://learnku.com/docs/laravel/6.x/urls/5142

--

1、简介

Laravel 提供了几个辅助函数来为应用程序生成 URL。主要用于在模板和 API 响应中构建 URL 或者在应用程序的其它部分生成重定向响应。

--

2、基础

(1)、生成基础 URL

辅助函数 url 可以用于应用的任何一个 URL。生成的 URL 将自动使用当前请求中的方案 (HTTP or HTTPS) 和主机。

$post = App\Post::find(1);

echo url("/posts/{$post->id}"); // http://example.com/posts/1

(2)、访问当前 URL

如果没有给辅助函数 url 提供路径，则会返回一个 Illuminate\Routing\UrlGenerator 实例，来允许你访问有关当前 URL 的信息。

// Get the current URL without the query string...

echo url()->current();

//

// Get the current URL including the query string...

echo url()->full();

//

// Get the full URL for the previous request...

echo url()->previous();

//

上面的这些方法都可以通过 URL facade 访问。

use Illuminate\Support\Facades\URL;

echo URL::current();

--

3、命名路由的 URL

辅助函数 route 可以用于为指定路由生成 URL。命名路由生成的 URL 不与路由上定义的 URL 相耦合。因此，就算路由的 URL 有任何改变，都不需要对 route 函数调用进行任何更改。例如，假设你的应用程序包含以下路由。

Route::get('/post/{post}', function () {

//

})->name('post.show');

//

要生成此路由的 URL ，可以像这样使用辅助函数 route。

echo route('post.show', ['post' => 1]); // http://example.com/post/1

//

您通常会使用 Eloquent 模型 的主键生成 URL。因此，您可以将 Eloquent 模型作为参数值传递。route 辅助函数将自动提取模型的主键。

echo route('post.show', ['post' => $post]);

//

辅助函数 route 也可用于为具有多个参数的路由生成 URL。

Route::get('/post/{post}/comment/{comment}', function () {

//

})->name('comment.show');

echo route('comment.show', ['post' => 1, 'comment' => 3]); // http://example.com/post/1/comment/3

(1)、签名 URL

Laravel 允许你轻松地为命名路径创建「签名」URL，这些 URL 在查询字符串后附加了「签名」哈希，允许 Laravel 验证 URL 自创建以来未被修改过。签名 URL 对于可公开访问但需要一层防止 URL 操作的路由特别有用。

例如，你可以使用签名 URL 来实现通过电子邮件发送给客户的公共「取消订阅」链接。要创建指向路径的签名 URL ，请使用 facade 的 signedRoute 方法 URL。

use Illuminate\Support\Facades\URL;

return URL::signedRoute('unsubscribe', ['user' => 1]);

//

如果要生成过期的临时签名路由 URL ，可以使用以下 temporarySignedRoute 方法。

use Illuminate\Support\Facades\URL;

return URL::temporarySignedRoute(

'unsubscribe', now()->addMinutes(30), ['user' => 1]

);

//

///1/ 验证签名路由请求

要验证传入请求是否具有有效签名，你应该调用 hasValidSignature 传入的方法 Request。

use Illuminate\Http\Request;

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

if (! $request->hasValidSignature()) {

abort(401);

}

// ...

})->name('unsubscribe');

或者，你可以将 Illuminate\Routing\Middleware\ValidateSignature 中间件分配给路由。如果它不存在，则应该在 HTTP 内核的 routeMiddleware 数组中为此中间件分配一个键。

/\*\*

\* 应用程序的路由中间件

\*

\* 这些中间件可能被分配给组或单独使用

\*

\* @var array

\*/

protected $routeMiddleware = [

'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,

];

在内核中注册中间件后，你可以将其附加到路由中。如果传入请求没有有效签名，则中间件将自动返回 403 错误响应。

Route::post('/unsubscribe/{user}', function (Request $request) {

// ...

})->name('unsubscribe')->middleware('signed');

--

4、控制器行为的 URL

action 功能可以为给定的控制器行为生成 URL 。这个功能不需要你传递控制器的完整命名空间，但你需要传递相对于命名空间 App\Http\Controllers 的控制器类名。

$url = action('HomeController@index');

//

你还可以使用 “可调用” 数组语法引用操作。

use App\Http\Controllers\HomeController;

$url = action([HomeController::class, 'index']);

//

如果控制器方法需要路由参数，那就将它们作为第二个参数传递给 action 函数。

$url = action('UserController@profile', ['id' => 1]);

--

5、默认值

对于某些应用程序，你可能希望为某些 URL 参数的请求范围指定默认值。例如，假设有些路由定义了 {locale} 参数。

Route::get('/{locale}/posts', function () {

//

})->name('post.index');

每次都通过 locale 来调用辅助函数 route 也是一件很麻烦的事情。 因此，使用 URL::defaults 方法定义这个参数的默认值，可以让该参数始终存在当前请求中。然后就能从 路由中间件( https://learnku.com/docs/laravel/6.x/middleware#assigning-middleware-to-routes ) 调用此方法来访问当前请求。

<?php

namespace App\Http\Middleware;

use Closure;

use Illuminate\Support\Facades\URL;

class SetDefaultLocaleForUrls

{

public function handle($request, Closure $next)

{

URL::defaults(['locale' => $request->user()->locale]);

return $next($request);

}

}

?>

一旦设置了 locale 参数的默认值，您就不再需要通过辅助函数 route 生成 URL 时传递它的值。

--

--

九、Session (HTTP 会话机制)

https://learnku.com/docs/laravel/6.x/session/5143

--

1、简介

由于 HTTP 驱动的应用程序是无状态的，Session 提供了一种在多个请求之间存储有关用户信息的方法，Laravel 通过同一个可读性强的 API 处理各种自带的后台驱动程序。支持诸如比较热门的 Memcached( https://memcached.org/ ), Redis( https://redis.io/ ) 和数据库。

(1)、配置

Session 的配置文件存储在 config/session.php 文件中。请务必查看此文件中对于你而言可用的选项。默认情况下，Laravel 为绝大多数应用程序配置的 Session 驱动为 file 。在生产环境中，你可以考虑使用 memcached 或 redis 驱动，让 Session 的性能更加出色。

Session driver 的配置预设了每个请求存储 Session 数据的位置。Laravel 自带了几个不错而且开箱即用的驱动。

//

///1/ 驱动类型

file - 将 Session 存储在 storage/framework/sessions 中。

cookie - Sessions 被存储在安全加密的 cookie 中。

database - Sessions 被存储在关系型数据库中。

memcached / redis - Sessions 被存储在基于高速缓存的存储系统中。

array - Sessions 存储在 PHP 数组中，但不会被持久化。

Tip：数组驱动一般用于 测试( https://learnku.com/docs/laravel/6.x/testing ) 并且防止存储在 Session 中的数据被持久化。

(2)、驱动程序先决条件

//

///1/ 数据库

使用 database 作为 Session 驱动时，你需要创建一张包含 Session 各项数据的表。以下是使用 Schema 建表的例子。

Schema::create('sessions', function ($table) {

$table->string('id')->unique();

$table->unsignedInteger('user\_id')->nullable();

$table->string('ip\_address', 45)->nullable();

$table->text('user\_agent')->nullable();

$table->text('payload');

$table->integer('last\_activity');

});

你可以使用 Artisan 命令 session:table 来生成此迁移。

php artisan session:table

php artisan migrate

//

///2/ Redis

Laravel 在使用 Redis 作为 Session 驱动之前，需要通过 Composer 安装 predis/predis 扩展包 (~1.0)。然后在 database 配置文件中配置 Redis 连接信息。在 session 配置文件中，connection 选项可用于指定 Session 使用哪个 Redis 连接。

--

2、使用 Session

(1)、获取数据

Laravel 中处理 Session 数据有两种主要方法：全局辅助函数 session 和通过一个 Request 实例。

//

///1/ 通过Request实例获取

首先，我们来看看通过控制器方法类型提示一个 Request 实例来访问 session。控制器方法依赖项会通过 Laravel 服务容器( https://learnku.com/docs/laravel/6.x/container ) 实现自动注入。

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Controllers\Controller;

class UserController extends Controller

{

/\*\*

\* 展示给定用户的配置文件。

\*

\* @param Request $request

\* @param int $id

\* @return Response

\*/

public function show(Request $request, $id)

{

$value = $request->session()->get('key');

//

}

}

?>

当你从 Session 获取值时，你还可以传递一个默认值作为 get 方法的第二个参数。如果 Session 中不存在指定的键，便会返回这个默认值。若传递一个闭包作为 get 方法的默认值，并且所请求的键并不存在时，get 方法将执行闭包并返回其结果。

$value = $request->session()->get('key', 'default');

$value = $request->session()->get('key', function () {

return 'default';

});

//

///2/ 全局辅助函数 Session

你也可以使用全局的 PHP 辅助函数 session 来获取和存储 Session 数据。 使用单个字符串类型的值作为参数调用辅助函数 session 时，它会返回该字该符串对应的 Session 键的值。当使用一个键值对数组作为参数调用辅助函数 session 时，传入的键值将会存储在 Session 中。

Route::get('home', function () {

// 获取 session 中的一条数据...

$value = session('key');

// 指定一个默认值...

$value = session('key', 'default');

// 在 Session 中存储一条数据...

session(['key' => 'value']);

});

Tip：通过 HTTP 请求实例操作 Session 与使用全局辅助函数 session 两者之间并没有实质上的区别。这两种方法都可以通过所有测试用例中可用的 assertSessionHas 方法进行 测试( https://learnku.com/docs/laravel/6.x/testing )。

//

///3/ 获取所有的 Session 数据

如果你想要获取所有的 Session 数据，可以使用 all 方法。

$data = $request->session()->all();

//

///4/ 判断 Session 中是否存在某个值

/// has()

要确定 Session 中是否存在某个值，可以使用 has 方法。如果该值存在且不为 null，那么 has 方法会返回 true。

if ($request->session()->has('users')) {

//

}

/// exists()

要确定 Session 中是否存在某个值，即使其值为 null，也可以使用 exists 方法。如果值存在，则 exists 方法返回 true。

if ($request->session()->exists('users')) {

//

}

(2)、存储数据

想要存储数据到 Session，你可以使用 put 方法，或者使用辅助函数 session。

// 通过请求实例

$request->session()->put('key', 'value');

// 通过全局辅助函数

session(['key' => 'value']);

//

///1/ 在 Session 数组中保存数据

push 方法可以将一个新的值添加到 Session 数组内。例如，假设 user.teams 这个键是包括团队名称的数组，你可以这样将一个新的值加入到数组中。

$request->session()->push('user.teams', 'developers');

//

///2/ 检索 & 删除一条数据

pull 方法可以只使用一条语句就从 Session 中检索并删除一条语句。

$value = $request->session()->pull('key', 'default');

(3)、闪存数据

有时候你可能想在 Session 中保存数据用于下一次请求，这时你可以使用 flash 方法。使用这个方法保存在 Session 中的数据，只会保留到下一个 HTTP 请求到来之前，然后就会被删除。闪存数据主要用于短期的状态消息。

$request->session()->flash('status', 'Task was successful!');

如果你需要在更多的请求中使用到该一次性数据，你可以使用 reflash 方法，该方法会将所有一次性请求保留到下一次请求。如果你想保存一次性数据，你可以用 keep 方法。

$request->session()->reflash();

$request->session()->keep(['username', 'email']);

(4)、删除数据

forget 方法会从 Session 中删除指定数据，如果想从 Session 中删除所有数据，可以使用 flush 方法。

// 删除单个值...

$request->session()->forget('key');

// 删除多个值...

$request->session()->forget(['key1', 'key2']);

$request->session()->flush();

(5)、重新生成 Session ID (重新生成 Session ID)

重新生成 session ID 通常是为了防止恶意用户利用 session fixation( https://en.wikipedia.org/wiki/Session\_fixation ) 对你的应用进行攻击。

如果你使用了内置函数 LoginController，Laravel 会自动重新生成身份认证中的 Session ID。否则，你需要手动使用 regenerate 方法重新生成 Session ID。

$request->session()->regenerate();

--

3、添加自定义 Session 驱动

(1)、实现驱动

//// MongoDB驱动

你自定义的 Session 驱动必须实现 SessionHandlerInterface 接口。这个接口包含了一些我们需要实现的简单方法。下面是 MongoDB 实现的大概流程示例。

<?php

namespace App\Extensions;

class MongoSessionHandler implements \SessionHandlerInterface

{

public function open($savePath, $sessionName) {}

public function close() {}

public function read($sessionId) {}

public function write($sessionId, $data) {}

public function destroy($sessionId) {}

public function gc($lifetime) {}

}

?>

Tip：Laravel 默认没有附带一个用于包扩展的目录，你可以把它放在你喜欢的目录内。在上面这个例子中，我们创建了一个 Extensions 目录用于存放 MongoSessionHandler。

//

///1/ 驱动方法说明

// 由于以上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 方法通常用于基于文件的 Session 存储系统。因为 Laravel 已经附带了一个 file Session 驱动。所以你不需要在该方法中放置任何代码。PHP 要求必须要有这个方法的实现（这只是一个糟糕的接口设计），你只需要把这个方法置空。

close 方法跟 open 方法相似，通常也可以被忽略。对大多数的驱动而言，此方法不是必须的。

read 方法应当返回与给定的 $sessionId 相匹配的 Session 数据的字符串格式。在你的自定义的驱动中获取或存储 Session 数据时，不需要进行任何序列化或者其他编码，因为 Laravel 会自动为你执行序列化。

write 方法将与 $sessionId 关联的给定的 $data 字符串写入到一些持久化存储系统，如 MongoDB、Dynamo 等。再次重申，你不需要进行任何序列化或其他编码，因为 Laravel 会自动为你处理这些事情。

destroy 方法将会从持久化存储中删除与 $sessionId 相关的数据。

gc 方法能销毁给定的 $lifetime（UNIX 的时间戳）之前的所有数据。对本身拥有过期机制的系统如 Memcached 和 Redis 而言，该方法可以置空。

(2)、注册驱动

当实现驱动后，需要在框架中注册它。在 Laravel 后端添加额外的驱动，需要使用 Session facade( https://learnku.com/docs/laravel/6.x/facades ) 的 extend 方法。你应该在 服务提供者( https://learnku.com/docs/laravel/6.x/providers )中的 boot 方法中调用 extend 方法。你可以在已有的 AppServiceProvider 或者另外创建一个服务提供者执行此操作。

<?php

namespace App\Providers;

use App\Extensions\MongoSessionHandler;

use Illuminate\Support\Facades\Session;

use Illuminate\Support\ServiceProvider;

class SessionServiceProvider extends ServiceProvider

{

/\*\*

\* 在容器中注册绑定关系

\*

\* @return void

\*/

public function register()

{

//

}

/\*\*

\* 执行服务的注册后启动

\*

\* @return void

\*/

public function boot()

{

Session::extend('mongo', function ($app) {

// 返回实现 SessionHandlerInterface 的对象…

return new MongoSessionHandler;

});

}

}

?>

驱动完成注册时，你可以在使用在配置文件 config/session.php 中使用 mongo 驱动。

--

--

十、 表单验证

https://learnku.com/docs/laravel/6.x/validation/5144

--

1、简介

Laravel 提供了几种不同的方法来验证传入应用程序的数据。默认情况下，Laravel 的控制器基类使用 ValidatesRequests trait，它提供了一种方便的方法去使用各种强大的验证规则来验证传入的 HTTP 请求。

--

2、快速验证

要了解 Laravel 强大的验证功能，让我们看一个验证表单并将错误消息显示回给用户的完整示例。

(1)、定义路由

首先，让我们假设在 routes/web.php 文件中定义了下面这些路由。

Route::get('post/create', 'PostController@create');

Route::post('post', 'PostController@store');

GET 路由会显示一个供用户创建一个新的博客帖子的表单，而 POST 路由会将新的博客文章存储在数据库中。

(2)、创建控制器

下面让我们一起来看看处理这些路由的控制器，store 方法暂时留空。

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Controllers\Controller;

class PostController extends Controller

{

/\*\*

\* 显示创建博客文章的表单

\*

\* @return Response

\*/

public function create()

{

return view('post.create');

}

/\*\*

\* 保存一篇新的博客文章

\*

\* @param Request $request

\* @return Response

\*/

public function store(Request $request)

{

// 验证并存储博客文章...

}

}

?>

(3)、编写验证逻辑 (编写验证器逻辑)

现在我们开始在 store 方法中编写逻辑来验证新的博客文章。为此，我们将使用 Illuminate\Http\Request 对象提供的 validate 方法 。如果验证通过，代码就可以正常的运行。如果验证失败，则会抛出异常，并自动将对应的错误响应返回给用户。在典型的 HTTP 请求的情况下，会生成一个重定向响应，而对于 AJAX 请求则会发送 JSON 响应。

让我们接着回到 store 方法来深入理解 validate 方法。

/\*\*

\* 保存一篇新的博客文章

\*

\* @param Request $request

\* @return Response

\*/

public function store(Request $request)

{

$validatedData = $request->validate([

'title' => 'required|unique:posts|max:255',

'body' => 'required',

]);

// 博客文章验证通过...

}

如你所见，我们将所需的验证规则传递至 validate 方法中。另外再提醒一次，如果验证失败，会自动生成一个对应的响应。如果验证通过，那我们的控制器将会继续正常运行。

//

或者，验证规则可以指定为一个数组而不是单个 | 分隔字符串。

$validatedData = $request->validate([

'title' => ['required', 'unique:posts', 'max:255'],

'body' => ['required'],

]);

//

///1/ 首次验证失败后停止运行

如果你希望在某个属性第一次验证失败后停止运行验证规则，你需要附加 bail 规则到该属性。

$request->validate([

'title' => 'bail|required|unique:posts|max:255',

'body' => 'required',

]);

在这个例子中，如果 title 字段没有通过 unique 规则，那么程序就不会继续检查 max 规则。规则会按照分配的顺序来验证。

//

///2/ 关于数组数据的注意实现

如果你的 HTTP 请求包含一个 「嵌套」 参数（即数组），那你可以在验证规则中通过 「点」 语法来指定这些参数。

$request->validate([

'title' => 'required|unique:posts|max:255',

'author.name' => 'required',

'author.description' => 'required',

]);

(4)、显示验证错误信息

如果传入的请求参数未通过给定的验证规则呢？正如前面所提到的，Laravel 会自动把用户重定向到之前的位置。另外，所有的验证错误信息会被自动 存储到 session( https://learnku.com/docs/laravel/6.x/session#flash-data )。

重申一次，我们不必在 GET 路由中将错误消息显式绑定到视图。因为 Lavarel 会检查在 Session 数据中的错误信息，并自动将其绑定到视图（如果这个视图文件存在）。而其中的变量 $errors 是 Illuminate\Support\MessageBag 的一个实例。要获取关于这个对象的更多信息，请 查阅这个文档( https://learnku.com/docs/laravel/6.x/validation/5144#working-with-error-messages )。

提示：$errors 变量被 web 中间件组提供的 Illuminate\View\Middleware\ShareErrorsFromSession 中间件绑定到视图中。 当这个中间件被应用后，在你的视图中就可以获取到 $error 变量 , 可以使一直假定 $errors 变量存在并且可以安全地使用。

//

//// 视图显示错误信息

因此，在我们的例子中，当验证失败的时候，用户将会被重定向到控制器的 create 方法，使我们能在视图中显示错误信息。

<!-- /resources/views/post/create.blade.php -->

<h1>Create Post</h1>

@if ($errors->any())

<div class="alert alert-danger">

<ul>

@foreach ($errors->all() as $error)

<li>{{ $error }}</li>

@endforeach

</ul>

</div>

@endif

<!-- Create Post Form -->

//

///1/ @error 指令

你还可以使用 @error Blade 模板( https://learnku.com/docs/laravel/6.x/blade ) 指令快速检查给定属性是否存在验证错误消息。 在 @error 指令中，你可以输出 $message 变量以显示错误消息。

<!-- /resources/views/post/create.blade.php -->

<label for="title">Post Title</label>

<input id="title" type="text" class="@error('title') is-invalid @enderror">

@error('title')

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

@enderror

(5)、可选字段注意事项

默认情况下，在 Laravel 应用的全局中间件堆栈 App\Http\Kernel 类中包含了 TrimStrings 和 ConvertEmptyStringsToNull 中间件。因此，如果你不希望验证程序将 null 值视为无效的话，那就需要将「可选」的请求字段标记为 nullable，举个例子。

$request->validate([

'title' => 'required|unique:posts|max:255',

'body' => 'required',

'publish\_at' => 'nullable|date',

]);

在这个例子里，我们指定 publish\_at 字段可以为 null 或者一个有效的日期格式。如果 nullable 的修饰词没有被添加到规则定义中，验证器会认为 null 是一个无效的日期格式。

//

///1/ AJAX 请求 & 验证

在这个例子中，我们使用传统的表单将数据发送到应用程序。但实际情况中，很多程序都会使用 AJAX 来发送请求。当我们对 AJAX 的请求中使用 validate 方法时，Laravel 并不会生成一个重定向响应，而是会生成一个包含所有验证错误信息的 JSON 响应。这个 JSON 响应会包含一个 HTTP 状态码 422 被发送出去。

--

3、表单请求验证 (验证表单请求)

(1)、创建表单请求验证

面对更复杂的验证情境中，你可以创建一个「表单请求」来处理更为复杂的逻辑。表单请求是包含验证逻辑的自定义请求类。可使用 Artisan 命令 make:request 来创建表单请求类。

//

///1/ 表单请求类

php artisan make:request StoreBlogPost

//

///2/ 添加验证规则

新生成的类保存在 app/Http/Requests 目录下。如果这个目录不存在，运行 make:request 命令时它会被创建出来。让我们添加一些验证规则到 rules 方法中。

/\*\*

\* 获取适用于请求的验证规则。

\*

\* @return array

\*/

public function rules()

{

return [

'title' => 'required|unique:posts|max:255',

'body' => 'required',

];

}

Tip：你可以向 rules 方法传入所需的任何依赖项。他们会自动被 Laravel 提供的 服务容器( https://learnku.com/docs/laravel/6.x/container ) 自动解析。

//

///3/ 表单请求类使用

所以，验证规则是如何运行的呢？你所需要做的就是在控制器方法中类型提示传入的请求。在调用控制器方法之前验证传入的表单请求，这意味着你不需要在控制器中写任何验证逻辑。

/\*\*

\* 存储传入的博客文章。

\*

\* @param StoreBlogPost $request

\* @return Response

\*/

public function store(StoreBlogPost $request)

{

// 传入的请求通过验证...

// 获取通过验证的数据...

$validated = $request->validated();

}

如果验证失败，就会生成一个让用户返回到先前的位置的重定向响应。这些错误也会被闪存到 session 中，以便这些错误都可以在页面中显示出来。如果传入的请求是 AJAX，会向用户返回具有 422 状态代码和验证错误信息的 JSON 数据的 HTTP 响应。

//

///4/ 添加表单请求后钩子

如果你想在表单请求「之后」添加钩子，可以使用 withValidator 方法。这个方法接收一个完整的验证构造器，允许你在验证结果返回之前调用任何方法。

/\*\*

\* 配置验证器实例。

\*

\* @param \Illuminate\Validation\Validator $validator

\* @return void

\*/

public function withValidator($validator)

{

$validator->after(function ($validator) {

if ($this->somethingElseIsInvalid()) {

$validator->errors()->add('field', 'Something is wrong with this field!');

}

});

}

(2)、表单请求授权验证

表单请求类内也包含了 authorize 方法。在这个方法中，你可以检查经过身份验证的用户确定其是否具有更新给定资源的权限。比方说，你可以判断用户是否拥有更新文章评论的权限。

/\*\*

\* 判断用户是否有权限做出此请求。

\*

\* @return bool

\*/

public function authorize()

{

$comment = Comment::find($this->route('comment'));

return $comment && $this->user()->can('update', $comment);

}

由于所有的表单请求都是继承了 Laravel 中的请求基类，所以我们可以使用 user 方法去获取当前认证登录的用户。同时请注意上述例子中对 route 方法的调用。这个方法允许你在被调用的路由上获取其定义的 URI 参数，譬如下面例子中的 {comment} 参数。

Route::post('comment/{comment}');

如果 authorize 方法返回 false，则会自动返回一个包含 403 状态码的 HTTP 响应，也不会运行控制器的方法。

如果你打算在应用程序的其它部分处理授权逻辑，只需从 authorize 方法返回 true。

/\*\*

\* 判断用户是否有权限进行此请求。

\*

\* @return bool

\*/

public function authorize()

{

return true;

}

Tip：你可以向 authorize 方法传入所需的任何依赖项。他们会自动被 Laravel 提供的 服务容器 自动解析。

(3)、自定义验证错误信息 (自定义错误消息)

你可以通过重写表单请求的 messages 方法来自定义错误消息。此方法应返回属性 / 规则对及其对应错误消息的数组。

/\*\*

\* 获取已定义验证规则的错误消息。

\*

\* @return array

\*/

public function messages()

{

return [

'title.required' => 'A title is required',

'body.required' => 'A message is required',

];

}

(4)、自定义验证属性

如果你希望将验证消息的 :attribute 部分替换为自定义属性名称，则可以重写 attributes 方法来指定自定义名称。此方法应返回属性 / 名称对的数组。

/\*\*

\* 获取验证错误的自定义属性。

\*

\* @return array

\*/

public function attributes()

{

return [

'email' => 'email address',

];

}

--

4、手动创建验证器

//

///1/ 创建验证器

如果你不想在请求上使用 validate 方法，你可以通过 Validator facade( https://learnku.com/docs/laravel/6.x/facades ) 手动创建一个验证器示例。用 Validator facade( https://learnku.com/docs/laravel/6.x/facades ) 上的 make 方法创建一个验证器示例。

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Controllers\Controller;

use Illuminate\Support\Facades\Validator;

class PostController extends Controller

{

/\*\*

\* 保存一篇新的博客文章。

\*

\* @param Request $request

\* @return Response

\*/

public function store(Request $request)

{

$validator = Validator::make($request->all(), [

'title' => 'required|unique:posts|max:255',

'body' => 'required',

]);

if ($validator->fails()) {

return redirect('post/create')

->withErrors($validator)

->withInput();

}

// 保存博客文章…

}

}

?>

传给 make 方法的第一个参数是需要验证的数据。第二个参数则是该数据的验证规则。

如果验证失败，则可以使用 withErrors 方法把错误消息闪存到 Session 。使用这个方法进行重定向后， $errors 变量会自动和视图共享，你可以把这些消息显示给用户。withErrors 方法接收验证器、MessageBag 或 PHP Array 。

(1)、自动转跳 (自动重定向)

如果你想手动创建验证器实例，又想使用 validates 方法提供的自动重定向，那么你可以在现有的验证器示例上调用 validate 方法。如果验证失败，用户将会自动重定向。在 AJAX 请求中，则会返回 JSON 格式的响应。

Validator::make($request->all(), [

'title' => 'required|unique:posts|max:255',

'body' => 'required',

])->validate();

(2)、命名错误包

如果你一个页面中有多个表单，你可以通过命名错误包来检索特定表单的错误消息。只需给 withErrors 方法传递一个名字作为第二个参数。

return redirect('register')

->withErrors($validator, 'login');

然后你就可以从 $errors 变量中获取指定表单的错误消息。

{{ $errors->login->first('email') }}

(3)、验证后钩子

验证器还允许附加回调并在验证完成后执行，以便你进行下一步的验证，甚至在消息集合中添加更多的错误消息。使用它只需在验证实例上使用 after 方法。

$validator = Validator::make(...);

$validator->after(function ($validator) {

if ($this->somethingElseIsInvalid()) {

$validator->errors()->add('field', 'Something is wrong with this field!');

}

});

if ($validator->fails()) {

//

}

--

5、处理错误信息

通过 Validator 实例调用 errors 方法，会返回 Illuminate\Support\MessageBag 实例，它拥有各种方便的方法处理错误信息。自动提供给所有视图的 $errors 变量，也是 MessageBag 类的一个实例。

(1)、自定义验证错误信息

//

///1/ 查看特定字段的第一个错误信息

要查看特定字段的第一个错误消息，可以使用 first 方法。

$errors = $validator->errors();

echo $errors->first('email');

//

///2/ 查看特定字段的所有错误消息

如果你需要获取指定字段的所有错误信息的数组，则可以使用 get 方法。

foreach ($errors->get('email') as $message) {

//

}

如果要验证表单的数组字段，你可以使用 \* 来获取每个数组元素的所有错误消息。

foreach ($errors->get('attachments.\*') as $message) {

//

}

//

///3/ 查看所有字段的所有错误消息

如果你想要得到所有字段的所有错误消息，可以使用 all 方法。

foreach ($errors->all() as $message) {

//

}

//

///4/ 判断特定字段是否含有错误消息

has 方法可以被用来判断指定字段是否存在错误信息。

if ($errors->has('email')) {

//

}

//

///5/ 自定义错误消息

如果有需要，你也可以使用自定义错误信息代替默认值进行验证。有几种方法可以指定自定义信息。首先，你可以将自定义信息作为第三个参数传递给 Validator::make 方法。

$messages = [

'required' => 'The :attribute field is required.',

];

$validator = Validator::make($input, $rules, $messages);

在这个例子中， :attribute 占位符会被验证字段的实际名称替换。除此之外，你还可以在验证消息中使用其它占位符。例如。

$messages = [

'same' => 'The :attribute and :other must match.',

'size' => 'The :attribute must be exactly :size.',

'between' => 'The :attribute value :input is not between :min - :max.',

'in' => 'The :attribute must be one of the following types: :values',

];

//

///6/ 为给定属性指定自定义信息

有时候你可能只想为特定的字段自定义错误信息。只需在属性名称后使用「点」来指定验证的规则即可。

$messages = [

'email.required' => 'We need to know your e-mail address!',

];

//

///7/ 在 PHP 文件中指定自定义信息

在大多数情况下，您可能会在文件中指定自定义信息，而不是直接将它们传递给 Validator 。为此，需要把你的信息放置于 resources/lang/xx/validation.php 语言文件内的 custom 数组中。

'custom' => [

'email' => [

'required' => 'We need to know your e-mail address!',

],

],

//

///8/ 在 PHP 文件中指定自定义属性

如果你希望将验证信息的 :attribute 部分替换为自定义属性名称，你可以在 resources/lang/xx/validation.php 语言文件的 attributes 数组中指定自定义名称。

'attributes' => [

'email' => 'email address',

],

//

///9/ 在 PHP 文件中指定自定义值

有时您可能需要将验证信息的 :value 替换为自定义的表示形式。例如，指定 payment\_type 的值为 cc。

$request->validate([

'credit\_card\_number' => 'required\_if:payment\_type,cc'

]);

如果此验证规则失败，将生成以下错误信息。

The credit card number field is required when payment type is cc.

您可以通过在 validation 语言文件中定义 values 数组指定自定义值表示形式，而不是将 cc 显示为 payment\_type 的值。

'values' => [

'payment\_type' => [

'cc' => 'credit card'

],

],

如果验证失败，将生成以下信息。

The credit card number field is required when payment type is credit card.

--

6、可用验证规则

以下是所有可用验证规则及其功能的列表。

(1)、accepted

验证字段必须是 yes， on， 1，或 true。这在确认「服务条款」是否同意时相当有用。

(2)、active\_url

根据 PHP 函数 dns\_get\_record ，验证字段必须具有有效的 A 或 AAAA 记录。

(3)、after:date

验证字段必须是给定日期之后的值。日期值将传递到 PHP 函数 strtotime。

'start\_date' => 'required|date|after:tomorrow'

您可以指定另一个要与日期进行比较的字段，而不是传递要由 strtotime 处理的日期字符串。

'finish\_date' => 'required|date|after:start\_date'

(4)、after\_or\_equal:date

验证字段必须是在给定日期之后或与此日期相同的值。更多信息，请参阅 after( https://learnku.com/docs/laravel/6.x/validation/5144#rule-after )。

(5)、alpha

验证字段必须完全由字母构成。

(6)、alpha\_dash

验证字段可能包含字母、数字，以及破折号 (-) 和下划线 ( \_ )。

(7)、alpha\_num

验证字段必须是完全是字母、数字。

(8)、array

验证的字段必须是一个 PHP 数组。

(9)、bail

在第一次验证失败后停止运行验证规则。

(10)、before:date

正在验证的字段必须是给定日期之前的值。这个日期将传递到 PHP 的 strtotime 函数中。此外，与 after 规则一样，另一个正在验证的字段可以作为 date 的值。

(11)、before\_or\_equal:date

验证字段必须是在给定日期之前或与之相同的日期。这个日期值将会被传递给 PHP 的 strtotime 函数来计算。

(12)、between:min,max

验证字段的大小必须在给定的 min 和 max 之间。字符串、数字、数组和文件的计算方式都使用 size 方法。

(13)、boolean

验证的字段必须可以转换为 Boolean 类型。 可接受的输入为 true ， false ， 1 ， 0 ， "1" 和 "0" 。

(14)、confirmed

验证字段必须具有匹配字段 foo\_confirmation 。例如，验证字段为 password ，输入中必须存在与之匹配的 password\_confirmation 字段。

(15)、date

根据 PHP strtotime 函数，验证的字段必须是有效的日期。

(16)、date\_equals:date

验证字段必须等于给定日期。日期将传递到 PHP strtotime 函数。

(17)、date\_format:format

验证字段必须匹配给定的日期格式。当验证某个字段的时候，你应该 只使用 date 或者 date\_format ，而不是同时使用。此验证规则支持 PHP 所有的 DateTime 类( https://www.php.net/manual/es/class.datetime.php )。

(18)、different:field

验证的字段值必须与字段 field 的值不同。

(19)、digits:value

验证的字段必须为 numeric ，并且必须具有确切长度 value。

(20)、digits\_between:min,max

验证字段的长度必须在给定的 min 和 max 之间。

(21)、dimensions

验证的文件必须是图片并且图片比例必须符合规则。

'avatar' => 'dimensions:min\_width=100,min\_height=200'

可用的规则为:min\_width, max\_width, min\_height, max\_height, width, height, ratio。

ratio  约束应该表示为宽度除以高度。 这可以通过像 3/2 这样的语句或像 1.5 这样的 float 来指定。

'avatar' => 'dimensions:ratio=3/2'

//

由于此规则需要多个参数，因此你可以 Rule::dimensions 方法来构造可读性高的规则。

use Illuminate\Validation\Rule;

Validator::make($data, [

'avatar' => [

'required',

Rule::dimensions()->maxWidth(1000)->maxHeight(500)->ratio(3 / 2),

],

]);

(22)、distinct

验证数组时，指定的字段不能有任何重复值。

'foo.\*.id' => 'distinct'

(23)、email

验证的字段必须符合 e-mail 地址格式。当前版本，此种验证规则由 egulias/email-validator 提供支持。默认使用 RFCValidation 验证样式，但你也可以使其他验证样式。

'email' => 'email:rfc,dns'

//

例子使用 RFCValidation 和 DNSCheckValidation 验证样式。所有可用验证样式列表。

rfc: RFCValidation

strict: NoRFCWarningsValidation

dns: DNSCheckValidation

spoof: SpoofCheckValidation

filter: FilterEmailValidation

当下版本 filter 验证规则使用 PHP 的 filter\_var 方法进行验证，在 5.8 版本接入 Laravel。

(24)、ends\_with:foo,bar,...

验证的字段必须以给定的值之一结尾。

(25)、exists:table,column

验证的字段必须存在于给定的数据库表中。

//

//// Exists 规则的基本用法

// 未指定列名

'state' => 'exists:states'

如果未指定 column 选项，则将使用字段名称。

//

// 指定列名

'state' => 'exists:states,abbreviation'

//

// 指定数据库连接

如果你需要指定 exists 方法用来查询的数据库。你可以通过使用「点」语法将数据库的名称添加到数据表前面来实现这个目的。

'email' => 'exists:connection.staff,email'

//

// 定义查询

如果要自定义验证规则执行的查询，可以使用 Rule 类来定义规则。在这个例子中，我们使用数组指定验证规则，而不是使用 | 字符来分隔它们。

use Illuminate\Validation\Rule;

Validator::make($data, [

'email' => [

'required',

Rule::exists('staff')->where(function ($query) {

$query->where('account\_id', 1);

}),

],

]);

(26)、file

验证的字段必须是成功上传的文件。

(27)、filled

验证的字段在存在时不能为空。

(28)、gt:field

验证字段必须大于给定的 field。两个字段必须是相同的类型。字符串、数字、数组和文件都使用 size 进行相同的评估。

(29)、验证字段必须大于或等于给定的 field。两个字段必须是相同的类型。字符串、数字、数组和文件都使用 size 进行相同的评估。

(30)、image

验证的文件必须是图片 (jpeg, png, bmp, gif, svg, or webp)

(31)、in:foo,bar,...

验证字段必须包含在给定的值列表中。由于此规则通常要求您 implode 数组，因此可以使用 Rule :: in 方法流畅地构造规则。

use Illuminate\Validation\Rule;

Validator::make($data, [

'zones' => [

'required',

Rule::in(['first-zone', 'second-zone']),

],

]);

(32)、in\_array:anotherfield.\*

验证的字段必须存在于另一个字段 anotherfield 的值中。

(33)、integer

验证的字段必须是整数。

{注} 此种验证规则不是验证数据是 “integer” 类型，仅验证字符串或数值包含一个 integer。

(34)、ip

验证的字段必须是 IP 地址。

(35)、ipv4

验证的字段必须是 IPv4 地址。

(36)、ipv6

验证的字段必须是 IPv6 地址。

(37)、json

验证的字段必须是有效的 JSON 字符串。

(38)、lt:field

验证的字段必须小于给定的 field.。这两个字段必须是相同的类型。字符串、数值、数组和文件大小的计算方式与 size 方法进行评估。

(39)、lte:field

验证中的字段必须小于或等于给定的 字段 。这两个字段必须是相同的类型。字符串、数值、数组和文件大小的计算方式与 size 方法进行评估。

(40)、max:value

验证中的字段必须小于或等于 value。字符串、数字、数组或是文件大小的计算方式都用 size 规则。

(41)、mimetypes:text/plain,...

验证的文件必须具备与列出的其中一个扩展相匹配的 MIME 类型。

'video' => 'mimetypes:video/avi,video/mpeg,video/quicktime'

为了确定上传文件的 MIME，框架将会读取文件，然后自动推测文件 MIME 类型，这可能与客户端提供的 MIME 类型不一致 。

(42)、mimes:foo,bar,...

验证的文件必须具有与列出的其中一个扩展名相对应的 MIME 类型。

//

//// MIME 规则基本用法

'photo' => 'mimes:jpeg,bmp,png'

即使你可能只需要验证指定扩展名，但此规则实际上会去验证文件的 MIME 类型，其通过读取文件内容来推测它的 MIME 类型。

可以在以下链接中找到完整的 MIME 类型列表及相对应的扩展名：https://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types 。

(43)、min:value

验证字段必须具有最小值。 字符串，数值，数组，文件大小的计算方式都与 size 规则一致。

(44)、not\_in:foo,bar,...

验证字段不能包含在给定的值的列表中。 使用 Rule::notIn 方法可以更流畅的构建这个规则。

use Illuminate\Validation\Rule;

Validator::make($data, [

'toppings' => [

'required',

Rule::notIn(['sprinkles', 'cherries']),

],

]);

(45)、not\_regex:pattern

验证字段必须与给定的正则表达式不匹配。

验证时，这个规则使用 PHP preg\_match 函数。指定的模式应遵循 preg\_match 所需的相同格式，也包括有效的分隔符。 例如： 'email' => 'not\_regex:/^.+$/i'。

注意： 当使用 regex / not\_regex 模式时， 可能需要在数组中指定规则，而不是使用 | 分隔符 ，特别是在正则表达式包含 | 字符 的情况下。

(46)、nullable

验证字段可以为 null。这在验证基本数据类型时特别有用，例如可以包含空值的字符串和整数。

(47)、numeric

验证字段必须为数值。

(48)、present

验证字段必须存在于输入数据中，但可以为空。

(49)、regex:pattern

验证字段必须与给定的正则表达式匹配。

验证时，这个规则使用 PHP 的 preg\_match 函数。 指定的模式应遵循 preg\_match 所需的相同格式，也包括有效的分隔符。 例如： 'email' => 'regex:/^.+$/i' 。

注意： 当使用 regex / not\_regex 模式时， 可能需要在数组中指定规则，而不是使用 | 分隔符 ，特别是在正则表达式包含 | 字符 的情况下。

(50)、required

验证的字段必须存在于输入数据中，而不是空。如果满足以下条件之一，则字段被视为「空」。

值为 null 。

值为空字符串。

值为空数组或空 Countable 对象。

值为无路径的上传文件。

(51)、required\_if:anotherfield,value1,value2,value3...

如果其它字段 \_anotherfield\_ 为任一值 \_value1\_ 或 \_value2\_ 或 \_value3\_ 等（也可只有一个 \_value1\_） ，则此验证字段必须存在且不为空。

如果您需要构造更复杂的条件 required\_if 规则， 您可以使用 Rule::requiredIf 方法。这个方法可以接受一个布尔值或是一个闭包函数，当传递闭包函数时，闭包函数应该返回 true 或 false ，以表明是否需要验证此字段。

use Illuminate\Validation\Rule;

Validator::make($request->all(), [

'role\_id' => Rule::requiredIf($request->user()->is\_admin),

]);

Validator::make($request->all(), [

'role\_id' => Rule::requiredIf(function () use ($request) {

return $request->user()->is\_admin;

}),

]);

(52)、required\_unless:anotherfield,value,...

如果其它字段 \_anotherfield\_ 不等于任一值 \_value\_ ，则此验证字段必须存在且不为空。

(53)、required\_with:foo,bar,...

在其他任一指定字段出现时，验证的字段才必须存在且不为空。

(54)、required\_with\_all:foo,bar,...

只有在其他指定字段全部出现时，验证的字段才必须存在且不为空。

(55)、required\_without:foo,bar,...

在其他指定任一字段不出现时，验证的字段才必须存在且不为空。

(56)、required\_without\_all:foo,bar,...

只有在其他指定字段全部不出现时，验证的字段才必须存在且不为空。

(57)、same:field

验证字段必须与给定字段相匹配。

(58)、size:value

验证字段必须与给定值的大小一致。对于字符串，value 对应字符数。对于数字，value 对应给定的整数值。对于数组，size 对应数组的 count 值。对于文件，size 对应文件大小（单位 kb）。

(59)、starts\_with:foo,bar,...

验证字段必须以给定值之一开头。

(60)、string

验证字段必须是一个字符串。如果允许这个字段为 null，需要给这个字段分配 nullable 规则。

(61)、timezone

验证字段必须为符合 PHP 函数 timezone\_identifiers\_list 所定义的有效时区标识。

(62)、unique:table,column,except,idColumn

验证字段在给定的数据库表中必须是唯一的。

//

/// 指定自定义列名

column 选项可用于指定相应数据库列的字段。 如果未指定 column 选项，则使用字段本身名称。

'email' => 'unique:users,email\_address'

//

/// 自定义数据库连接

有时，你可能需要为验证器创建的数据库查询设置自定义连接。上面的例子中，将 unique:users 设置为验证规则，等于使用默认数据库连接来查询数据库。如果要对其进行修改，请使用「点」语法来指定连接和表名。

'email' => 'unique:connection.users,email\_address'

//

/// 强迫 Unique 规则忽略指定ID

有时，你可能希望在进行字段唯一性验证时忽略指定 ID 。例如， 在「更新个人资料」页面会包含用户名、邮箱和地点。这时你会想要验证更新的 E-mail 值是否唯一。如果用户仅更改了用户名字段而没有改 E-mail 字段，就不需要抛出验证错误，因为此用户已经是这个 E-mail 的拥有者了。

使用 Rule 类定义规则来指示验证器忽略用户的 ID 。这个例子中通过数组来指定验证规则，而不是使用 | 字符来分隔。

use Illuminate\Validation\Rule;

Validator::make($data, [

'email' => [

'required',

Rule::unique('users')->ignore($user->id),

],

]);

{备注} 您永远不应该将任何用户控制的请求输入传递给 ignore 方法。您应该只通过 Eloquent 模型的实例来传递系统生成的唯一 ID ，例如自动递增 ID 或 UUID 。 否则，您的应用程序将更容易受到 SQL 注入攻击。

您可以传递整个模型实例，而不是将模型实例的主键值 传递给 ignore 方法。 Laravel 将自动从模型实例中获取主键值。

Rule::unique('users')->ignore($user)

如果您的数据表使用的主键名称不是 id ，那就在调用 ignore 方法时指定字段的名称。

Rule::unique('users')->ignore($user->id, 'user\_id')

默认情况下， unique 规则将检查与要验证字段名称相匹配的列的唯一性。 但是，您可以传递一个不同的列名作为 unique 方法的第二个参数。

Rule::unique('users', 'email\_address')->ignore($user->id)

//

/// 增加额外的 Where 语句

您也可以通过 where 方法指定额外的查询条件。例如， 我们添加 account\_id 为 1 的约束。

'email' => Rule::unique('users')->where(function ($query) {

return $query->where('account\_id', 1);

})

(63)、url

验证的字段必须是有效的 URL。

(64)、uuid

验证字段必须是有效的 RFC 4122（版本 1,3,4 或 5）通用唯一标识符（UUID）。

--

7、按条件添加验证规则 (按条件增加规则)

(1)、存在时则验证

在某些情况下，你可能希望将要验证的字段存在于输入数组中时，才对该字段执行验证。可以在规则列表中增加 sometimes 来实现。

$v = Validator::make($data, [

'email' => 'sometimes|required|email',

]);

在上面的例子中， email 字段只有在 $data 数组中存在时才会被验证。

Tip：如果你尝试验证应该始终存在但可能为空的字段，请查阅 可选字段的注意事项( https://learnku.com/docs/laravel/6.x/validation/5144#a-note-on-optional-fields )。

(2)、复杂的条件验证

有时候你可能需要增加基于更复杂的条件逻辑的验证规则。例如，你可以希望某个指定字段在另一个字段的值超过 100 时才为必填。或者当某个指定字段存在时，另外两个字段才能具有给定的值。增加这样的验证条件并不难。首先，使用 静态规则 创建一个 Validator 实例。

$v = Validator::make($data, [

'email' => 'required|email',

'games' => 'required|numeric',

]);

假设我们有一个专为游戏收藏家所设计的网页应用程序。如果游戏收藏家收藏超过一百款游戏，我们会希望他们来说明下为什么他们会拥有这么多游戏。比如说他们有可能经营了一家游戏分销商店，或者只是为了享受收集的乐趣。为了在特定条件下加入此验证需求，可以在 Validator 实例中使用 sometimes 方法。

$v->sometimes('reason', 'required|max:500', function ($input) {

return $input->games >= 100;

});

传入 sometimes 方法的第一个参数是要用来验证的字段名称。第二个参数是我们想使用的验证规则。 闭包 作为第三个参数传入，如果其返回 true ， 则额外的规则就会被加入。这个方法可以轻松地创建复杂的条件验证。你甚至可以一次对多个字段增加条件验证。

$v->sometimes(['reason', 'cost'], 'required', function ($input) {

return $input->games >= 100;

});

Tip：传入 闭包 的 $input 参数是 Illuminate\Support\Fluent 的一个实例，可用来访问你的输入或文件对象。

--

8、验证数组

验证表单的输入为数组的字段也不难。你可以使用 「点」方法来验证数组中的属性。例如，如果传入的 HTTP 请求中包含 photos[profile] 字段， 可以如下验证。

$validator = Validator::make($request->all(), [

'photos.profile' => 'required|image',

]);

你也可以验证数组中的每个元素。例如，要验证指定数组输入字段中的每一个 email 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',

]);

同理，你可以在语言文件定义验证信息时使用 \* 字符，为基于数组的字段使用单个验证消息。

'custom' => [

'person.\*.email' => [

'unique' => 'Each person must have a unique e-mail address',

]

],

--

9、自定义验证规则

(1)、使用验证规则对象 (使用规则对象)

Laravel 提供了许多有用的验证规则；同时也支持自定义规则。注册自定义验证规则的方法之一，就是使用规则对象。可以使用 Artisan 命令 make:rule 来生成新的规则对象。接下来，让我们用这个命令生成一个验证字符串是否是大写的规则。Laravel 会将新的规则存放在 app/Rules 目录中。

//

///1/ 生成规则对象，验证字符串是否是大写的规则

php artisan make:rule Uppercase

//

一旦创建了规则，我们就可以定义它的行为。规则对象包含两个方法： passes 和 message。passes 方法接收属性值和名称，并根据属性值是否符合规则而返回  true 或 false。 message 方法应返回验证失败时应使用的验证错误消息。

//

///2/ Uppercase.php

<?php

namespace App\Rules;

use Illuminate\Contracts\Validation\Rule;

class Uppercase implements Rule

{

/\*\*

\* 确定验证规则是否通过。

\*

\* @param string $attribute

\* @param mixed $value

\* @return bool

\*/

public function passes($attribute, $value)

{

return strtoupper($value) === $value;

}

/\*\*

\* 获取验证错误消息。

\*

\* @return string

\*/

public function message()

{

return 'The :attribute must be uppercase.';

}

}

?>

//

当然， 如果你希望从翻译文件中返回一个错误消息，你可以从 message 方法中调用辅助函数 trans。

/\*\*

\* 获取验证错误消息。

\*

\* @return string

\*/

public function message()

{

return trans('validation.uppercase');

}

//

///3/ 使用规则

一旦规则对象被定义好后，你可以通过将规则对象的实例和其他验证规则一起来传递给验证器。

use App\Rules\Uppercase;

$request->validate([

'name' => ['required', 'string', new Uppercase],

]);

(2)、使用闭包

如果你在应用程序中只需要一次自定义规则的功能，则可以使用闭包代替规则对象。闭包接收属性的名称、属性的值、失败回调，如果验证失败需要调用 $fail。

$validator = Validator::make($request->all(), [

'title' => [

'required',

'max:255',

function ($attribute, $value, $fail) {

if ($value === 'foo') {

$fail($attribute.' is invalid.');

}

},

],

]);

(3)、使用扩展

注册自定义的验证规则的另一种方法是使用 Validator facade( https://learnku.com/docs/laravel/5.8/facades ) 中的 extend 方法。让我们在 服务容器( https://learnku.com/docs/laravel/5.8/providers ) 中使用这个方法来注册自定义验证规则。

<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

use Illuminate\Support\Facades\Validator;

class AppServiceProvider extends ServiceProvider

{

/\*\*

\* 注册服务提供器。

\*

\* @return void

\*/

public function register()

{

//

}

/\*\*

\* 引导应用程序。

\*

\* @return void

\*/

public function boot()

{

Validator::extend('foo', function ($attribute, $value, $parameters, $validator) {

return $value == 'foo';

});

}

}

?>

自定义的验证闭包接收四个参数：要被验证的属性名称 $attribute、属性的值 $value、传入验证规则的参数数组 $parameters 、以及 Validator 实列。

除了使用闭包，你也可以传入类和方法到 extend 方法中。

Validator::extend('foo', 'FooValidator@validate');

//

///1/ 定义错误消息

你还需要为自定义规则定义错误信息。你可以使用内联自定义消息数组或者在验证语言文件中添加条目来实现这一功能。 消息应该被放到数组的第一位， 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.",默认情况下，当所要验证的属性不存在或包含一个空字符串时，使用包含自定义扩展的正常的验证规则是不会执行的。例如，unique 规则将不会检验空字符串。

$rules = ['name' => 'unique:users,name'];

$input = ['name' => ''];

Validator::make($input, $rules)->passes(); // true

//

如果即使属性为空也要验证规则，则一定要暗示属性是必须的。要创建这样一个「隐式」扩展，可以使用 Validator::extendImplicit() 方法。

Validator::extendImplicit('foo', function ($attribute, $value, $parameters, $validator) {

return $value == 'foo';

});

注意：「隐式」扩展只暗示该属性是必需的。至于它到底是缺失还是空值这取决于你。

// ...其余的验证错误消息...//

///1/ 隐式规则对象

如果你想要在属性为空时执行规则对象，你应该实现 Illuminate\Contracts\Validation\ImplicitRule 接口。这个接口将充当验证器的「标记接口」；因此，它不包含你要实现的任何方法。

--

--

十一、错误

https://learnku.com/docs/laravel/6.x/errors/5145

--

//

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。你可以在 服务容器 的 boot 方法中执行如下操作。

/\*\*

\* 引导应用程序。

\*

\* @return void

\*/

public function boot()

{

Validator::extend(...);

Validator::replacer('foo', function ($message, $attribute, $rule, $parameters) {

return str\_replace(...);

});

}

(4)、隐式扩展

默认情况下，当所要验证的属性不存在或包含一个空字符串时，使用包含自定义扩展的正常的验证规则是不会执行的。例如，unique 规则将不会检验空字符串。

$rules = ['name' => 'unique:users,name'];

$input = ['name' => ''];

Validator::make($input, $rules)->passes(); // true

//

如果即使属性为空也要验证规则，则一定要暗示属性是必须的。要创建这样一个「隐式」扩展，可以使用 Validator::extendImplicit() 方法。

Validator::extendImplicit('foo', function ($attribute, $value, $parameters, $validator) {

return $value == 'foo';

});

注意：「隐式」扩展只暗示该属性是必需的。至于它到底是缺失还是空值这取决于你。

//

///1/ 隐式规则对象

如果你想要在属性为空时执行规则对象，你应该实现 Illuminate\Contracts\Validation\ImplicitRule 接口。这个接口将充当验证器的「标记接口」；因此，它不包含你要实现的任何方法。

--

--

十一、错误 (错误处理)

https://learnku.com/docs/laravel/6.x/errors/5145

--

1、介绍

当你开始一个新的 Laravel 项目时，它已经为您配置了错误和异常处理。app\exceptions\handler 类用于记录应用程序触发的所有异常，然后将其呈现回用户。我们将在本文中深入讨论这个类。

--

2、配置

你的 config/app.php 配置文件中的 debug 选项决定了对于一个错误实际上将显示多少信息给用户。默认情况下，该选项的设置将遵照存储在 .env 文件中的 APP\_DEBUG 环境变量的值。

对于本地开发，你应该将 APP\_DEBUG 环境变量的值设置为 true 。在生产环境中，该值应始终为 false 。如果在生产中将该值设置为 true ，则可能会将敏感配置值暴露给应用程序的终端用户。

--

3、异常处理

(1)、Report 方法

所有异常都是由 App\Exceptions\Handler 处理。 这个类包含了两个方法：report 和 render。我们将详细的剖析这些方法。report 方法用于记录异常或将它们发送给如 Bugsnag( https://bugsnag.com/ ) 或 Sentry( https://github.com/getsentry/sentry-laravel ) 等外部服务。默认情况下，report 方法将异常传递给记录异常的基类。不过，你可以用任何自己喜欢的方式来记录异常。

例如，如果你需要以不同方式报告不同类型的异常，则可以使用 PHP 的 instanceof 比较运算符。

/\*\*

\* 报告或记录异常

\*

\* 这是一个向 Sentry、Bugsnag 等外部服务发送异常信息的好位置

\*

\* @param \Exception $exception

\* @return void

\*/

public function report(Exception $exception)

{

if ($exception instanceof CustomException) {

//

}

parent::report($exception);

}

Tip：不要在 report 方法中进行太多的 instanceof 检查，而应考虑使用 可报告异常（Reportable exception https://learnku.com/docs/laravel/6.x/errors#renderable-exceptions ）。

//

///1/ 全局日志

在正常情况下，Laravel 会自动将当前用户的 ID 作为数据添加到每一条异常日志中。 你可以在通过重写 App\Exceptions\Handler 类中的 context 方法来定义你的全局环境变量。 之后，这个变量将包含在每一条异常日志中。

/\*\*

\* 定义默认的环境变量

\*

\* @return array

\*/

protected function context()

{

return array\_merge(parent::context(), [

'foo' => 'bar',

]);

}

//

///2/ report 辅助函数

有时你可能需要报告异常，但又不希望终止当前请求的处理。 report 辅助函数允许你使用异常处理器的 report 方法在不显示错误页面的情况下快速报告异常。

public function isValid($value)

{

try {

// 验证值...

} catch (Exception $e) {

report($e);

return false;

}

}

//

///3/ 按类型忽略异常

异常处理器的 $dontReport 属性包含一组不会被记录的异常类型。例如，由 404 错误导致的异常以及其他几种类型的错误不会写入日志文件。你可以根据需要添加其他异常类型到此数组中。

/\*\*

\* 不应被报告的异常类型清单

render 方法负责将给定的异常转换为将被发送回浏览器的 HTTP 响应。默认情况下，异常将传递给为你生成响应的基类。不过，你可以按自己意愿检查异常类型或返回自己的自定义响应。

\*

\* @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,

];

(2)、Render 方法

render 方法负责将给定的异常转换为将被发送回浏览器的 HTTP 响应。默认情况下，异常将传递给为你生成响应的基类。不过，你可以按自己意愿检查异常类型或返回自己的自定义响应。

/\*\*

\* 将异常转换为 HTTP 响应。

\*

\* @param \Illuminate\Http\Request $request

\* @param \Exception $exception

\* @return \Illuminate\Http\Response

\*/

public function render($request, Exception $exception)

{

if ($exception instanceof CustomException) {

return response()->view('errors.custom', [], 500);

}

return parent::render($request, $exception);

}

(3)、Reportable & Renderable 异常

除了在异常处理器的 report 和 render 方法中检查异常类型，你还可以直接在自定义异常上定义 report 和 render 方法。当定义了这些方法时，它们会被框架自动调用。

<?php

namespace App\Exceptions;

use Exception;

class RenderException extends Exception

{

/\*\*

\* 报告异常

\*

\* @return void

\*/

public function report()

{

//

}

/\*\*

\* 转换异常为 HTTP 响应

\*

\* @param \Illuminate\Http\Request

\* @return \Illuminate\Http\Response

\*/

public function render($request)

{

return response(...);

}

}

?>

Tip：你可以声明 report 方法和必要参数，它们将通过 Laravel 的 服务容器( https://learnku.com/docs/laravel/6.x/container ) 自动注入方法中。

--

4、HTTP 异常

一些异常用于描述产生自服务器的 HTTP 错误代码。例如，「页面未找到」错误 (404), 「未经授权的错误」(401) ，甚至可以是开发人员引起的 500 错误。 你可以使用 abort 辅助函数从应用程序的任何地方生成这样的响应。

abort(404);

辅助函数 abort 会立即引发一个由异常处理器渲染的异常。你还可选择性地提供响应文本。

abort(403, 'Unauthorized action.');

(1)、自定义 HTTP 错误页面

Laravel 可以轻松显示各种 HTTP 状态代码的自定义错误页面。例如，如果你希望自定义 404 HTTP 状态码的错误页面，可以创建一个 resources/views/errors/404.blade.php 视图文件。该文件将被用于你的应用程序产生的所有 404 错误。此目录中的视图文件的命名应匹配它们对应的 HTTP 状态码。由 abort 函数引发的 HttpException 实例将作为 $exception 变量传递给视图。

你可以使用 vendor:publish Artisan 命令来定义错误模板页面。模板页面生成后，就可以自定义模板页面的内容。

//

/// 定义错误模板页面

php artisan vendor:publish --tag=laravel-errors

--

--

十二、日志

https://learnku.com/docs/laravel/6.x/logging/5146

--

1、简介

为了帮助你更多的了解应用程序中到底发生了什么，Laravel 提供了强大的日志服务，允许你将日志消息、系统错误日志记录到文件，甚至使用 Slack 通知到你的整个团队。

在 Laravel 框架中，Laravel 使用 Monolog( https://github.com/Seldaek/monolog ) 库，它为各种强大的日志处理提供支持。Laravel 使配置这些处理程序变得简单，允许你混合并匹配它们自定义的应用程序日志处理。

--

2、配置

所有的应用程序日志系统配置都位于 config/logging.php 配置文件中。这个文件允许你配置你的应用程序日志通道，所以务必查看每个可用的通道及它们的选项。当然，我们将在下面回顾一些常用的选项。

默认情况下，Laravel 将使用 stack 去记录日志消息。stack 通道被用来将多个日志通道聚合到一个单一的通道中。关于堆栈的更多信息，查看 以下文档( https://learnku.com/docs/laravel/6.x/logging/5146#building-log-stacks )。

(1)、配置通道名

默认情况下，Monolog 使用与当前环境匹配的『通道名称』进行实例化，比如 production 或者 local。要改变这个值，需添加一个 name 选项到你的通道配置中。

'stack' => [

'driver' => 'stack',

'name' => 'channel-name',

'channels' => ['single', 'slack'],

],

(2)、可用的通道驱动

名称/描述

stack 一个便于创建『多通道』通道的包装器

single 单个文件或者基于日志通道的路径 (StreamHandler)

daily 一个每天轮换的基于 Monolog 驱动的 RotatingFileHandler

slack 一个基于 Monolog 驱动的 SlackWebhookHandler

papertrail 一个基于 Monolog 驱动的 SyslogUdpHandler

syslog 一个基于 Monolog 驱动的 SyslogHandler

errorlog 一个基于 Monolog 驱动的 ErrorLogHandler

monolog 一个可以使用任何支持 Monolog 处理程序的 Monolog 工厂驱动程序

custom 一个调用指定工厂创建通道的驱动程序

Tip：有关 monolog 和 custom 驱动，查看 高级通道自定义( https://learnku.com/docs/laravel/6.x/logging/5146#advanced-monolog-channel-customization )。

(3)、配置 Single 和 Daily 通道

single 和 daily 通道包含三个可选配置项：bubble、permission 和 locking。

名称/描述/默认值

bubble 消息处理后，指示消息是否推送到其他通道 true

permission 日志文件权限 0644

locking 写入之前尝试锁定日志文件 false

(4)、配置 Papertrail 通道

papertrail 通道需要 url 和 端口 配置选项。你可一一从这里获取这些值 Papertrail( https://help.papertrailapp.com/kb/configuration/configuring-centralized-logging-from-php-apps/#send-events-from-php-app )。

(5)、配置 Slack 通道

slack 通道需要 url 配置选项。这个 URL 应当与你为 Slack 团队配置的一个 incoming webhook( https://slack.com/apps/A0F7XDUAZ-incoming-webhooks ) 相匹配。

(6)、构建日志堆栈

前面说过， stack 驱动允许你在单一日志通道中整合多个通道。让我们通过一个产品级应用的配置实例来看看如何使用日志堆栈。

'channels' => [

'stack' => [

'driver' => 'stack',

'channels' => ['syslog', 'slack'],

],

'syslog' => [

'driver' => 'syslog',

'level' => 'debug',

],

'slack' => [

'driver' => 'slack',

'url' => env('LOG\_SLACK\_WEBHOOK\_URL'),

'username' => 'Laravel Log',

'emoji' => ':boom:',

'level' => 'critical',

],

],

我们来分析这个配置。首先要注意的是 stack 通过借助它的 channels 选项聚合了另外两个通道： syslog 和 slack 。因此，在记录日志消息时，这两个通道都有机会完成日志消息记录。

(7)、日志级别

请留意上面例子中 syslog 和 slack 中存在的 level 配置项。这个选项决定了需要被该通道记录的日志的最低 「级别」。Monolog （一个功能强劲的 Laravel 日志服务）接受定义在 RFC 5424 specification( https://tools.ietf.org/html/rfc5424 ) 中的全部级别： emergency、alert、 critical、 error、 warning、 notice、 info 和 debug。

// 假设我们使用 debug 方法记录日志消息。

Log::debug('An informational message.');

根据我们的配置， syslog 通道将把该消息记录到系统日志；不过因为错误消息不是 critical 或更高级别，它将不会被发送到 Slack。如果我们记录一条 emergency 消息，它将被发送给系统日志和 Slack，因为 emergency 的级别高于两个通道的最低级别限制。

Log::emergency('The system is down!');

--

3、写日志消息

可以使用 Log facade ( https://learnku.com/docs/laravel/6.x/facades ) 将信息写入日志。如前所述，日志提供定义在 RFC 5424 specification( https://tools.ietf.org/html/rfc5424 ) 中的可用日志级别： emergency、 alert、 critical、 error、 warning、 notice、 info 和 debug。

Log::emergency($message);

Log::alert($message);

Log::critical($message);

Log::error($message);

Log::warning($message);

Log::notice($message);

Log::info($message);

Log::debug($message);

因此，你可以调用这些方法中的任一方法记录相应级别的日志。默认情况下，消息被写入到在 config/logging.php 配置文件中定义的默认日志通道。

<?php

namespace App\Http\Controllers;

use App\User;

use Illuminate\Support\Facades\Log;

use App\Http\Controllers\Controller;

class UserController extends Controller

{

/\*\*

\* 显示给定用户的配置信息。

\*

\* @param int $id

\* @return Response

\*/

public function showProfile($id)

{

Log::info('Showing user profile for user: '.$id);

return view('user.profile', ['user' => User::findOrFail($id)]);

}

}

?>

//

///1/ 上下文信息

可以将上下文数据数组传递给日志方法。这些信息将被格式化，并与日志消息一起显示。

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

(1)、写入指定的通道 (写入指定通道)

有时候你可能希望将消息写入到应用默认通道之外的通道中。可以使用 Log facade 的 channel 方法获取定义在配置文件中的任一通道并将消息写入其中。

Log::channel('slack')->info('Something happened!');

//

如果想要创建一个由多通道构成的按需记录的堆栈，可以使用 stack 方法。

Log::stack(['single', 'slack'])->info('Something happened!');

--

4、先进的 Monolog 日志通道定制 (高度自定义 Monolog 通道)

(1)、自定义的 Monolog 通道 (为通道自定义 Monolog)

有时需要完全控制已存在通道的 Monolog： 比如，你可能想要为给定通道的日志处理配置自定义的 Monolog FormatterInterface 实现。

先在通道配置中定义一个 tap 数组。 tap 数组包含一个在通道创建后有机会用于自定义 Monolog 实例的类列表。

'single' => [

'driver' => 'single',

'tap' => [App\Logging\CustomizeFormatter::class],

'path' => storage\_path('logs/laravel.log'),

'level' => 'debug',

],

一旦在通道中有了 tap 选项配置，就要准备用于自定义 Monolog 实例的类。这种类这需要一个方法： \_\_invoke，它接受一个 Illuminate\Log\Logger 实例作为其参数。 Illuminate\Log\Logger 实例将所有方法调用代理到基础的 Monolog 实例。

<?php

namespace App\Logging;

class CustomizeFormatter

{

/\*\*

\* 自定义给定的日志实例。

\*

\* @param \Illuminate\Log\Logger $logger

\* @return void

\*/

public function \_\_invoke($logger)

{

foreach ($logger->getHandlers() as $handler) {

$handler->setFormatter(...);

}

}

}

?>

Tip：所有的 "tap" 类都是由 服务容器( https://learnku.com/docs/laravel/6.x/container ) 解析的，因此任何依赖它们的构造器都会自动被注入。

(2)、创建 Monolog 处理通道 (创建 Monolog 处理器通道)

Monolog 多种 可用处理器( https://github.com/Seldaek/monolog/tree/master/src/Monolog/Handler )。在某些情况下，你会希望仅创建一个带有指定处理器的 Monolog 驱动的日志类型。这些通道可以使用 monolog 驱动创建。

在使用 monolog 驱动时， handler 配置项用于指定被实例化的处理器。如果该处理器的构造器需要参数，可以使用可选的 with 配置项来指定。

'logentries' => [

'driver' => 'monolog',

'handler' => Monolog\Handler\SyslogUdpHandler::class,

'with' => [

'host' => 'my.logentries.internal.datahubhost.company.com',

'port' => '10000',

],

],

//

///1/ Monolog 格式化

使用 monolog 驱动时，Monolog 的 LineFormatter 用于默认的格式化处理器。当然，你也可以使用 formatter and formatter\_with 配置项自定义格式化处理器类型。

'browser' => [

'driver' => 'monolog',

'handler' => Monolog\Handler\BrowserConsoleHandler::class,

'formatter' => Monolog\Formatter\HtmlFormatter::class,

'formatter\_with' => [

'dateFormat' => 'Y-m-d',

],

],

如果所用的 Monolog 处理器能够提供自带的格式代处理器，可以将 formatter 配置项指定为 default。

'newrelic' => [

'driver' => 'monolog',

'handler' => Monolog\Handler\NewRelicHandler::class,

'formatter' => 'default',

],

(3)、通过工厂创建通道 (通过工厂创建渠道)

如果你想定义一个完全自定义的通道，你可以完全控制 Monolog 的实例化和配置，你可以在 config/logging.php 配置文件中指定 custom 驱动程序类型。 你的配置应该包含一个 via 选项，指向将被调用以创建 Monolog 实例的工厂类。

'channels' => [

'custom' => [

'driver' => 'custom',

'via' => App\Logging\CreateCustomLogger::class,

],

],

一旦配置了 custom 通道，就可以定义创建 Monolog 实例的类。 这个类只需要一个方法： \_\_invoke ，它就可以返回 Monolog 实例。

<?php

namespace App\Logging;

use Monolog\Logger;

class CreateCustomLogger

{

/\*\*

\* 创建一个 Monolog 实例.

\*

\* @param array $config

\* @return \Monolog\Logger

\*/

public function \_\_invoke(array $config)

{

return new Logger(...);

}

}

?>

--

--

第五卷 前端开发

--

--

一、Blade 模板 (Blade Templates)

https://learnku.com/docs/laravel/6.x/blade/5147

--

1、简介

Blade 是 Laravel 提供的一个简单而又强大的模板引擎。和其他流行的 PHP 模板引擎不同，Blade 并不限制你在视图中使用原生 PHP 代码。所有 Blade 视图文件都将被编译成原生的 PHP 代码并缓存起来，除非它被修改，否则不会重新编译，这就意味着 Blade 基本上不会给你的应用增加任何负担。Blade 视图文件使用 .blade.php 作为文件扩展名，被存放在 resources/views 目录。

--

2、模板继承

(1)、定义布局

Blade 的两个主要优点是 模板继承和区块。为方便入门，让我们先通过一个简单的例子来上手。首先，我们来研究一个「主」页面布局。因为大多数 web 应用会在不同的页面中使用相同的布局方式，因此可以很方便地定义单个 Blade 布局视图。

//

//// 应用布局视图

<!-- 保存在 resources/views/layouts/app.blade.php 文件中 -->

<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>

如你所见，该文件包含了典型的 HTML 语法。不过，请注意 @section 和 @yield 指令。 @section 指令定义了视图的一部分内容，而 @yield 指令是用来显示指定部分的内容。

现在，我们已经定义好了这个应用程序的布局，接下来，我们定义一个继承此布局的子页面。

(2)、继承布局 (扩展布局)

在定义一个子视图时，使用 Blade 的 @extends 指令指定子视图要「继承」的视图。扩展自 Blade 布局的视图可以使用 @section 指令向布局片段注入内容。就如前面的示例中所示，这些片段的内容将由布局中的 @yield 指令控制显示。

//

//// 子布局

～～～

<!-- 保存在 resources/views/child.blade.php 文件中 -->

@extends('layouts.app')

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

@section('sidebar')

@parent

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

@endsection

@section('content')

<p>This is my body content.</p>

@endsection

～～～

在这个示例中， sidebar 片段利用 @parent 指令向布局的 sidebar 追加（而非覆盖）内容。 在渲染视图时，@parent 指令将被布局中的内容替换。

Tip：和上一个示例相反，这里的 sidebar 片段使用 @endsection 代替 @show 来结尾。 @endsection 指令仅定义了一个片段， @show 则在定义的同时 立即 yield 这个片段。

//

///1/ @yield指令

@yield 指令还接受一个默认值作为第二个参数。如果被 「yield」的片段未定义，则该默认值被渲染。

@yield('content', View::make('view.name'))

//

// Blade 视图可以使用全局 view 助手自路由中返回。

Route::get('blade', function () {

return view('child');

});

--

3、组件 & 插槽

组件和插槽提供了与片段和布局类似的好处；不过组件和插槽的思维模型更易于理解。我们先来看一个可复用的「alert」组件，我们想在应用中复用它。

//

///1/ 「alert」组件

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

<div class="alert alert-danger">

{{ $slot }}

</div>

{{ $slot }} 变量将包含我们想要注入到组件的内容。现在，我们使用 Blade 的 @component 指令构建这个组件。

@component('alert')

<strong>Whoops!</strong> Something went wrong!

@endcomponent

//

若要指示 Laravel 从组件的给定数组中加载存在的第一个视图，可以使用 componentFirst 指令。

@componentFirst(['custom.alert', 'alert'])

<strong>Whoops!</strong> Something went wrong!

@endcomponent

//

///2/ 插槽

有时候为一个组件定义多个插槽是很有用的。修改 alert 组件以允许其注入 「title」。命名插槽可以通过与其匹配的 「回显」 变量显示。

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

<div class="alert alert-danger">

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

{{ $slot }}

</div>

现在，我们能够使用 @slot 指令向命名插槽注入内容。不在 @slot 指令内的内容都将传递给组件中的 $slot 变量。

@component('alert')

@slot('title')

Forbidden

@endslot

You are not allowed to access this resource!

@endcomponent

(1)、向组件传递额外的数据

有时你可能需要向组件传递额外的数据。在这种情况下，可以把包含数据组织成数组，作为 @component 指令的第二个参数。所有的数据将作为变更提供给组件模板。

@component('alert', ['foo' => 'bar'])

...

@endcomponent

(2)、给组件起别名

如果组件存储在子目录中，你可能希望给它们起个别名以方便访问。举例来说，如果一个 Blade 组件存储在 resources/views/components/alert.blade.php 中，. 就可以使用 component 方法将 components.alert 的别名命名为 alert。 通常情况下，这一过程将在 AppServiceProvider 的 boot 方法中完成。

use Illuminate\Support\Facades\Blade;

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

//

// 一旦组件有了别名，就可以使用一条指令渲染它。

@alert(['type' => 'danger'])

You are not allowed to access this resource!

@endalert

//

// 如果没有额外的插槽，还可以省略组件参数。

@alert

You are not allowed to access this resource!

@endalert

--

4、显示数据

// 可以通过包裹在双花括号内的变量显示传递给 Blade 视图的数据。比如给出如下路由。

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

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

});

//

// 就可以这样利用 name 变量显示其内容。

Hello, {{ $name }}.

//

Tip：Blade {{ }} 语句是自动经过 PHP 的 htmlspecialchars 函数传递来防范 XSS 攻击的。

不限于显示传递给视图的变量的内容，你还可以显示任一 PHP 函数的结果。实际上，你可以在 Blade 的回显语句中放置你想要的任意 PHP 代码。

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

//

///1/ 显示非转义字符

默认情况下， Blade 中 {{ }} 语句自动经由 PHP 的 htmlspecialchars 函数传递以防范 XSS 攻击。如果不希望数据被转义，可以使用下面的语法。

Hello, {!! $name !!}.

注意：在回显应用的用户提供的内容时需要谨慎小心。在显示用户提供的数据时，有必要一直使用双花括号语法转义来防范 XSS 攻击。

//

///2/ 渲染 JSON

有时，为了初始化一个 JavaScript 变量，你可能会向视图传递一个数组，并将其渲染成 JSON。例如。

<script>

var app = <?php echo json\_encode($array); ?>;

</script>

然而，你需要使用 @jsonBlade 指令，而不是手动调用 json\_encode。@json 指令和 PHP 的 json\_encode 函数具有相同的参数。

<script>

var app = @json($array);

var app = @json($array, JSON\_PRETTY\_PRINT);

</script>

{注意} 你应该只使用 @json 指令去将现有的变量渲染为 JSON。Blade 模板基于正则表达式，并且尝试将复杂表达式传递给指令这可能会导致意外故障。

@json 指令还可用于植入 Vue 组件或 data-\* 属性。

<example-component :some-prop='@json($array)'></example-component>

{注意} 在元素属性中使用 @json 需要用单引号括起来。

//

///3/ HTML 实体编码

默认情况下，Blade（和 Laravel 的 e 助手函数）将对 HTML 实体进行双重编码。

如果你想要禁用双重编码，请从 AppServiceProvider 的 boot 方法中调用 Blade::withoutDoubleEncoding 方法。

<?php

namespace App\Providers;

use Illuminate\Support\Facades\Blade;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider

{

/\*\*

\* Bootstrap any application services.

\*

\* @return void

\*/

public function boot()

{

Blade::withoutDoubleEncoding();

}

}

?>

(1)、Blade & JavaScript 框架

由于许多 JavaScript 框架也使用 “大括号” 来表示预设的表达式应该显示在浏览器中，因此可以使用 @ 符号通知 Blade 渲染引擎表达式应保持不变。例如。

<h1>Laravel</h1>

Hello, @{{ name }}.

在这个例子中， @ 符号将被 Blade 删除；然而，在 Blade 引擎中 {{ name }} 表达式将保持不变，取而代之的是 JavaScript 引擎来渲染该表达式。

//

///1/ @verbatim 指令

verbatim 指令

如果你在模板中显示了很大一部分的 JavaScript 变量，可以将 HTML 套嵌在 @verbatim 指令中，这样就不必在每个 Blade 回显语句前面加上 @ 符号了。

@verbatim

<div class="container">

Hello, {{ name }}.

</div>

@endverbatim

--

5、流程控制

(1)、If 语句

除了模板继承和显示数据之外，Blade 还为常见的 PHP 控制结构提供了便捷的快捷方式，例如条件和循环语句。这些快捷方式提供了一种简洁的处理 PHP 控制语句的方法，同时保持了与 PHP 中的相应结构的相似性。

你可以使用 @if、@elseif、@else 和 @endif 指令来构造 if 语句。这些指令的功能与它们在 PHP 中对应的语句功能相同。

@if (count($records) === 1)

I have one record!

@elseif (count($records) > 1)

I have multiple records!

@else

I don't have any records!

@endif

//

为了方便，Blade 也提供了一个 @unless 指令。

@unless (Auth::check())

You are not signed in.

@endunless

//

除了已经讨论过的条件指令之外， @isset 和 @empty 指令还可以用作各自 PHP 函数的快捷方式。

@isset($records)

// $records is defined and is not null...

@endisset

@empty($records)

// $records is "empty"...

@endempty

//

///1/ 身份验证指令

@auth 和 @guest 指令可用于快速确定当前用户是否经过身份验证或是 guest。

//

@auth

// The user is authenticated...

@endauth

//

@guest

// The user is not authenticated...

@endguest

//

如果需要，可以在使用 @auth 和 @guest 指令时指定应被校验的 身份验证( https://learnku.com/docs/laravel/6.x/authentication )。

@auth('admin')

// The user is authenticated...

@endauth

@guest('admin')

// The user is not authenticated...

@endguest

//

///2/ 你可以使用 @hasSection 指令检查片段是否包含指定内容。

@hasSection('navigation')

<div class="pull-right">

@yield('navigation')

</div>

<div class="clearfix"></div>

@endif

(2)、Switch 语句

Switch 语句可以使用 @switch, @case, @break, @default 和 @endswitch 指令来构造。

@switch($i)

@case(1)

First case...

@break

@case(2)

Second case...

@break

@default

Default case...

@endswitch

(3)、循环

除了条件语句之外，Blade 还提供了使用 PHP 循环结构的简单指令。同样，这些指令的每个函数都与它们相对于的 PHP 函数相同。

@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

{提示} 循环时，你可以使用 循环遍历( https://learnku.com/docs/laravel/6.x/blade/5147#the-loop-variable ) 来获取循环中的有关信息，例如你是否在循环的第一次或最后一次迭代中。

//

// 使用循环时，也可以结束循环或跳过当前迭代。

@foreach ($users as $user)

@if ($user->type == 1)

@continue

@endif

<li>{{ $user->name }}</li>

@if ($user->number == 5)

@break

@endif

@endforeach

//

// 你也可以在一行中声明带有条件的指令。

@foreach ($users as $user)

@continue($user->type == 1)

<li>{{ $user->name }}</li>

@break($user->number == 5)

@endforeach

(4)、循环变量

循环过程中，在循环体内有一个可用的 $loop 变量。该变量提供了用于访问诸如当前循环的索引、当前是否为第一次或最后一次循环之类的少数有用的信息的途径。

@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

//

// 在嵌套循环中，可以借助 parent 属性访问父循环的 $loop 变量：

@foreach ($users as $user)

@foreach ($user->posts as $post)

@if ($loop->parent->first)

This is first iteration of the parent loop.

@endif

@endforeach

@endforeach

//

// $loop 变量还包含其它几种有用的属性。

属性/描述

$loop->index 当前迭代的索引 (从 0 开始计数)。

$loop->iteration 当前循环迭代 (从 1 开始计算)。

$loop->remaining 循环中剩余迭代的数量。

$loop->count 被迭代的数组元素的总数。

$loop->first 是否为循环的第一次迭代。

$loop->last 是否为循环的最后一次迭代。

$loop->even 是否是循环中的偶数迭代。

$loop->odd 是否是循环中的奇数迭代。

$loop->depth 当前迭代的嵌套深度级数。

$loop->parent 嵌套循环中，父循环的循环变量。

(5)、注释

Blade 也允许在视图中定义注释。不过与 HTML 注释不同，Blade 注释不会包含在返回给应用的 HTML 中。

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

(6)、PHP

某些情况下，在视图中嵌入 PHP 代码很有用。可以在模板中使用 @php 指令执行原生的 PHP 代码块。

@php

//

@endphp

Tip：尽管 Blade 提供了这个特性，但频繁使用意味着模板中嵌入了过多的逻辑。

--

6、表单

(1)、CSRF 字段

只要在应用中定义了 HTML 表单，就一定要在表单中包含隐藏的 CSRF 令牌域，这样一来 CSRF 保护( https://learnku.com/docs/laravel/laravel/6.x/csrf ) 中间件就能校验请求。可以使用 Blade 的 @csrf 指令生成令牌域。

<form method="POST" action="/profile">

@csrf

...

</form>

(2)、Method 字段 (Method 域)

HTML 表单不能发出 PUT、 PATCH 及 DELETE 请求，需要加入隐藏的 \_method 域来模仿这些 HTTP 动词。Blade 的 @method 指令能够帮你创建这个域。

<form action="/foo/bar" method="POST">

@method('PUT')

...

</form>

(3)、验证错误

@error 指令可以快速检查 验证错误消息( https://learnku.com/docs/laravel/6.x/validation#quick-displaying-the-validation-errors ) 这个属性是否存在。在 @error 指令中，你可以使用 $message 变量来显示验证错误内容。

<!-- /resources/views/post/create.blade.php -->

<label for="title">Post Title</label>

<input id="title" type="text" class="@error('title') is-invalid @enderror">

@error('title')

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

@enderror

//

你也可以指定错误验证字段名字( https://learnku.com/docs/laravel/6.x/validation#named-error-bags ) 作为第二个参数传递给 @error 指令去获取验证错误内容并可以通过多种形式展示在页面上。

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

<label for="email">Email address</label>

<input id="email" type="email" class="@error('email', 'login') is-invalid @enderror">

@error('email', 'login')

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

@enderror

--

7、引入子视图

Blade 的 @include 指令允许你从其它视图中引入 Blade 视图。父视图中所有可用的变量都将在被引入的视图中可用。

<div>

@include('shared.errors')

<form>

<!-- Form Contents -->

</form>

</div>

//

被包含的视图不仅会继承父视图的所有可用数据，还能够以数组形式向被包含的视图传递额外数据。

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

//

如果传递给 @include 一个不存在的视图，Laravel 会抛出错误。想要包含一个不能确定存在与否的视图，需要使用 @includeIf 指令。

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

//

想要包含一个依赖于给定布尔条件的视图，可以使用 @includeWhen 指令。

@includeWhen($boolean, 'view.name', ['some' => 'data'])

//

要包含给定视图数组中第一个存在的视图，可以使用 includeFirst 指令。

@includeFirst(['custom.admin', 'admin'], ['some' => 'data'])

注意：应当尽量避免在 Blade 视图中使用 DIR 和 FILE 魔术常量，因为它们将指向缓存中经过编译的视图的位置。

(1)、给被包含的视图起别名

如果你的 Blade 被包含视图们存储在子目录中，你可能会希望为它们起个易于访问的别名。例如，一个带有如下内容的 Blade 视图内容被存储在 resources/views/includes/input.blade.php 文件中。

<input type="{{ $type ?? 'text' }}">

//

可以使用 include 方法为 includes.input 起一个叫做 input 的别名。通常，这会在 AppServiceProvider 的 boot 方法中完成。

use Illuminate\Support\Facades\Blade;

Blade::include('includes.input', 'input');

//

一旦被包含的视图拥有了别名，就可以像 Blade 指令一样使用别名渲染它。

@input(['type' => 'email'])

(2)、为集合渲染视图

可以使用 Blade 的 @each 指令在一行中整合循环和包含。

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

第一个参数是渲染数组或集合的每个元素的视图片段。第二个参数是希望被迭代的数组或集合，第三个参数则是将被分配给视图中当前迭代的变量名。例如，想要迭代 jobs 数组，通常会在视图片段中使用 job 变量访问每个任务。当前迭代的 key 将作为视图片段中的 key 变量。

也可以向 @each 指令传递第四个参数。这个参数是当给定数组为空时要渲染的视图片段。

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

注意：借助 @each 渲染视图，无法从父视图中继承变量。如果子视图需要这些变量，就必须使用 @foreach 和 @include 代替它。

--

8、堆栈

Blade 允许你将视图压入堆栈，这些视图能够在其它视图或布局中被渲染。这在子视图中指定需要的 JavaScript 库时非常有用。

@push('scripts')

<script src="/example.js"></script>

@endpush

//

如果需要，可以多次压入堆栈。通过向 @stack 指令传递堆栈名称来完成堆栈内容的渲染。

<head>

<!-- 头部内容 -->

@stack('scripts')

</head>

//

如果想要将内容预置在栈顶，需要使用 @prepend 指令。

@push('scripts')

This will be second...

@endpush

// 然后...

@prepend('scripts')

This will be first...

@endprepend

--

9、服务注入 (Service 注入)

@inject 指令可以用于自 Laravel 的 服务容器( https://learnku.com/docs/laravel/6.x/container ) 中获取服务。传递给 @inject 的第一个参数是将要置入的服务变量名，第二个参数是希望被解析的类或接口名。

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

<div>

Monthly Revenue: {{ $metrics->monthlyRevenue() }}.

</div>

--

10、Blade 扩展 (扩展 Blade)

Blade 允许你使用 directive 方法自定义指令。当 Blade 编译器遇到自定义指令时，这会调用该指令包含的表达式提供的回调。

//// 下面的例子创建了 @datetime($var) 指令，一个格式化给定的 DateTime 的实例 $var。

<?php

namespace App\Providers;

use Illuminate\Support\Facades\Blade;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider

{

/\*\*

\* 在容器中注册绑定.

\*

\* @return void

\*/

public function register()

{

//

}

/\*\*

\* 执行注册后引导服务.

\*

\* @return void

\*/

public function boot()

{

Blade::directive('datetime', function ($expression) {

return "<?php echo ($expression)->format('m/d/Y H:i'); ?>";

});

}

}

?>

如你所见，我们将在传递给该指令的任意表达式中链式调用 format 方法。在这个例子中，该指令将生成如下原生 PHP 代码。

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

注意：在更新 Blade 指令的逻辑之后，需要删除 Blade 视图的所有缓存。可以使用 view:clear Artisan 命令删除 Blade 视图缓存。

(1)、自定义 If 语句

在定义简单的、自定义条件语句时，编写自定义指令比必须的步骤复杂。在这种情况下，Blade 提供了 Blade::if 方法，它允许你使用闭包快速度定义条件指令。例如，定义一个校验当前应用环境的自定义指令，可以在 AppServiceProvider 的 boot 方法中这样做。

use Illuminate\Support\Facades\Blade;

/\*\*

\* 执行注册后引导服务

\*

\* @return void

\*/

public function boot()

{

Blade::if('env', function ($environment) {

return app()->environment($environment);

});

}

//

一旦定义了自定义条件指令，就可以在模板中轻松的使用。

@env('local')

// 应用在本地环境中运行...

@elseenv('testing')

// 应用在测试环境中运行...

@else

// 应用没有在本地和测试环境中运行...

@endenv

--

--

二、本地化

https://learnku.com/docs/laravel/6.x/localization/5148

--

1、简介

Laravel 本地化特性提供了从不同语言文件中检索字符串的简单方法，让您的应用更好地支持多语言开发。语言文件默认都放在 resources/lang 目录中。在此目录中，相应的语言文件存放在相应的子目录下，例如。

/resources

/lang

/en

messages.php

/es

messages.php

// 所有的语言文件都返回一个键值对数组，例如。

<?php

return [

'welcome' => 'Welcome to our application'

];

?>

(1)、区域配置

应用的默认语言设置保存在 config/app.php 配置文件中。你可以根据需要修改当前设置，还可以使用 App Facade 的 setLocale 方法动态地更改应用运行过程中使用的语言。

Route::get('welcome/{locale}', function ($locale) {

App::setLocale($locale);

//

});

//

///1/ 备用语言设置

你也可以设置『备用语言』，它会在当前语言不包含给定的翻译字符串时被使用。像默认语言一样，备用语言也可以在 config/app.php 配置文件中设置。

'fallback\_locale' => 'en',

(2)、确定当前语言环境

你可以使用 App Facade 的 getLocale 和 isLocale 方法确定当前的区域设置或者检查语言环境是否为给定值。

$locale = App::getLocale();

if (App::isLocale('en')) {

//

}

--

2、定义翻译字符串

(1)、如何使用短键 (使用短键)

通常，翻译字符串都存放在 resources/lang 目录下的文件里。在此目录中，但凡应用支持的每种语言都应该有一个对应的子目录。

/resources

/lang

/en

messages.php

/es

messages.php

// 所有语言文件都返回键值对数组，例如。

<?php

// resources/lang/en/messages.php

return [

'welcome' => 'Welcome to our application'

];

(2)、如何使用翻译字符串作为键 (使用翻译字符串作为键)

对于有大量翻译需求的应用，如果每条翻译语句都要一一使用 『短键』 来定义，那么当你在视图中尝试去引用这些 『短键』 的时候，很容易变得混乱。因此， Laravel 也支持使用字符串 『默认』 翻译作为关键字来定义翻译字符串。

使用翻译字符串作为键的翻译文件以 JSON 格式存储在 resources/lang 目录中。例如，如果你的应用中有西班牙语翻译，你应该在该目录下新建一个 resources/lang/es.json 文件。

{

"I love programming.": "Me encanta programar."

}

--

3、检索翻译字符串

你可以使用辅助函数 \_\_ 从语言文件中检索， \_\_ 函数接受翻译字符串所在的文件名加键名作为其第一个参数。例如，我们要检索 resources/lang/messages.php 语言文件中的翻译字符串 welcome。

echo \_\_('messages.welcome');

echo \_\_('I love programming.');

如果你正使用 Blade 模板引擎( https://learnku.com/docs/laravel/6.x/blade )，你可以在视图文件中使用 {{ }} 语法或者使用 @lang 指令来打印翻译字符串。

{{ \_\_('messages.welcome') }}

@lang('messages.welcome')

如果指定的翻译字符串不存在，那么 \_\_ 函数会直接返回该翻译字符串的键名。所以，如果上述示例中的翻译字符串对应的键值对不存在， \_\_ 函数将会直接返回 messages.welcome。

注意：@lang 指令不会对任何输出进行转义。当你使用这个指令时，你必须 完全由自己承担 对输出内容的转义工作。

(1)、替换翻译字符串中的参数 (翻译字符串中的参数替换)

如果需要，你可以在翻译字符串中定义占位符。所有的占位符都有一个 : 前缀。例如，你可以使用占位符 name 定义欢迎消息。

'welcome' => 'Welcome, :name',

你可以在 \_\_ 函数中传递一个数组作为第二个参数，它会将数组中的值替换到翻译字符串的占位符中。

echo \_\_('messages.welcome', ['name' => 'dayle']);

如果你的占位符中包含了首字母大写或者全部为大写，翻译过来的内容也会做相应的大写处理。

'welcome' => 'Welcome, :NAME', // Welcome, DAYLE

'goodbye' => 'Goodbye, :Name', // Goodbye, Dayle

(2)、复数

复数是一个复杂的问题，因为不同的语言对复数有不同的规则， 使用 『管道符』 | ，可以区分字符串的单复数形式。

'apples' => 'There is one apple|There are many apples',

你甚至可以创建更复杂的复数规则，为多个数字范围指定翻译字符串。

'apples' => '{0} There are none|[1,19] There are some|[20,\*] There are many',

在定义具有复数选项的翻译字符串之后，你可以使用 trans\_choice 函数来检索给定『数量』 的内容。例如，设置 『总数』 为 10 ，符合数量范围 1 至 19 ，所以会得到 There are some 这条复数语句。

echo trans\_choice('messages.apples', 10);

你也可以在复数字符串中插入占位符。 trans\_choice 函数第三个参数所传递数组将会替换占位符。

'minutes\_ago' => '{1} :value minute ago|[2,\*] :value minutes ago',

echo trans\_choice('time.minutes\_ago', 5, ['value' => 5]);

//

// :count

如果你想让传递给 trans\_choice 函数的 『数量』 参数显示在翻译字符串中，你可以使用 :count 占位符。

'apples' => '{0} There are none|{1} There is one|[2,\*] There are :count',

--

4、重写扩展包的语言文件

部分扩展包可能会附带自己的语言文件。你可以通过在 resources/lang/vendor/{package}/{locale} 目录放置文件来重写它们，而不要直接修改扩展包的核心文件。

例如，当你需要重写 skyrim/hearthfire 扩展包的英语语言文件 messages.php ，则需要把文件存放为 resources/lang/vendor/hearthfire/en/messages.php 。在这个文件中，你只需要定义你想要修改的翻译字符串。任何没有被重写的翻译字符串仍将从扩展包的原始语言文件中加载。

--

--

三、前端指南 (JavaScript & CSS 脚手架)

https://learnku.com/docs/laravel/6.x/frontend/5149

--

1、简介

虽然 Laravel 并不强制你使用哪个 JavaScript 或 CSS 预处理器，不过它提供了使用适用于许多应用场景的 Bootstrap( https://getbootstrap.com/ ) 和 Vue( https://vuejs.org/ ) 的起点。默认情况下， Laravel 使用 NPM( https://www.npmjs.org/ ) 安装这两个前端包。

//

///1/ 安装vue脚手架

Laravel 提供的引导和 vue 脚手架位于 laravel/ui composer 包中，可以使用 composer 进行安装。

composer require laravel/ui:^1.0 --dev

安装完 laravel/ui 包后，可以使用 ui artisan 命令安装前端脚手架。

// 生成基本脚手架

php artisan ui bootstrap

php artisan ui vue

php artisan ui react

// 生成 登录/注册 脚手架...

php artisan ui bootstrap --auth

php artisan ui vue --auth

php artisan ui react --auth

(1)、CSS

Laravel Mix( https://learnku.com/docs/laravel/6.x/mix ) 提供了干净的、富有表现力的 API，用于编译 SASS 和 Less，它们能够拓展原始的 CSS，拥有向 CSS 中添加变量、mixins 和其它使 CSS 更好用的强大特性。在这篇文档中，我们将简要说明 CSS 的大体编译过程；不过，你最好翻阅完整的 Laravel Mix 文档( https://learnku.com/docs/laravel/6.x/mix ) 以获取编译 SASS 和 Less 的详细信息。

(2)、JavaScript

Laravel 不要求你使用特定的 JavaScript 框架或库来构建应用。事实上，你完全可以不使用 JavaScript。但是 Laravel 包括了几个基本的脚手架，它们可以使创建基于 Vue 库( https://vuejs.org/ )的现代 JavaScript 变得更容易。Vue 提供了一个极富表现力的 API，使用组件来构建健壮的 JavaScript。像 CSS 一样，可以使用 Laravel Mix 轻松将 JavaScript 组件编译到单个的、基于浏览器的 JavaScript 文件中。

--

2、编写 CSS

在安装 laravel/ui 编写器包并生成前端脚手架( https://learnku.com/docs/laravel/6.x/frontend/5149#introduction ) 之后，Laravel 的 package.json 文件将包含 bootstrap 包，以帮助你开始使用 bootstrap 对应用程序的前端进行原型设计。但是你可以根据自己应用的需要，在 package.json 文件中，随意添加或删除依赖包。不是一定要使用 Bootstrap 框架来构建您的 Laravel 应用程序，它只是为想使用它的人提供了一个易用的起点。

//

编译 CSS 之前，请使用 Node 包管理器 (NPM https://www.npmjs.org/ ) 安装项目前端依赖。

npm install

一旦已经使用 npm install 安装了依赖包，就可以使用 Laravel Mix( https://learnku.com/docs/laravel/6.x/mix#working-with-stylesheets ) 将 SASS 编译为原生 CSS。npm run dev 命令将处理 webpack.mix.js 文件中的声明。通常编译后的 CSS 放在 public/css 目录中：npm run dev。

Laravel 前端框架包含的 webpack.mix.js 文件将编译 resources/sass/app.scss SASS 文件。 app.scss 导入 SASS 变量文件并加载 Bootstrap，它为多数应用提供了一个易用的起点。 可以自由的定制 app.scss 文件，以通过 配置 Laravel Mix( https://learnku.com/docs/laravel/6.x/mix ) 来使用你所希望的甚至完全不同的预处理器。

--

3、编写 JavaScript

应用所依赖的所有 JavaScript 包必须能在项目根目录的 package.json 文件中找到。这个文件与 composer.json 文件类似，composer.json 文件解决 PHP 的依赖关系，package.json 文件则解决 JavaScript 的依赖关系。使用 Node 包管理器 (NPM https://www.npmjs.org/ ) 安装这些依赖包：npm install。

Tip：默认情况下， Laravel 的 package.json 文件包含 vue 、 axios 等几个包，以帮助你开始构建 JavaScript 应用。可以根据应用的需要随意添加或删除 package.json 文件中的依赖。

一旦安装了这些包，就可以使用 npm run dev 命令 编译你的资源。Webpack 现代 JavaScript 应用的模块打包器。运行 npm run dev 命令时， Webpack 将执行 webpack.mix.js 文件中的指令：npm run dev。

默认情况下， Laravel 的 webpack.mix.js 文件编译 SASS 和 resources/js/app.js 文件。你可以在 app.js 文件中注册组件，如果你喜欢利用其它框架配置自己的 JavaScript 应用。编译过的 JavaScript 通常放置在 public/js 目录。

Tip： app.js 文件将载入 resources/js/bootstrap.js 文件，它负责配置和启动 Vue、 Axios、 jQuery 和其它 JavaScript 依赖。如果你想配置额外的 JavaScript 依赖，可以在这个文件内完成。

(1)、编写 Vue 组件

当使用 laravel / ui 包给你的前端搭脚手架时，ExampleComponent.vue Vue 组件将会放置在 resources/js/components 目录中。 ExampleComponent.vue 文件是一个 单文件 Vue 组件( https://vuejs.org/guide/single-file-components ) 示例，在同一个文件中定义了组件的 JavaScript 和 HTML 。单文件组件提供了构建 JavaScript 驱动应用的简便方法。这个示例组件是在 app.js 文件中注册的。

Vue.component(

'example-component',

require('./components/ExampleComponent.vue').default

);

要在应用中使用该组件，需要将其放入 HTML 模板。比如，运行 php artisan ui vue --auth Artisan 命令搭建应用的用户身份验证和注册页面的骨架，只要把这个组件放到 home.blade.php Blade 模板中。

@extends('layouts.app')

@section('content')

<example-component></example-component>

@endsection

Tip：切记，每次修改了 Vue 组件，都要运行 npm run dev 命令。也可以运行 npm run watch 命令监视和自动重编译发生变化的组件。

如果有进一步学习编写 Vue 组件的兴趣，可以阅读 Vue 文档( https://vuejs.org/guide/ )，它提供了整个 Vue 框架全面、易读的概述。

(2)、使用 React

如果更喜欢使用 React 构建 JavaScript 应用， Laravel 使得在 Vue 脚手架和 React 脚手架间切换变得非常容易。在纯净的 Laravel 应用中，可以使用带有 react 参数的 preset 命令来达到此目的。

composer require laravel/ui

php artisan ui react

// 生成 登录 / 注册 脚手架...

php artisan ui react --auth

--

--

四、编辑资源 Mix

https://learnku.com/docs/laravel/6.x/mix/5150

--

1、简介

Laravel Mix( https://github.com/JeffreyWay/laravel-mix ) 提供了定义 Webpack 构建步骤的 API，用于在应用中使用常见的 CSS 和 JavaScript 预处理器。通过链式调用这些简洁方法，可以流畅地定义资源管道。例如。

如果对如何开始使用 Webpack 和资源编译感到困惑和不知所措，那么你一定会喜欢 Laravel Mix。但是，你不一定非要使用它来开发应用；可以使用你喜欢的任何资源管道工具，甚至干脆不用。

--

2、安装 & 设置

(1)、安装 Node

在触发 Mix 之前，要先确保机器上已经安装了 Node.js 和 NPM。

node -v

npm -v

默认情况下， Laravel Homestead 包含了你需要的所有东西；不过如果你使用 Vagrant，你也能使用从 它们的下载页面( https://nodejs.org/en/download/ ) 获取的更易用的图形化安装器安装最新版本的 Node 和 NPM。

(2)、Laravel Mix

剩下的事儿就是安装 Laravel Mix。在 Laravel 的纯净安装版中，可以在目录结构的根目录中找到 package.json 文件。默认的 package.json 文件包含了起步所需的所有东西，把它想像成 composer.json 文件，只不过它是定义 Node 的依赖项，而不是 PHP 的。通过运行如下命令来安装它引用的依赖：npm install。

--

3、运行 Mix

Mix 是 Webpack( https://webpack.js.org/ ) 的顶层配置，因此运行 Mix 任务，只需要执行一条包含在 Laravel 默认的 package.json 文件中的脚本。

// 运行 Mix 任务...

npm run dev

// 运行所有的 Mix 任务并最小化输出结果...

npm run production

(1)、监听资源变化

npm run watch 命令能够在终端持续运行并且监听相关文件的变化。一旦它发现改变，Webpack 将自动重新编译资源：npm run watch。

你会发现在特定环境下，文件发生改变并不会促使 Webpack 更新。如果系统中出现这种情况，不妨考虑使用 watch-poll 命令：npm run watch-poll。

--

4、使用样式表

webpack.mix.js 文件所有资源编译的切入点。把它想像为 Webpack 的轻量级配置封装。Mix 任务可以与如何编译资源的配置一起被链式调用。

(1)、Less

less 方法可用于将 Less( http://lesscss.org/ ) 编译成 CSS。下面的语句将 app.less 文件编译为 public/css/app.css。

mix.less('resources/less/app.less', 'public/css');

//

可以通过多次调用 less 方法完成多文件编译。

mix.less('resources/less/app.less', 'public/css')

.less('resources/less/admin.less', 'public/css');

//

如果希望自定义编译后的 CSS 文件名，可以把完整的文件路径作为传递给 less 方法的第二个参数。

mix.less('resources/less/app.less', 'public/stylesheets/styles.css');

//

如果需要覆盖 隐含的 Less 插件选项( https://github.com/webpack-contrib/less-loader#options )，可以传递一个对象作为 mix.less() 的第三个参数。

mix.less('resources/less/app.less', 'public/css', {

strictMath: true

});

(2)、Sass

sass 方法能将 Sass( https://sass-lang.com/ ) 编译成 CSS。可以像下面这样使用这个方法。

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

//

和 less 方法类似，可以将多个 Sass 文件编译为各自的 CSS 文件，并为结果 CSS 自定义输出目录。

mix.sass('resources/sass/app.sass', 'public/css')

.sass('resources/sass/admin.sass', 'public/css/admin');

//

额外的 Node-Sass 插件选项( https://github.com/sass/node-sass#options ) 可以作为第三个参数。

mix.sass('resources/sass/app.sass', 'public/css', {

precision: 5

});

(3)、Stylus

与 Less 和 Sass 相似， stylus 方法将 Stylus( http://stylus-lang.com/ ) 编译成 CSS。

mix.stylus('resources/stylus/app.styl', 'public/css');

还可以安装额外的 Stylus 插件，比如 Rupture( https://github.com/jescalan/rupture )。先通过 NPM （npm install rupture）安装有插件，然后在调用 mix.stylus() 时包含它。

mix.stylus('resources/stylus/app.styl', 'public/css', {

use: [

require('rupture')()

]

});

(4)、PostCSS

PostCSS( https://postcss.org/ ) 是一个转换 CSS 的强力工具，它已经包含在 Laravel Mix 中。默认情况下，Mix 利用流行的 Autoprefixer( https://github.com/postcss/autoprefixer ) 插件自动附加所有必要的 CSS3 第三方前缀。不过，你可以自由添加应用中所需的任何附加插件。首先通过 NPM 安装所需插件，然后在 webpack.mix.js 文件引用。

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

.options({

postCss: [

require('postcss-css-variables')()

]

});

(5)、Plain CSS (原生 CSS)

如果想要将几个原生 CSS 样式表合并到单个文件中，可以使用 styles 方法。

mix.styles([

'public/css/vendor/normalize.css',

'public/css/vendor/videojs.css'

], 'public/css/all.css');

(6)、URL 处理

因为是基于 Webpack 之上构建，了解几个 Webpack 概念就很重要了。对于 CSS 编译，Webpack 将重写和优化任何带有样式表的 url() 调用。虽然初听上去好像很奇怪，但这确实是个强大的功能。 想像一下我们想要编译包含图片相对 URL 的 Sass。

.example {

background: url('../images/example.png');

}

注意：任何给定 url() 的绝对路径将被排除在 URL 重写之外。例如 url('/images/thing.png') 或 url('http://example.com/images/thing.png') 将不会被修改。

默认情况下，Laravel Mix 和 Webpack 将找到 example.png, 将其复制到 public/images 文件夹，然后重写生成的样式表中的 url()。如此一来，编译后的 CSS 将变成。

.example {

background: url(/images/example.png?d41d8cd98f00b204e9800998ecf8427e);

}

//

尽管此功能可能很有用，但是已有文件夹结构可能已经按你的预期做了配置。这种情况下，你可以禁用 url() 重写。

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

.options({

processCssUrls: false

});

在 webpack.mix.js 文件中加入这项配置，Mix 将不再匹配任何 url() 或者复制资源到 public 目录。换句话说，编译后的 CSS 看上去和你原来输入的内容一样。

.example {

background: url("../images/thing.png");

}

(7)、资源映射 (源码映射)

虽然默认情况下源码映射被禁用，但是可以通过在 webpack.mix.js 文件中调用 mix.sourceMaps() 方法来激活。尽管会提高编译 / 执行的成本，却可以在在编译资源时为浏览器开发工具提供额外的调试信息。

mix.js('resources/js/app.js', 'public/js')

.sourceMaps();

--

5、使用 JavaScript

Mix 提供了几个有助于使用 JavaScript 文件的特性，比如编译 ECMAScript 2015、模块打包、最小化和合并原生 JavaScript 文件。更棒的是，所有这些都无需配置就可以很好地工作。

mix.js('resources/js/app.js', 'public/js');

只需这一行代码，就可以支持：ES2015 语法；模块；编译 .vue 文件；为生产环境最小化代码。

(1)、Vendor 提取

将应用自身的 JavaScript 和第三方库绑定在一起有个潜在的缺点：会使长期缓存变得困难。例如，应用代码的单独更新将强制浏览器重新下载全部第三方库，即使第三方库没有发生变化。

//

如果你打算频繁地更新应用中的 JavaScript，应该考虑将第三方库提取到它自己的文件中。这样一来，应用代码的改变将不影响 vendor.js 文件缓存。Mix 的 extract 方法为之提供了便利。

mix.js('resources/js/app.js', 'public/js')

.extract(['vue'])

extract 方法接受你想要提取到 vendor.js 的全部库或模块的数组作为参数。使用上面例子中的代码片段，Mix 将生成如下文件。

public/js/manifest.js: Webpack 运行时清单；

public/js/vendor.js: 第三方库代码；

public/js/app.js: 应用代码；

//

为避免 JavaScript 错误，应确保按适应顺序加载这些文件。

<script src="/js/manifest.js"></script>

<script src="/js/vendor.js"></script>

<script src="/js/app.js"></script>

(2)、React

Mix 能自动安装 React 支持所需的 Babel 插件。要达成此目的，只需要用 mix.react() 替换 mix.js()。

mix.react('resources/js/app.jsx', 'public/js');

Mix 将在后台下载并包含适用的 babel-preset-react Babel 插件。

(3)、Vanilla JS

与利用 mix.styles() 整合样式表类似，你也可以使用 scripts() 方法整合并最小化任意数量的 JavaScript 文件

mix.scripts([

'public/js/admin.js',

'public/js/dashboard.js'

], 'public/js/all.js');

对于不需要使用 Webpack 编译 JavaScript 的遗留代码，这个选项尤其有用。

Tip： mix.babel() 是 mix.scripts() 的一个小变种。它的方法签名与 scripts 完全相同；但整合的文件将经由 Babel 编译，它会将 ES2015 代码转换为浏览器能够理解的 vanilla JavaScript。

(4)、自定义 Webpack 配置

Lavarel Mix 在后台引用预先配置的 webpack.config.js 文件以加速启动和运行。有时，你可能需要手动编辑这个文件。你可能有一个特殊的加载器或插件需要引用，或者可能更愿意用 Stylus 代替 Sass。这种情况下，你有两个选择。

//

///1/ 选项一：合并自定义配

Mix 提供了 webpackConfig 方法来合并任意简短的 Webpack 配置以覆盖已有配置。这是个非常勾人的选择，因为它不要求你复制和维护你自己的 webpack.config.js 副本。 webpackConfig 方法接受一个对象作为作参数，它包含你想要应用的任何 Webpack 特定配置( https://webpack.js.org/configuration/ )。

mix.webpackConfig({

resolve: {

modules: [

path.resolve(\_\_dirname, 'vendor/laravel/spark/resources/assets/js')

]

}

});

//

///2/ 选项二：自定义配置文件

如果你想完全自定义 Webpack 配置，将 node\_modules/laravel-mix/setup/webpack.config.js 文件复制到项目的根目录。然后将 package.json 文件中所有的 --config 引用都指向新复制的配置文件。如果选择用这种方式自定义配置，Mix webpack.config.js 后续的任何更新，都必须手动合并到你的自定义配置文件中。

--

6、复制文件 & 目录

copy 方法可以用于将文件和目录复制到新位置。当 node\_modules 目录中特殊的资源需要重新定位到 public 文件夹时，这个方法很有用。

mix.copy('node\_modules/foo/bar.css', 'public/css/bar.css');

//

复制文件夹时， copy 方法将扁平化目录结构。想要保持目录的原始结构，需要使用 copyDirectory 方法。

mix.copyDirectory('resources/img', 'public/img');

--

7、版本控制 / 缓存清除

许多开发者在他们编译后的资源添加时间戳或唯一令牌作后缀，强制浏览器加载新的资源，以替换旧的代码副本。Mix 可以使用 version 方法替你处理它们。

version 方法自动在所有编译后的文件名后追加唯一的哈希值，从而实现更方便的缓存销毁。

mix.js('resources/js/app.js', 'public/js')

.version();

//

在生成版本化文件后，你不会知道确切的文件名。因此，你需要在 views( https://learnku.com/docs/laravel/6.x/views ) 中使用 Laravel 的全局 mix 函数载入相应的哈希资源。 mix 函数自动判断哈希文件的当前文件名。

<script src="{{ mix('/js/app.js') }}"></script>

通常在开发阶段不需要版本化文件，你可以仅在运行 npm run production 时执行版本化处理。

mix.js('resources/js/app.js', 'public/js');

if (mix.inProduction()) {

mix.version();

}

--

8、Browsersync 重新加载

BrowserSync( https://browsersync.io/ ) 能够自动监测文件变化，并且无需手动刷新就将变化注入到浏览器。可以调用 mix.browserSync() 方法开启此项支持。

mix.browserSync('my-domain.test');

// 或...

// https://browsersync.io/docs/options

mix.browserSync({

proxy: 'my-domain.test'

});

可以传递字符串（代理）或对象 Y（BrowserSync 设置）给此方法。然后使用 npm run watch 命令启动 Webpack 的开发服务器。再编辑脚本或者 PHP 文件，就会看到浏览器立即刷新以响应你的修改。

--

9、环境变量

可以通过在 .env 文件中添加 MIX\_ 前缀，将环境变量注入到 Mix。

MIX\_SENTRY\_DSN\_PUBLIC=http://example.com

一旦此变量被定义在 .env 文件中，就可以借助 process.env 对象访问它。如果在运行 watch 任务时这个值发生改变，就需要重启任务。

process.env.MIX\_SENTRY\_DSN\_PUBLIC

--

10、通知

如果可用，Mix 将自动为每次绑定显示操作系统通知。这将为你提供编译是否成功的实时反馈。不过，可能在某些情况下，你希望禁用通知。一个例子是在生产服务器上触发 Mix。通知可以借助 disableNotifications 方法来禁用。

mix.disableNotifications();

--

--

第六卷 安全相关

--

--

一、用户认证

https://learnku.com/docs/laravel/6.x/authentication/5151

--

1、介绍

提示： 想要快速开始吗？ Composer 安装 laravel/ui 包并在一个全新的 Laravel 应用程序中运行 php artisan ui vue --auth 。迁移数据库后，将浏览器导航到 http://your-app.test/register 或分配给应用程序的任何其他 URL。这两个命令将负责构建整个认证系统！

//

///1/ 构建整个认证系统(vue)( 可以是bootstrap、react )

// Composer 安装 laravel/ui 包

// php artisan ui vue --auth

//

Laravel 使得实现身份验证非常简单。 事实上，几乎所有的配置都是现成的。 身份验证配置文件位于 config/auth.php， 其中包含几个有良好文档记录的选项，用于调整身份验证服务的行为。

在其核心，Laravel 的认证设施由「守卫」和「提供者」组成。守卫决定如何对每个请求的用户进行身份验证。比如，Laravel 带有一个 session 保护，它使用会话存储和 Cookies 来维护状态。

提供者决定如何从持久储存中检索用户。 Laravel 支持使用 Eloquent 和数据库查询生成器检索用户。但是，你可以根据应用程序的需要来自由定义其他提供者。

如果这些听起来让你有些疑惑，不要担心！许多应用程序永远不需要修改默认的身份验证配置。

(1)、数据库注意事项

默认情况下， Laravel 包含一个 App\User Eloquent model( https://learnku.com/docs/laravel/6.x/eloquent ) 在你的 app 目录下。 这个模型可与默认的 Eloquent 身份验证驱动程序一起使用。如果你的应用程序没有使用 Eloquent，你可以用 database 身份验证驱动程序，它用的是 Laravel 查询生成器。

当为 App\User 模型生成数据库架构时，确保密码的长度至少为 60 个字符。保持默认的字符串长度为 255 个字符是一个不错的选择。

另外，你应该验证 users（或等效）表是否包含一个可空的，含有 100 个字符的 remember\_token 字符串。此列将用于存储用户登录应用程序时选择「记住我」选项的令牌。

--

2、用户认证快速指南

Laravel 附带了几个预构建的身份验证控制器，它们位于 App\Http\Controllers\Auth 命名空间中。RegisterController 处理新的用户注册，LoginController 处理身份验证，ForgotPasswordController 处理用于重置密码的电子邮件链接，ResetPasswordController 包含重置密码的逻辑。这些控制器中的每一个都使用一个特性来包含它们的必要方法。对于许多应用程序，你根本不需要修改这些控制器。

(1)、路由

Laravel 的 laravel/ui 包提供了一种快速方法，可以使用一些简单的命令来支持你进行身份验证所需的所有路由和视图。

// laravel/ui包

// composer require laravel/ui (6.18.14，安装报错)

composer require laravel/ui:^1.0 --dev (^2.0安装报错，只能执行^1.0版本)

/// 可选择bootstrap/vue/react其中之一

// bootstrap

php artisan ui bootstrap --auth

npm install && npm run dev //安装编译资源(Please run "npm install && npm run dev" to compile your fresh scaffolding.);

//

// veu

php artisan ui vue --auth

//

// react

php artisan ui react --auth

//

这个命令应该在新应用程序上使用，并将安装布局视图，注册和登录视图以及所有身份验证端点的路由。 还将生成一个 HomeController 来处理登录后跳转到 Home 页面。

提示：如果应用程序不需要注册，可以通过删除新创建的 RegisterController 并修改路由声明来禁用它：Auth::routes(['register' => false]);。

(2)、视图

如上一节所述，laravel/ui 包的 php artisan ui vue --auth 命令将创建验证所需的所有视图，并将它们放在 resources/views/auth 目录下。

ui 命令还将创建一个 resources/views/layouts 目录，其中包含应用程序的基本布局。 这些视图都使用 Bootstrap CSS 框架，但是你可以自由地定制它们。

(3)、认证

现在你已经给身份验证控制器设置了路由和视图，你可以为应用程序注册和验证新用户了！ 因为控制器已经默认包含了验证用户是否存在和保存用户到数据库中的认证逻辑（通过 traits 实现的），现在你已经可以在浏览器中访问应用了。

//

///1/ 自定义路径

当用户认证成功，他们会被重定向到 /home 这个 URI 下。你可以在 RouteServiceProvider 中修改常量 HOME 的来定义认证成功后的重定向路径。

public const HOME = '/home';

如果你想要更加灵活的自定义用户验证通过后的 HTTP 响应，你可以重写 trait AuthenticatesUsers 的 authenticated(Request $request, $user) 方法。

/\*\*

\* The user has been authenticated.

\*

\* @param \Illuminate\Http\Request $request

\* @param mixed $user

\* @return mixed

\*/

protected function authenticated(Request $request, $user)

{

return redirect('/');

}

//

///2/ 自定义用户名

Laravel 默认使用 email 字段来认证。如果你想使用其他的字段，可以在 LoginController 控制器里面定义一个 username 方法。

public function username()

{

return 'username';

}

//

///3/ 自定义看守器

你还可以自定义用户认证和注册的「看守器」。要实现这一功能，需要在 LoginController，RegisterController 和 ResetPasswordController 中定义 guard 方法。该方法会返回一个看守器实例。

use Illuminate\Support\Facades\Auth;

protected function guard()

{

return Auth::guard('guard-name');

}

//

///4/ 自定义验证/存储

为了修改新用户在注册时所需要填写的表单字段，或者自定义如何将新用户存储到数据库中，你可以修改 RegisterController 类。在你的应用程序中，该类负责验证和创建新用户。

RegisterController 类的 validator 方法包含了验证新用户的规则，你可以随心所欲地自定义该方法。

RegisterController 的 create 方法负责使用 Eloquent ORM( https://learnku.com/docs/laravel/6.x/eloquent ) 在数据库中创建新的 App\User 记录。你可以根据数据库的需要自定义该方法。

(4)、检索已验证的用户 (检索认证用户)

你可以通过 Auth facade 来访问已认证的用户。

use Illuminate\Support\Facades\Auth;

// 获取当前通过认证的用户...

$user = Auth::user();

// 获取当前通过认证的用户 ID...

$id = Auth::id();

//

或者，你可以通过 Illuminate\Http\Request 实例来访问已认证的用户。别忘了，类型提示的类会被自动注入到你的控制器方法中。

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class ProfileController extends Controller

{

/\*\*

\* 更新用户资料

\*

\* @param Request $request

\* @return Response

\*/

public function update(Request $request)

{

// $request->user() 返回一个认证用户实例...

}

}

?>

//

///1/ 确定当前用户是否已经认证

你可以使用 Auth facade 的 check 方法来检查用户是否已认证。如果已认证，将会返回 true。

use Illuminate\Support\Facades\Auth;

if (Auth::check()) {

// 用户已经登录了...

}

提示：虽然可以使用 check 方法确认用户是否被认证，但是在允许用户访问的某些路由 / 控制器之前，通常还是会使用中间件来验证用户是否进行过身份验证。想要了解更多信息，请查看有关 保护路由( https://learnku.com/docs/laravel/6.x/authentication#protecting-routes ) 的文档。

(5)、路由保护

路由中间件( https://learnku.com/docs/laravel/6.x/middleware ) 可以用于只允许通过认证的用户访问给定的路由。Laravel 自带了一个 auth 中间件，它定义在 Illuminate\Auth\Middleware\Authenticate 中。由于这个中间件已经在 HTTP 内核中注册，你只需把这个中间件附加到路由定义中。

Route::get('profile', function () {

// 只有认证过的用户可以进入...

})->middleware('auth');

当然，如果你使用 控制器( https://learnku.com/docs/laravel/6.x/controllers )，你可以在控制器的构造函数中调用 middleware 方法来直接将其附加到路由定义中。

public function \_\_construct()

{

$this->middleware('auth');

}

//

///2/ 重定向未认证的用户

当 auth 中间件检测到一个未认证用户时，它会把用户重定向到名为 login 的 命名路由( https://learnku.com/docs/laravel/6.x/routing#named-routes )上。

您可以通过修改 app/Http/Middleware/Authenticate.php 文件中的 redirectTo 函数来修改此行为。

/\*\*

\* Get the path the user should be redirected to.

\*

\* @param \Illuminate\Http\Request $request

\* @return string

\*/

protected function redirectTo($request)

{

return route('login');

}

//

///3/ 指定看守器

当你把 auth 中间件添加到路由中时，同时也能指定使用哪个看守器进行用户认证。指定的看守器应该对应 auth.php 配置文件中 guards 数组中的的一个键。

public function \_\_construct()

{

$this->middleware('auth:api');

}

(6)、密码确认

有时候你希望当用户访问某些特定敏感页面时候，需要重新输入密码进行确认。比如当用户访问账单设置页面时。

为实现这个目的，Laravel 提供了 password.confirm 中间件。将 password.confirm 中间件关联到一个路由上，当用户访问该路由时就会自动跳转到一个密码确认页面。

Route::get('/settings/security', function () {

// Users must confirm their password before continuing...

})->middleware(['auth', 'password.confirm']);

当用户成功完成密码确认后，将会被重定向到原先要访问的路由。默认情况下密码确认之后的 3 小时内再访问敏感页面是无需重复确认的。你可以通过 auth.password\_timeout 配置项来修改这个有效时间。

(7)、登录限制 (登录限流)

如果你使用 Laravel 内置的 LoginController 类，Illuminate\Foundation\Auth\ThrottlesLogins trait 已经包含在该控制器中了。 默认情况下，如果用户多次尝试却无法提供正确的登录凭据，那么该用户在一分钟内将不能再次尝试登录。这种限流策略基于用户的用户名 / 邮箱地址及其 IP 地址的唯一性。

--

3、手动验证用户

不一定非要在 Lavarel 中使用验证控制器。如果选择删除这些控制器，就需要直接使用 Lavarel 验证类。别担心，很容易！

可以借助 Auth facade( https://learnku.com/docs/laravel/5.8/facades ) 访问 Laravel 服务，因此需要在类的开头导入 Auth 。下面来看看 attempt 方法。

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use Illuminate\Support\Facades\Auth;

class LoginController extends Controller

{

/\*\*

\* Handle an authentication attempt.

\*

\* @param \Illuminate\Http\Request $request

\*

\* @return Response

\*/

public function authenticate(Request $request)

{

$credentials = $request->only('email', 'password');

if (Auth::attempt($credentials)) {

// 通过认证..

return redirect()->intended('dashboard');

}

}

}

?>

attempt 方法的每个参数是一个关联数组。数组值用于在数据库中查找用户。在上面的例子中，将通过 email 列的值查找用户。如果找到该用户，将用存储在数据库中的哈希密码与数组中的 password 值做比较。不需要对 password 做哈希运算，框架在与数据库中的哈希密码做比较前自动对此值做哈希运算。如果两个哈希值匹配，将为该用户建立验证通过的 session。

如果验证成功， attempt 方法返回 true ，否则返回 false。

重定向中的 intended 方法将经由身份验证中间件将用户重定向到身份验证前截获的 URL 。如果预期目标不存在，可以为此方法指定一个回退 URI。

//

///1/ 指定额外条件

除了用户的电子邮件和密码之外，还可以向身份验证查询添加其他条件。例如， 我们可以验证用户是不是已经被标记为「激活」。

if (Auth::attempt(['email' => $email, 'password' => $password, 'active' => 1])) {

//验证处于激活状态，并且存在的用户

}

注意：在这些例子中， email 不是必须的选项，它只用来做示范。你应该使用与你的数据库中 「用户名」 对应的列名。

//

///2/ 访问指定的看守器实例

可以使用 Auth facade 的 guard 方法指定想要使用的看守器实例。这允许你使用完全独立的可验证模型或用户表来管理应用程序各个部分的验证。

传递给 guard 方法的看守器名称需要与 auth.php 配置中的配置项之一相匹配。

if (Auth::guard('admin')->attempt($credentials)) {

//

}

//

///3/ 登出

用户登出需要使用 Auth facade 的 logout 方法。它会清除用户会话（session）中的用户验证信息。

Auth::logout();

(1)、记住用户

如果想在应用中提供 「记住我」功能，可以给 attempt 方法传递一个布尔值作为其第二个参数，这会无限期保持用户身份验证，直到用户手动登出。用户表需要包含字符串类型的 remember\_token 列用于存储令牌。

if (Auth::attempt(['email' => $email, 'password' => $password], $remember)) {

// 用户已经被记住...

}

Tip：如果使用了 Laravel 内置的 LoginController，「记住」用户的正确逻辑已经由控制器所用的 traits 实现。

如果启用了「记住用户」，可以使用 viaRemember 方法判断是否使用了「记住我」cookie 对用户做身份验证。

if (Auth::viaRemember()) {

//

}

(2)、其他认证方法 (其它身份验证方法)

//

///1/ 验证用户实例

如果要将已经存在的用户登入应用，可以调用 login 方法，并以用户实例作为其参数 。该对象必须实现 Illuminate\Contracts\Auth\Authenticatable 契约( https://learnku.com/docs/laravel/6.x/contracts )。Laravel 自带的 App\User 模型已经实现了这个接口。

Auth::login($user);

Auth::login($user, true); //登录并记住给定用户;

//

使用如下方式指定想要的看守器实例。

Auth::guard('admin')->login($user);

//

///2/ 通过 ID 验证用户身份

可以使用 loginUsingId 方法通过 ID 将用户登录到应用。这个方法接受希望验证身份用户的主键。

Auth::loginUsingId(1);

Auth::loginUsingId(1, true); //登录并记住给定用户;

//

///3/ 仅验证一次用户身份

可以使用 once 方法在单次请求中将用户登录到应用中。这样做将不使用 session 或 cookies，这意味着此方法有助于构建一个无状态 API。

if (Auth::once($credentials)) {

//

}

--

4、HTTP 基本认证 (HTTP 基础认证)

HTTP 基础认证( https://en.wikipedia.org/wiki/Basic\_access\_authentication ) 提供了一种快速方法来验证你应用程序中的用户，而无需设置专用的「登录」页面。 开始之前， 先把 auth.basic 中间件 附加到你的路由中。auth.basic 中间件已包含在 Laravel 框架中，所以你不需要定义它。

Route::get('profile', function () {

// 只有经过身份验证的用户才能进...

})->middleware('auth.basic');

将中间件附加到路由后，在浏览器中访问此路由时将自动提示您输入凭据。默认的，auth.basic 中间件把用户记录上的 email 字段 作为「用户名」。

//

///1/ FastCGI的注意事项

如果你正使用 PHP FastCGI 模式，HTTP 基础认证可能无法正常工作。需要把下面几行添加到你的 .htaccess 文件中。

RewriteCond %{HTTP:Authorization} ^(.+)$

RewriteRule .\* - [E=HTTP\_AUTHORIZATION:%{HTTP:Authorization}]

(1)、无状态 HTTP 基本认证

你也可以使用 HTTP 基础身份验证，而无需在会话中设置用户标识符 cookie，这对 API 的身份验证特别有用。为此 ，请 定义一个中间件( https://learnku.com/docs/laravel/6.x/middleware ) 它将调用 onceBasic 方法。如果 onceBasic 方法没有返回任何响应，那么请求就可以进一步传递到应用程序中。

<?php

namespace App\Http\Middleware;

use Illuminate\Support\Facades\Auth;

class AuthenticateOnceWithBasicAuth

{

/\*\*

\* 处理传入的请求。

\*

\* @param \Illuminate\Http\Request $request

\* @param \Closure $next

\* @return mixed

\*/

public function handle($request, $next)

{

return Auth::onceBasic() ?: $next($request);

}

}

?>

接着，注册路由中间件( https://learnku.com/docs/laravel/5.8/middleware#registering-middleware ) 并将它附加到路由。

Route::get('api/user', function () {

// 只有经过身份验证的用户才能进...

})->middleware('auth.basic.once');

--

5、登出 (退出)

要手动把用户从应用中退出登录，你可以使用 Auth facade 上的 logout 方法。这将清除用户会话中的身份认证信息。

use Illuminate\Support\Facades\Auth;

Auth::logout();

(1)、其他设备的会话无效 (让其它设备上的 Session 失效)

Laravel 还提供了一种机制，用于将其它设备上的用户 Session 失效和「注销」，而不会使其当前设备上的 Session 失效。首先，你需要保证 Illuminate\Session\Middleware\AuthenticateSession 中间件在你的 app/Http/Kernel.php 类中的 web 中间件组中，并且没有被注释掉。

'web' => [

// ...

\Illuminate\Session\Middleware\AuthenticateSession::class,

// ...

],

然后， 你就可以使用 Auth facade 上的 logoutOtherDevices 方法。此方法要求用户提供其当前密码，你的应用程序应通过输入表单接受该密码。

use Illuminate\Support\Facades\Auth;

Auth::logoutOtherDevices($password);

注意：当调用 logoutOtherDevices 方法后，用户的其它 Session 将完全失效，这意味着他们将「退出」他们之前通过身份认证的所有看守器。

--

6、社会认证

https://github.com/laravel/socialite

--

7、添加自定义保护 (添加自定义的看守器)

你可以使用 Auth facade 的 extend 方法来定义自己的身份验证看守器。你应该在 服务提供器( https://learnku.com/docs/laravel/6.x/providers ) 中调用 extend 方法。由于 Laravel 已经附带了 AuthServiceProvider，我们可以将代码放在该提供器中。

<?php

namespace App\Providers;

use App\Services\Auth\JwtGuard;

use Illuminate\Support\Facades\Auth;

use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

class AuthServiceProvider extends ServiceProvider

{

/\*\*

\* 注册任何应用程序验证/授权服务。

\*

\* @return void

\*/

public function boot()

{

$this->registerPolicies();

Auth::extend('jwt', function ($app, $name, array $config) {

// 返回一个 Illuminate\Contracts\Auth\Guard 实例...

return new JwtGuard(Auth::createUserProvider($config['provider']));

});

}

}

?>

正如你在上面的示例中所看到的，传递给 extend 方法的回调应该返回一个实现 Illuminate\Contracts\Auth\Guard 接口的实例。这个接口包含了一些你需要在自定义的看守器中实现的方法。当你的自定义看守器定义完成之后，你可以在 auth.php 配置文件的 guards 配置中使用这个看守器。

'guards' => [

'api' => [

'driver' => 'jwt',

'provider' => 'users',

],

],

--

8、闭包看守器 (请求闭包看守器)

实现基于 HTTP 请求的自定义身份验证系统的最简单方法，是使用 Auth::viaRequest 方法。此方法允许您使用单个闭包来快速定义身份验证过程。

首先，在 AuthServiceProvider 的 boot 方法中调用 Auth::viaRequest 方法。viaRequest 方法接受一个看守器名称作为其第一个参数。此名称可以是描述你自定义看守器的任何字符串。传递给该方法的第二个参数应该是一个闭包函数，它接收传入的 HTTP 请求并返回一个用户实例，或者，如果验证失败，则为 null。

use App\User;

use Illuminate\Http\Request;

use Illuminate\Support\Facades\Auth;

/\*\*

\* 注册任意应用认证／授权服务。

\*

\* @return void

\*/

public function boot()

{

$this->registerPolicies();

Auth::viaRequest('custom-token', function ($request) {

return User::where('token', $request->token)->first();

});

}

当你完成了自定义看守器后，就可以在 auth.php 配置文件的 guards 配置中使用这个看守器。

'guards' => [

'api' => [

'driver' => 'custom-token',

],

],

--

9、添加自定义用户提供程序 (添加自定义用户提供器)

如果不使用传统的关系数据库存储用户，就需要使用自己的身份验证用户提供器扩展 Lavarel。可以使用 Auth facade 的 provider 方法自定义用户提供器。

<?php

namespace App\Providers;

use Illuminate\Support\Facades\Auth;

use App\Extensions\RiakUserProvider;

use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

class AuthServiceProvider extends ServiceProvider

{

/\*\*

\* 注册任意应用身份验证 / 授权服务

\*

\* @return void

\*/

public function boot()

{

$this->registerPolicies();

Auth::provider('riak', function ($app, array $config) {

// 返回 Illuminate\Contracts\Auth\UserProvider... 实例

return new RiakUserProvider($app->make('riak.connection'));

});

}

}

?>

一旦使用 provider 方法注册完毕，就可以在 auth.php 配置文件中切换到新的用户提供器。先定义一个使用新驱动的 provider。

'providers' => [

'users' => [

'driver' => 'riak',

],

],

随后就可以在 guards 配置中使用这个提供器。

'guards' => [

'web' => [

'driver' => 'session',

'provider' => 'users',

],

],

(1)、用户提供程序合同 (用户提供器契约)

Illuminate\Contracts\Auth\UserProvider 实现仅负责从 MySQL、Riak 等持久化存储系统中提取 Illuminate\Contracts\Auth\Authenticatable 实现。无论用户如何存储及用于表示它的类是什么类型，这两个接口都允许 Laravel 身份验证机制继续运行。

我们来看看 Illuminate\Contracts\Auth\UserProvider 契约。

<?php

namespace Illuminate\Contracts\Auth;

interface UserProvider {

public function retrieveById($identifier);

public function retrieveByToken($identifier, $token);

public function updateRememberToken(Authenticatable $user, $token);

public function retrieveByCredentials(array $credentials);

public function validateCredentials(Authenticatable $user, array $credentials);

}

?>

retrieveById 函数通常接受用于表示类的 key ，如 MySQL 数据库中自动递增的 ID 作为参数，并获取和返回与这个 ID 匹配的 Authenticatable 实现。

retrieveByToken 函数通过用户的唯一 $identifier 和存储在 remember\_token 列的 「记住我」 令牌获取用户。与前一方法相同，它返回 Authenticatable 实现。

updateRememberToken 方法用新 $token 更新 $user 的 remember\_token 列。在「记住我」登录校验成功或者用户登出时分配「刷新令牌」。

在尝试登录到应用时，retrieveByCredentials 方法接受凭证数组传递给 Auth::attempt 方法。此方法在底层持久化存储中「查询」与这些凭证匹配的用户。通常，此方法运行一个基于 $credentials['username'] 的 「where」 条件，它应该返回一个 Authenticatable 实现。此方法不就尝试进行任何密码校验或身份验证。

validateCredentials 方法应该比较给定的 $user 与 $credentials 来验证用户身份。例如，此方法或许应该使用 Hash::check 来比较 $user->getAuthPassword() 的值与 $credentials['password'] 的值。它应该返回 true 或 false ，以表明用户密码是否有效。

(2)、可认证的合同 (身份验证契约)

我们已经剖析了 UserProvider 的每个方法。下面再来看看 Authenticatable 契约。切记，用户提供器的 retrieveById、 retrieveByToken 和 retrieveByCredentials 方法将返回此接口的实例。

<?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 方法应该返回用户 「主键」 列的名字， getAuthIdentifier 方法则返回用户 「主键」。在 MySQL 后台，它会是自增主键。 getAuthPassword 方法应该返回用户的哈希密码。此接口允许身份验证系统与任一 User 类一直工作，不管使用的是哪种 ORM 或抽象存储层。默认情况下，Laravel 的 app 目录会包含一个实现了此接口的 User 类，你可以以这个实现示例作为参考。

--

10、事件

在身份验证处理过程中 Laravel 引发了多种 事件( https://learnku.com/docs/laravel/6.0/events )。 可以在 EventServiceProvider 中附着这些事件的监听器。

/\*\*

\* 应用的事件监听器映射。

\*

\* @var array

\*/

protected $listen = [

'Illuminate\Auth\Events\Registered' => [

'App\Listeners\LogRegisteredUser',

],

'Illuminate\Auth\Events\Attempting' => [

'App\Listeners\LogAuthenticationAttempt',

],

'Illuminate\Auth\Events\Authenticated' => [

'App\Listeners\LogAuthenticated',

],

'Illuminate\Auth\Events\Login' => [

'App\Listeners\LogSuccessfulLogin',

],

'Illuminate\Auth\Events\Failed' => [

'App\Listeners\LogFailedLogin',

],

'Illuminate\Auth\Events\Logout' => [

'App\Listeners\LogSuccessfulLogout',

],

'Illuminate\Auth\Events\Lockout' => [

'App\Listeners\LogLockout',

],

'Illuminate\Auth\Events\PasswordReset' => [

'App\Listeners\LogPasswordReset',

],

];

--

--

二、API 认证

https://learnku.com/docs/laravel/6.x/api-authentication/5429

--

1、简介

默认情况下，Laravel 为 API 认证提供了一个简单的解决方案，它通过一个随机令牌分配给应用程序的每个用户。在你的 config/auth.php 配置文件中，已经定义了一个使用 token 驱动的 api 看守器。 这个驱动程序负责检查传入请求上的 API 令牌，并验证它是否匹配数据库中用户分配的令牌。

注意: 虽然 Laravel 附带了一个简单的基于令牌的身份验证保护，但我们强烈建议您考虑 Laravel Passport( https://learnku.com/docs/laravel/6.x/passport ) 来实现提供 API 身份验证的健壮的生产应用程序。

--

2、配置

(1)、数据库迁移 (数据库准备)

在使用 token 驱动程序之前，你需要 创建一个迁移( https://learnku.com/docs/laravel/6.x/migrations ) 它会在你的 users 表中添加一个 api\_token 列。

//

///1/ 添加api\_token列

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

$table->string('api\_token', 80)->after('password')

->unique()

->nullable()

->default(null);

});

迁移创建后，运行 migrate Artisan 命令。

Tip：如果你选择使用不同的字段名，请确保更新了配置文件 config/auth.php 中的 storage\_key 配置项。

--

3、令牌生成

将 api\_token 列添加到你的 users 表之后，你可以将随机 API 令牌分配给应用程序中的每个用户。 在注册期间创建 User 模型时，应该分配这些令牌。 如果你使用的是 laravel/ui 拓展包提供的 认证脚手架( https://learnku.com/docs/laravel/6.x/authentication#authentication-quickstart )，该操作可以在 RegisterController 控制器的 create 方法中完成。

use Illuminate\Support\Facades\Hash;

use Illuminate\Support\Str;

/\*\*

\* 在有效注册之后创建一个新用户实例

\*

\* @param array $data

\* @return \App\User

\*/

protected function create(array $data)

{

return User::create([

'name' => $data['name'],

'email' => $data['email'],

'password' => Hash::make($data['password']),

'api\_token' => Str::random(60),

]);

}

(1)、哈希令牌

在上面的示例中，API 令牌以纯文本形式存储在数据库中。 如果你想使用 SHA-256 哈希算法来散列你的 API 令牌，你可以将 api 配置的 hash 选项设置为 true 。 api 在你的 config/auth.php 配置文件中定义。

'api' => [

'driver' => 'token',

'provider' => 'users',

'hash' => true,

],

//

///1/ 生成哈希标记

使用散列 API 令牌时，你不应在用户注册期间生成 API 令牌。 取而代之，你需要在应用程序中实现自己的 API 令牌管理页面。 这个页面应允许用户初始化和刷新其 API 令牌。 当用户发出初始化或刷新其令牌的请求时，你应该在数据库中存储令牌的哈希副本，并将令牌的纯文本副本返回到视图 / 前端客户端以进行一次性显示。

例如，为给定用户初始化 / 刷新令牌并将纯文本令牌作为 JSON 响应返回的控制器方法如下所示。

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use Illuminate\Support\Str;

class ApiTokenController extends Controller

{

/\*\*

\* 更新认证用户的 API 令牌。

\*

\* @param \Illuminate\Http\Request $request

\* @return array

\*/

public function update(Request $request)

{

$token = Str::random(60);

$request->user()->forceFill([

'api\_token' => hash('sha256', $token),

])->save();

return ['token' => $token];

}

}

?>

{注} 由于上面例子中， API 令牌有足够的熵（熵的概念是由德国物理学家克劳修斯于 1865 年所提出。最初是用来描述 “能量退化” 的物质状态参数之一，在热力学中有广泛的应用），创建「彩虹表」来暴力查找哈希令牌的原始值基本是徒劳的，所以也就不需要使用类似于 bcrypt 这种哈希方法。

--

4、路由保护

Laravel 包含一个 身份认证看守器( https://learnku.com/docs/laravel/6.x/authentication#adding-custom-guards ) 可以自动验证传入请求的 API 令牌。 你只需要在任何需要有效访问令牌的路由上指定 auth:api 中间件。

use Illuminate\Http\Request;

Route::middleware('auth:api')->get('/user', function(Request $request) {

return $request->user();

});

--

5、请求中传递令牌

有几种方法可以将 API 令牌传递给你的应用程序。 我们将在使用 Guzzle HTTP 库演示其用法时去讨论这些方法。 你可以根据应用程序的需要选择其中的任何方法。

(1)、请求参数

你的应用程序的 API 使用者可以将其令牌作为 api\_token 查询字符串值。

$response = $client->request('GET', '/api/user?api\_token='.$token);

(2)、请求负载

应用程序的 API 使用者可以在请求的表单参数中以 api\_token 的形式包含其 API 令牌。

$response = $client->request('POST', '/api/user', [

'headers' => [

'Accept' => 'application/json',

],

'form\_params' => [

'api\_token' => $token,

],

]);

(3)、Bearer 令牌

应用程序的 API 使用者可以在请求的 Authorization 头中提供其 API 令牌作为 Bearer 令牌。

$response = $client->request('POST', '/api/user', [

'headers' => [

'Authorization' => 'Bearer '.$token,

'Accept' => 'application/json',

],

]);

--

--

三、用户授权

https://learnku.com/docs/laravel/6.x/authorization/5153

--

1、简介

除了提供开箱即用的 用户认证( https://learnku.com/docs/laravel/6.x/authentication ) 服务外，Laravel 还提供了一种简单的方法来处理用户的授权动作。与用户认证一样，Laravel 的授权方法很简单，授权操作有两种主要方式：gates 和策略。

可以把 gates 和策略比作路由和控制器。Gates 提供了一种简单的基于闭包的授权方法，而策略和控制器类似，围绕特定模型或资源对其逻辑进行分组来实现授权认证。我们先探索 gates，然后研究策略。

在构建一个应用的时候，不用在专门使用 gates 或者只使用策略之间进行选择。大部分应用很可能同时包含 gates 和策略， 并且能够很好的进行工作。 Gates 大部分应用在模型和资源没有关系的地方，比如查看管理员的面板。与之相反，策略应该在特定的模型或者资源中使用。

--

2、Gates (网关)

(1)、编写 Gates

Gates 是用来决定用户是否授权执行给予动作的一个闭包函数，并且典型的做法就是在 App\Providers\AuthServiceProvider 中使用 Gate 来定义。Gates 总是接收一个用户实例作为第一个参数，并且可以接收可选参数，比如相关的 Eloquent 模型。

/\*\*

\* 注册任意用户认证、用户授权服务。

\*

\* @return void

\*/

public function boot()

{

$this->registerPolicies();

// edit-settings网关

Gate::define('edit-settings', function ($user) {

return $user->isAdmin;

});

// update-post网关

Gate::define('update-post', function ($user, $post) {

return $user->id === $post->user\_id;

});

}

//

Gates 也可以使用类似控制器方法 Class@method 风格的回调字符串来定义。

/\*\*

\* 注册任意用户认证、用户授权服务。

\*

\* @return void

\*/

public function boot()

{

$this->registerPolicies();

Gate::define('update-post', 'App\Policies\PostPolicy@update');

}

(2)、授权动作

使用 Gate 来授权动作的时候， 你应该使用 allows 或者 denies 方法。 注意，不需要将当前已认证用户传递给这些方法。 Laravel 会自动处理好已经认证通过的用户，然后传递给 Gate 闭包函数。

if (Gate::allows('edit-settings')) {

// 指定当前用户可以编辑设置

}

//

if (Gate::allows('update-post', $post)) {

// 指定当前用户可以进行更新...

}

//

if (Gate::denies('update-post', $post)) {

// 指定当前用户不能更新...

}

//

如果你想判断一个特定的用户是否已经被授权访问某个动作， 你可以使用在 Gate 在 facade 的 forUser 方法。

if (Gate::forUser($user)->allows('update-post', $post)) {

// 用户可以更新...

}

//

if (Gate::forUser($user)->denies('update-post', $post)) {

// 用户不能更新...

}

//

您可以使用 any 或 none 方法一次授权多个操作。

if (Gate::any(['update-post', 'delete-post'], $post)) {

// 用户可以更新或删除

}

if (Gate::none(['update-post', 'delete-post'], $post)) {

// 用户不能更新或删除

}

//

///1/ 授权或抛出异常

如果要尝试对某个操作进行授权，并在未授权用户进行该操作的情况下抛出 illuminate\auth\access\authorizationexception，则可以使用 gate::authorize 方法。authorizationexception 的实例将自动转换为 403 http 响应。

Gate::authorize('update-post', $post); // 当前行为已授权...

//

///2/ 提供上下文

能够用于授权的 Gate 方法（allows，denies，check，any，none，authorize，can，cannot）和授权 [blade directives]（@can，@cannot，@canany）可以接收一个数组作为第二个参数。这些数组元素作为参数传递给 gate ，在做出授权决策时可用于其他上下文。

// 定义

Gate::define('create-post', function ($user, $category, $extraFlag) {

return $category->group > 3 && $extraFlag === true;

});

// 授权检查

if (Gate::check('create-post', [$category, $extraFlag])) {

// The user can create the post...

}

(3)、Gate 响应

到目前为止，我们只检查了返回简单布尔值的 Gate 。但是，有时你可能希望返回更详细的响应，包括错误消息。为此，你可以从你的 Gate 返回 illuminate\auth\access\response。

use Illuminate\Support\Facades\Gate;

use Illuminate\Auth\Access\Response;

Gate::define('edit-settings', function ($user) {

return $user->isAdmin

? Response::allow()

: Response::deny('You must be a super administrator.');

});

//

从 Gate 返回授权响应时，gate::allows 方法仍将返回一个简单的布尔值；但是可以使用 gate::inspect 方法获取 Gate 返回的完整授权响应。

$response = Gate::inspect('edit-settings', $post);

if ($response->allowed()) {

// 当前行为已授权...

} else {

echo $response->message();

}

//

当然，当使用 gate::authorize 方法抛出 authorizationexception 时，如果操作未经授权，则授权响应提供的错误消息将传播到 http 响应。

Gate::authorize('edit-settings', $post);// 当前行为已授权...

(4)、Gate 拦截检查

有时，你可能希望将所有能力授予特定用户。所以你可以在所有其他授权检查之前使用 before 方法来定义运行的回调。

Gate::before(function ($user, $ability) {

if ($user->isSuperAdmin()) {

return true;

}

});

如果 before 回调方法返回的是非 null 的结果，则结果将被视为检查结果。

//

您可以使用 after 方法，定义一个在所有其他授权检查后执行的回调。

Gate::after(function ($user, $ability, $result, $arguments) {

if ($user->isSuperAdmin()) {

return true;

}

});

与 before 检查类似，如果 after 回调返回非 null 结果，则结果将被视为检查结果。

--

3、创建策略

(1)、生成策略

策略是在特定模型或者资源中组织授权逻辑的类。例如，你的应用是一个博客，那么你在创建或者更新博客的时候，你可能会有一个 Post 模型和一个对应的 PostPolicy 来授权用户动作。

可以使用 make:policy artisan command( https://learnku.com/docs/laravel/6.x/artisan ) 生成策略。 生成的策略将放在 app/Policies 目录。如果您的应用程序中不存在此目录，Laravel 将为您创建它。

//

///1/ 生成策略

// 空策略类

php artisan make:policy PostPolicy

// 模型策略

php artisan make:policy PostPolicy --model=Post

//

建议：所有策略都通过 Laravel 解析 service container( https://learnku.com/docs/laravel/6.x/container )，允许您在策略的构造函数中键入提示任何需要的依赖项，以便自动注入它们。

(2)、注册策略

一旦策略存在，就需要注册它。新 Laravel 应用程序中包含的 AuthServiceProvider 包含一个 policy 属性，它将您的 Eloquent 模型映射到它们相应的策略。注册一个策略将指导 Laravel 在授权针对给定模型的操作时使用哪个策略。

<?php

namespace App\Providers;

use App\Post;

use App\Policies\PostPolicy;

use Illuminate\Support\Facades\Gate;

use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

class AuthServiceProvider extends ServiceProvider

{

/\*\*

\* 应用程序的策略映射。

\*

\* @var array

\*/

protected $policies = [

Post::class => PostPolicy::class,

];

/\*\*

\* 注册任何应用程序 authentication / authorization 服务。

\*

\* @return void

\*/

public function boot()

{

$this->registerPolicies();

//

}

}

?>

//

///1/ 策略自动发现

不需要手动注册模型策略，只要模型和策略遵循标准的 Laravel 命名约定，Laravel 就可以自动发现策略。具体来说，策略必须位于包含模型的目录下的 Policies 目录中。例如，模型可以放在 app 目录中，而策略可以放在 app/Policies 目录中。此外，策略名称必须与模型名称匹配，并具有 Policy 后缀。因此，一个 User 模型将对应于一个 UserPolicy 类。

如果希望提供自己的策略发现逻辑，可以使用 Gate::guessPolicyNamesUsing 方法注册自定义回调。通常，这个方法应该从应用程序的 AuthServiceProvider 的 boot 方法中调用。

use Illuminate\Support\Facades\Gate;

Gate::guessPolicyNamesUsing(function ($modelClass) {

// 返回策略类名…

});

注意：在 AuthServiceProvider 中显式映射的任何策略都将优先于任何潜在的自动发现策略。

--

4、编写策略

(1)、策略方法

一旦注册了策略，您可以为它授权的每个操作添加方法。例如，让我们在 PostPolicy 上定义一个 update 方法，它确定给定的 User 是否可以更新给定的 Post 实例。

update 方法将接收一个 User 和一个 Post 实例作为参数，并应返回 true 或 false，指示用户是否被授权更新给定的 Post。所以，在这个例子中，让我们验证用户的 id 是否与帖子上的 user\_id 匹配。

//// PostPolicy.php post策略

<?php

namespace App\Policies;

use App\User;

use App\Post;

class PostPolicy

{

/\*\*

\* 确定用户是否可以更新给定的帖子。

\*

\* @param \App\User $user

\* @param \App\Post $post

\* @return bool

\*/

public function update(User $user, Post $post)

{

return $user->id === $post->user\_id;

}

}

?>

您可以根据策略授权的各种操作的需要，继续在策略上定义其他方法。例如，您可以定义 view 或 delete 方法来授权各种 Post 操作，但请记住，您可以自由地为策略方法指定任何名称。

建议： 如果您在通过 Artisan 控制台生成策略时使用了 --model 选项，那么它已经包含了 view、create、update、delete、restore 和 forceDelete 操作的方法。

(2)、Policy 响应 (策略响应)

到目前为止，我们只研究了返回简单布尔值的策略方法。然而，有时您可能希望返回更详细的响应，包括错误消息。为此，您可以从您的策略方法返回一个 \Auth\Access\Response。

use Illuminate\Auth\Access\Response;

/\*\*

\* 确定用户是否可以更新给定的帖子。

\*

\* @param \App\User $user

\* @param \App\Post $post

\* @return bool

\*/

public function update(User $user, Post $post)

{

return $user->id === $post->user\_id

? Response::allow()

: Response::deny('You do not own this post.');

}

当从策略返回授权响应时，Gate::allows 方法仍然返回一个简单的布尔值；但是，您可以使用 Gate::inspect 方法来获得 Gate 返回的完整授权响应。

$response = Gate::inspect('update', $post);

if ($response->allowed()) {

// The action is authorized...

} else {

echo $response->message();

}

当然，当使用 Gate::authorize 方法在未授权操作时抛出 AuthorizationException，授权响应提供的错误消息将传播到 HTTP 响应。

Gate::authorize('update', $post);// 该行动被授权…

(3)、不包含模型方法

一些策略方法只接收当前经过身份验证的用户，而不接收它们授权的模型的实例。这种情况在授权 create 操作时最为常见。例如，如果您正在创建博客，您可能希望检查用户是否被授权创建任何帖子。

当定义不接收模型实例的策略方法时，例如 create 方法，它将不接收模型实例。相反，您应该将方法定义为只期望经过身份验证的用户。

/\*\*

\* 确定给定用户是否可以创建帖子。

\*

\* @param \App\User $user

\* @return bool

\*/

public function create(User $user)

{

//

}

(4)、访客 - 用户 (Guest 用户)

默认情况下，如果传入的 HTTP 请求不是由经过身份验证的用户发起的，所有的门和策略都会自动返回 false。但是，您可以通过声明一个「可选」类型提示或为用户参数定义提供一个 null 默认值，从而允许这些授权检查通过您的门和策略。

<?php

namespace App\Policies;

use App\User;

use App\Post;

class PostPolicy

{

/\*\*

\* 确定用户是否可以更新给定的帖子。

\*

\* @param \App\User $user

\* @param \App\Post $post

\* @return bool

\*/

public function update(?User $user, Post $post)

{

return $user->id === $post->user\_id;

}

}

?>

(5)、策略过滤器

对于某些用户，您可能希望授权给定策略中的所有操作。为此，在策略上定义一个 before 方法。before 方法将在策略上的任何其他方法之前执行，从而使您有机会在实际调用预期的策略方法之前授权操作。此功能最常用于授权应用程序管理员执行任何操作。

public function before($user, $ability)

{

if ($user->isSuperAdmin()) {

return true;

}

}

如果您想拒绝用户的所有授权，您应该从 before 方法返回 false。如果返回 null，则授权将传递给策略方法。

注意：如果策略类的 before 方法不包含与正在检查的功能名称匹配的名称的方法，则不会调用该方法。

--

5、使用策略去授权动作 (使用策略授权动作)

(1)、通过用户模型

Laravel 应用程序中包含的 User 模型包括两个用于授权操作的有用方法：can 和 cant。can 方法接收您希望授权的操作和相关模型。例如，让我们来确定一个用户是否被授权更新一个给定的 Post 模型。

if ($user->can('update', $post)) {

//

}

如果为给定模型 注册了策略( https://learnku.com/docs/laravel/6.x/authorization/5153#registering-policies )，can 方法将自动调用适当的策略并返回布尔结果。如果没有为模型注册策略，can 方法将尝试调用匹配给定操作名称的基于闭包的 Gate。

//

///1/ 不需要指定模型的动作

记住，像 create 这样的操作可能不需要模型实例。在这些情况下，可以将类名传递给 can 方法。类名将用于确定在授权操作时使用哪个策略。

use App\Post;

if ($user->can('create', Post::class)) {

// 在相关策略上执行 "create" 方法…

}

(2)、通过中间件

Laravel 包含一个中间件，可以在传入的请求到达路由或控制器之前对操作进行授权。默认情况下，Illuminate\Auth\Middleware\Authorize 中间件被分配给 App\Http\Kernel 类中的 can 键。让我们来探索一个使用 can 中间件授权用户更新博客文章的例子。

use App\Post;

Route::put('/post/{post}', function (Post $post) {

// 当前用户可以更新帖子…

})->middleware('can:update,post');

在本例中，我们传递了 can 中间件两个参数。第一个是希望授权的操作的名称，第二个是希望传递给策略方法的路由参数。在本例中，由于我们使用隐式模型绑定( https://learnku.com/docs/laravel/6.x/routing#implicit-binding )，一个 Post 模型将被传递给策略方法。如果用户没有被授权执行给定的操作，则中间件将生成一个带有 403 状态代码的 HTTP 响应。

//

///1/ 不需要指定模型的动作

同样，像 create 这样的一些操作可能不需要模型实例。在这些情况下，可以将类名传递给中间件。类名将用于确定在授权操作时使用哪个策略。

Route::post('/post', function () {

// 当前用户可以创建贴子…

})->middleware('can:create,App\Post');

(3)、通过控制器辅助函数

除了为 User 模型提供的有用方法之外，Laravel 还为任何扩展 App\Http\Controllers\Controller 基类的控制器提供了一个有用的 authorize 方法。 与 can 方法一样，此方法接受您要授权的操作的名称和相关模型。 如果操作未被授权，authorize 方法将抛出一个 Illuminate\Auth\Access\AuthorizationException ，默认的 Laravel 异常处理程序将转换为具有 403 状态代码的 HTTP 响应。

<?php

namespace App\Http\Controllers;

use App\Post;

use Illuminate\Http\Request;

use App\Http\Controllers\Controller;

class PostController extends Controller

{

/\*\*

\* 更新指定博客帖子。

\*

\* @param Request $request

\* @param Post $post

\* @return Response

\* @throws \Illuminate\Auth\Access\AuthorizationException

\*/

public function update(Request $request, Post $post)

{

$this->authorize('update', $post);

// 当前用户可以更新博客....

}

}

?>

//

///1/ 不需要指定模型的动作

如前所述，像 create 这样的一些动作可能不需要模型实例。 在这些情况下，您应该将类名传递给 authorize 方法。 类名将用于确定授权操作时使用的策略。

/\*\*

\* 创建一个新的博客

\*

\* @param Request $request

\* @return Response

\* @throws \Illuminate\Auth\Access\AuthorizationException

\*/

public function create(Request $request)

{

$this->authorize('create', Post::class);

// 当前用户可以新建博客...

}

//

///2/ 授权资源控制器

如果你使用的是 资源控制器( https://learnku.com/docs/laravel/6.x/controllers#resource-controllers )，那么你就可以在控制器构造方法里使用 authorizeResource 方法。此方法将适当的 can 中间件定义附加到资源控制器相应的方法中。

authorizeResource 方法接收模型类名作为第一个参数，以及路由名称 / 包含模型 ID 的请求参数作为其第二个参数。

<?php

namespace App\Http\Controllers;

use App\Post;

use Illuminate\Http\Request;

use App\Http\Controllers\Controller;

class PostController extends Controller

{

public function \_\_construct()

{

$this->authorizeResource(Post::class, 'post');

}

}

?>

//

以下控制器方法将映射到其对应的策略方法。

控制器方法/策略方法

index viewAny

show view

create create

store create

edit update

update update

destroy delete

{提示} 你可以使用带有 --model 选项的 make:policy 命令去快速生成基于给定模型的策略类：php artisan make:policy PostPolicy --model=Post。

(4)、通过 Blade 模板

当编写 Blade 模板时，可能希望仅在用户被授权执行给定操作时才显示页面的一部分。比如，你可能希望仅在用户可以实际更新帖子时显示博客帖子的更新表单。在这样情况下，你可以使用 @can 和 @cannot 等一系列指令。

@can('update', $post)

<!-- 当前用户可以更新文章 -->

@elsecan('create', App\Post::class)

<!-- 当前用户可以新建文章 -->

@endcan

//

@cannot('update', $post)

<!-- 当前用户不可以更新文章 -->

@elsecannot('create', App\Post::class)

<!-- 当前用户不可以新建文章 -->

@endcannot

//

这些指令是编写 @if 和 @unless 语句的捷径。 @can 和 @cannot 语句分别转化为以下语句。

@if (Auth::user()->can('update', $post))

<!-- 当前用户可以更新文章 -->

@endif

//

@unless (Auth::user()->can('update', $post))

<!-- 当前用户不可以更新文章 -->

@endunless

//

您还可以确定用户是否具有来自给定能力列表的任何授权能力。 要实现这一点，请使用 @ canany 指令。

@canany(['update', 'view', 'delete'], $post)

// 当前用户可以更新，查看和删除文章

@elsecanany(['create'], \App\Post::class)

// 当前用户可以创建文章

@endcanany

//

///1/ 不依赖模型的动作

与大多数其他授权方法一样，如果动作不需要模型实例，则可以将类名传递给 @ can 和 @ cannot 指令。

@can('create', App\Post::class)

<!-- 当前用户可以创建文章 -->

@endcan

//

@cannot('create', App\Post::class)

<!-- 当前用户不可以创建文章 -->

@endcannot

(5)、提供上下文 (提供附加上下文)

使用策略授权操作时，您可以将数组作为第二个参数传递给各种授权函数和辅助方法。 数组中的第一个元素将用于确定应调用哪个策略，而其余数组元素作为参数传递给策略方法，并可在进行授权决策时用于其他上下文。 例如，考虑以下 PostPolicy 方法定义，其中包含一个额外的 $ category 参数。

/\*\*

\* 确定用户是否可以更新给定的帖子。

\*

\* @param \App\User $user

\* @param \App\Post $post

\* @param int $category

\* @return bool

\*/

public function update(User $user, Post $post, int $category)

{

return $user->id === $post->user\_id &&

$category > 3;

}

//

在尝试确定验证过的用户是否可以更新给定文章时，我们可以像这样调用此策略方法。

/\*\*

\* 更新给定的博客文章.

\*

\* @param Request $request

\* @param Post $post

\* @return Response

\* @throws \Illuminate\Auth\Access\AuthorizationException

\*/

public function update(Request $request, Post $post)

{

$this->authorize('update', [$post, $request->input('category')]);

//当前用户可以更新这篇博客文章...

}

--

--

四、Email 认证

https://learnku.com/docs/laravel/6.x/verification/5154

--

1、简介

很多 Web 应用会要求用户在使用之前进行 Email 地址验证。Laravel 不会强迫你在每个应用中重复实现它，而是提供了便捷的方法来发送和校验电子邮件的验证请求。

(1)、Model 准备

在开始之前，需要验证你的 App\User 模型是否实现了 Illuminate\Contracts\Auth\MustVerifyEmail 契约。

<?php

namespace App;

use Illuminate\Notifications\Notifiable;

use Illuminate\Contracts\Auth\MustVerifyEmail;

use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable implements MustVerifyEmail

{

use Notifiable;

// ...

}

?>

--

2、数据库注意事项

(1)、Email 验证字段

接下来，你的 user 表必须包含一个 email\_verified\_at 字段用来存储 Email 地址通过验证的时间。默认情况下，Laravel 框架中 users 表的数据迁移已经包含了这个字段。所以，您需要做的就只是执行数据库迁移：php artisan migrate 。

--

3、路由

Laravel 的 Auth\VerificationController 类包含了发送验证链接和验证 Email 的必要逻辑。通过将 verify 选项传给 Auth::routes 方法，就能为这个控制器注册所需要的路由。

Auth::routes(['verify' => true]);

(1)、保护路由

路由中间件( https://learnku.com/docs/laravel/6.x/middleware ) 可用于仅允许经过验证的用户访问指定路由。Laravel 附带了 verified 中间件，它定义在 Illuminate\Auth\Middleware\EnsureEmailIsVerified。由于此中间件已在应用程序的 HTTP 内核中注册，因此您需要做的就是将中间件附加到路由定义。

Route::get('profile', function () {

// 只有经过验证的用户才能进..

})->middleware('verified');

--

4、视图

如果要生成 Email 验证的所有必要视图，您可以使用 laravel / ui Composer 包。

composer require laravel/ui

php artisan ui vue --auth

Email 验证视图文件位于 resources / views / auth / verify.blade.php 中。 你可以根据自己的应用自由地调整这些视图的样式。

--

5、Email 认证之后

在邮箱认证之后，用户会自动被重定向至 /home。你可通过在 VerificationController 中定义一个 redirectTo 方法或者属性来调整认证之后的跳转位置。

protected $redirectTo = '/dashboard';

--

6、事件

Laravel 在 Email 验证过程中发送事件( https://learnku.com/docs/laravel/6.x/events )。 你应该在 EventServiceProvider 中注册监听者。

/\*\*

\* 应用程序的事件监听器

\*

\* @var array

\*/

protected $listen = [

'Illuminate\Auth\Events\Verified' => [

'App\Listeners\LogVerifiedUser',

],

];

--

--

五、加密解密 (加密解密机制)

https://learnku.com/docs/laravel/6.x/encryption/5155

--

1、简介

Laravel 的加密机制使用的是 OpenSSL 所提供的 AES-256 和 AES-128 加密。强烈建议你使用 Laravel 内建的加密工具，而不是用其它的加密算法。所有 Laravel 加密之后的结果都会使用消息认证码 (MAC) 签名，使其底层值不能在加密后再次修改。

--

2、设置

在使用 Laravel 的加密工具之前，你必须先设置 config/app.php 配置文件中的 key 选项。你应当使用 php artisan key:generate 命令来生成密钥，这条 Artisan 命令会使用 PHP 的安全随机字节生成器来构建密钥。如果这个 key 值没有被正确设置，则所有由 Laravel 加密的值都会是不安全的。

--

3、基本用法

(1)、加密一个值

你可以使用辅助函数 encrypt 来加密一个值。所有加密的值都使用 OpenSSL 的 AES-256-CBC 来进行加密。此外，所有加密过的值都会使用消息认证码 (MAC) 来签名，以检测加密字符串是否被篡改过。

<?php

namespace App\Http\Controllers;

use App\User;

use Illuminate\Http\Request;

use App\Http\Controllers\Controller;

class UserController extends Controller

{

/\*\*

\* 存储用户的加密信息

\*

\* @param Request $request

\* @param int $id

\* @return Response

\*/

public function storeSecret(Request $request, $id)

{

$user = User::findOrFail($id);

$user->fill([

'secret' => encrypt($request->secret),

])->save();

}

}

?>

(2)、无序列化加密

加密的时候，对象和数组加密过的值都需要通过 serialize 序列化后传递。因此，非 PHP 客户端接收加密值，需要对数据进行 unserialize 反序列化。如果想要在不序列化的情况下加密或解密值，你可以使用 Crypt Facade 的 encryptString 和 decryptString 方法。

use Illuminate\Support\Facades\Crypt;

$encrypted = Crypt::encryptString('Hello world.');

$decrypted = Crypt::decryptString($encrypted);

(3)、解密一个值

你可以使用辅助函数 decrypt 来进行解密。如果该值不能被正确解密，例如 MAC 无效时，会抛出异常 Illuminate\Contracts\Encryption\DecryptException。

use Illuminate\Contracts\Encryption\DecryptException;

try {

$decrypted = decrypt($encryptedValue);

} catch (DecryptException $e) {

//

}

--

--

六、哈希

https://learnku.com/docs/laravel/6.x/hashing/5156

--

1、简介

Laravel Hash facade( https://learnku.com/docs/laravel/6.x/facades ) 为存储用户密码提供了安全的 Bcrypt 和 Argon2 哈希加密方式。如果你在你的 Laravel 应用程序中使用了内置的 LoginController 和 RegisterController 类，那么它们默认使用 Bcrypt 进行注册和身份认证。

Tip：Bcrypt 是哈希密码的理想选择，因为它的 「加密系数」 可以任意调整，这意味着生成哈希所需的时间可以随着硬件功率的增加而增加。

--

2、配置

你可以在 config/hashing.php 配置文件中配置默认哈希驱动程序。目前支持三种驱动程序：Bcrypt( https://en.wikipedia.org/wiki/Bcrypt ) 和 Argon2( https://en.wikipedia.org/wiki/Argon2 ) (Argon2i and Argon2id variants)。

注意：Argon2i 驱动程序需要 PHP 7.2.0 或更高版本，而 Argon2id 驱动程序则需要 PHP 7.3.0 或更高版本。

--

3、基本用法

你可以通过调用 Hash facade 的 make 方法来加密你的密码。

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use Illuminate\Support\Facades\Hash;

use App\Http\Controllers\Controller;

class UpdatePasswordController extends Controller

{

/\*\*

\* 更新用户密码。

\*

\* @param Request $request

\* @return Response

\*/

public function update(Request $request)

{

// 验证新密码的长度…

$request->user()->fill([

'password' => Hash::make($request->newPassword)

])->save();

}

}

?>

(1)、调整 Bcrypt 加密系数

如果使用 Bcrypt 算法，你可以在 make 方法中使用 rounds 选项来配置该算法的加密系数。然而，对大多数应用程序来说，默认值就足够了。

$hashed = Hash::make('password', [

'rounds' => 12

]);

(2)、调整 Argon2 加密系数

如果使用 Argon2 算法，你可以在 make 方法中使用 memory， time 和 threads 选项来配置该算法的加密系数。然后，对大多数应用程序来说，默认值就足够了。

$hashed = Hash::make('password', [

'memory' => 1024,

'time' => 2,

'threads' => 2,

]);

Tip：有关这些选项的更多信息，请查阅 PHP 官方文档( https://secure.php.net/manual/en/function.password-hash.php )。

(3)、密码哈希验证

check 方法能为您验证一段给定的未加密字符串与给定的哈希值是否一致。然而，如果您使用 Laravel 内置的 LoginController 控制器( https://learnku.com/docs/laravel/6.x/authentication )，您可能不需要直接使用这个方法，因为该控制器会自动调用这个方法。

if (Hash::check('plain-text', $hashedPassword)) {

// 密码匹配…

}

(4)、检查密码是否需要重新哈希

needsRehash 方法可以为您检查当哈希的加密系数改变时，您的密码是否被新的加密系数重新加密过。

if (Hash::needsRehash($hashed)) {

$hashed = Hash::make('plain-text');

}

--

--

七、重置密码

https://learnku.com/docs/laravel/6.x/passwords/5157

--

1、简介

Tip：想要快速上手？ 只需要在新建的 Laravel 应用中使用 Composer 安装 laravel/ui 拓展包并且运行 php artisan ui vue --auth， 然后在浏览器中打开 http://your-app.test/register 或者给你的应用分配任意一个 URL。 这个命令将会负责搭建起整个身份验证系统，包括重置密码！

大多数 web 应用都为用户提供了重置密码的功能。相较于强迫你在每个应用中都要重新实现一遍此功能，Laravel 提供了便捷的方法来发送密码提醒以及执行密码重置。

注意：在使用 Laravel 的密码重置功能之前，你的用户模型必须使用 Illuminate\Notifications\Notifiable trait。

--

2、数据库注意事项

首先，验证你的 App\User 模型是否实现了 Illuminate\Contracts\Auth\CanResetPassword 契约。当然，框架中包含的 App\User 模型已经实现了该接口，并且使用了 Illuminate\Auth\Passwords\CanResetPassword trait 来包含实现该接口所需的方法。

(1)、生成重置令牌的表迁移

接下来，必须创建一张数据表来存储密码重置令牌。该数据表的迁移已包含在 Laravel 应用的 database/migrations 目录中。 所以，你需要做的只是执行数据库迁移命令。

php artisan migrate

--

3、路由

Laravel 已在 Auth\ForgotPasswordController 和 Auth\ResetPasswordController 类中包含了发送密码重置链接电子邮件和重置用户密码的逻辑。 所有重置密码需要的路由都可以通过 Composer 的 laravel/ui 拓展包生成。

composer require laravel/ui

php artisan ui vue --auth

--

4、视图

要生成重置密码必要的视图文件，你可以使用 Composer 的 laravel/ui 拓展包。

composer require laravel/ui

php artisan ui vue --auth

这些视图在 resources/views/auth/passwords 中。 你可以随意根据需要为您的应用程序自定义它们。

--

5、重置密码后

一旦你定义了重置用户密码的路由和视图，你可以在浏览器中访问 /password/reset 这个路由来重置密码。框架中的 ForgotPasswordController 已包含发送密码重置链接电子邮件的逻辑，同时 ResetPasswordController 包含了重置用户密码的逻辑。

在重置密码后，用户将会自动登录并重定向到 /home。 你可以通过在 ResetPasswordController 中定义一个 redirectTo 属性来自定义密码重置后重定向的位置。

protected $redirectTo = '/dashboard';

注意：默认情况下，密码重置令牌会在一小时后过期。你可以通过 config/auth.php 文件中的密码重置 expire 选项对此进行修改。

--

6、自定义

(1)、自定义身份验证看守器

在你的 auth.php 配置文件中，你可以配置多个「看守器」，可以用来定义多个用户表的身份验证行为。你可以自定义框架中的 ResetPasswordController ，通过重写该控制器中的 guard 方法来使用你所选择的看守器。这个方法应当返回一个看守器实例。

use Illuminate\Support\Facades\Auth;

/\*\*

\* 在密码重置期间看守

\*

\* @return \Illuminate\Contracts\Auth\StatefulGuard

\*/

protected function guard()

{

return Auth::guard('guard-name');

}

(2)、自定义密码代理

在你的 auth.php 配置文件中，你可以配置多个密码「代理」，可以用来重置多个用户表上的密码。你可以自定义框架中的 ForgotPasswordController 和 ResetPasswordController ， 通过重写控制器中的 broker 方法来使用你所选择的代理。

use Illuminate\Support\Facades\Password;

/\*\*

\* 获取在密码重置期间使用的代理。

\*

\* @return PasswordBroker

\*/

public function broker()

{

return Password::broker('name');

}

(3)、自定义密码重置邮件

你可以轻松地修改用于向用户发送密码重置链接的通知类。首先，重写 User 模型中的 sendPasswordResetNotification 方法。 在此方法中，你可以使用任何你所选择的通知类来发送通知。该方法接收的第一个参数是密码重置令牌 $token。

/\*\*

\* 发送密码重置通知。

\*

\* @param string $token

\* @return void

\*/

public function sendPasswordResetNotification($token)

{

$this->notify(new ResetPasswordNotification($token));

}

--

--

第七卷 综合话题

--

--

一、Artisan 命令行

https://learnku.com/docs/laravel/6.x/artisan/5158

--

1、简介

Artisan 是 Laravel 自带的命令行接口，他提供了许多使用的命令来帮助你构建 Laravel 应用 。要查看所有可用的 Artisan 命令的列表，可以使用 list 命令。

// 查看所有命令列表

php artisan list

//

每个命令都包含了「帮助」 界面 ，它会显示和概述命令的可用参数及选项。只需要在命令前加上 help 即可查看命令帮助界面。

// 查看命令帮助信息

php artisan help migrate

(1)、Tinker 命令 (REPL Read-Eval-Print-Loop，“读取-求值-输出-循环”)

所有 Laravel 应用都包含了 Tinker，一个基于 PsySH( https://github.com/bobthecow/psysh ) 包提供支持的 REPL 。Tinker 让你可以在命令行中与你整个的 Laravel 应用进行交互 。 包括 Eloquent ORM、任务、事件等等。运行 Artisan 命令 tinker 进入 Tinker 环境。

// 进入 Tinker 环境

php artisan tinker

//

// 发布Tinker配置文件

你可以通过使用 vendor:publish 命令发布 Tinker 配置文件。

php artisan vendor:publish --provider="Laravel\Tinker\TinkerServiceProvider"

(2)、命令白名单

Tinker 通过采用白名单的方式来确定允许哪些 Artisan 命令可以在 shell 中运行。默认情况下，你可以运行 clear-compiled 、down、env、 inspire、migrate、 optimize、和 up 命令。如果你想要添加更多的白名单命令，可以将它们添加到 tinker.php 配置文件中的 commands 数组里：

// tinker.php配置文件

'commands' => [

// App\Console\Commands\ExampleCommand::class,

],

(3)、黑名单别名

通常，Tinker 会在 Tinker 中根据你的需要自动为类添加别名。然而，你可能不希望为某些类添加别名。你可以在 tinker.php 配置文件中的 dont\_alias 数组里列举这些类来完成此操作。

// tinker.php配置文件，不希望为某些类添加别名

'dont\_alias' => [

App\User::class,

],

--

2、编写命令

除 Artisan 提供的命令外，你还可以构建自己的自定义命令。 命令通常存储在 app/Console/Commands 目录中。不过，只要你的命令可以由 Composer 加载，你就可以自由选择自己的存储位置。

(1)、生成命令

要创建一个新的命令，可以使用 Artisan 命令 make:command。这个命令会在 app/Console/Commands 目录中创建一个新的命令类。 不必担心应用中不存在这个目录，因为它会在你第一次运行 Artisan 命令 make:command 时创建。生成的命令会包括所有命令中默认存在的属性和方法。

// 生成命令

php artisan make:command SendEmails

(2)、命令结构

命令运行后，你应该先填写 signature 和 description 属性以便你在输入 php artisan list 时能够清楚知道用法。执行命令时会调用 handle 方法，你可以在这个方法中放置命令逻辑。

Tip：为了代码更好的复用，最好保持你的控制台代码轻量并且能够延迟到应用服务中完成。在下面的例子中，我们将注入一个服务类来完成发送邮件的重任。

来看一个简单的例子。我们可以在 handle 方法中注入我们需要的任何依赖项。 Laravel 服务容器( https://learnku.com/docs/laravel/6.x/container ) 将会自动注入所有在构造函数中带类型约束的依赖。

//

//// SendEmails.php

<?php

namespace App\Console\Commands;

use App\User;

use App\DripEmailer;

use Illuminate\Console\Command;

class SendEmails extends Command

{

/\*\*

\* 命令行的名称及签名。

\*

\* @var string

\*/

protected $signature = 'email:send {user}';

/\*\*

\* 命令行的描述

\*

\* @var string

\*/

protected $description = 'Send drip e-mails to a user';

/\*\*

\* 创建新的命令行实例。

\*

\* @return void

\*/

public function \_\_construct()

{

parent::\_\_construct();

}

/\*\*

\* 执行命令行。

\*

\* @param \App\DripEmailer $drip

\* @return mixed

\*/

public function handle(DripEmailer $drip)

{

$drip->send(User::find($this->argument('user')));

}

}

?>

(3)、闭包命令

基于闭包的命令提供了一个用类代替定义控制台命令的方法。同理，闭包路由是控制器的一种替代方法，而闭包命令则可以认为是命令类的替代方法。在 app/Console/Kernel.php 文件的 commands 方法中，Laravel 加载了 routes/console.php file。

/\*\*

\* 注册闭包命令

\*

\* @return void

\*/

protected function commands()

{

require base\_path('routes/console.php');

}

虽然这个文件没有定义 HTTP 路由，但是它将基于控制台的入口点（路由）定义到了应用中。在这个文件，你可以使用 Artisan::command 方法定义所有的闭包路由。 command 方法接受两个参数: 命令名称( https://learnku.com/docs/laravel/6.x/artisan/5158#defining-input-expectations )和一个接受命令参数及选项的闭包。

Artisan::command('build {project}', function ($project) {

$this->info("Building {$project}!");

});

闭包绑定底层的命令实例，因为你可以访问所有能够在完整命令类中的所有辅助方法。

//

///1/ 类型提示依赖

除了接收命令的参数和选项外，命令闭包也可以使用类型提示从 服务容器 中解析其他的依赖关系。

use App\User;

use App\DripEmailer;

Artisan::command('email:send {user}', function (DripEmailer $drip, $user) {

$drip->send(User::find($user));

});

//

///2/ 闭包命令描述

当你定义一个闭包命令，你应当使用 describe 方法来为命令添加描述。这个描述会在你运行 php artisan list 或者 php artisan help 命令时显示。

Artisan::command('build {project}', function ($project) {

$this->info("Building {$project}!");

})->describe('Build the project');

--

3、定义输入期望

在编写控制台命令时，通常是通过参数和选项来收集用户输入的。Laravel 可以通过 signature 属性非常方便的定义你期望用户输入的内容。signature 允许使用单一且可读性高，类似路由的语法定义命令的名称、参数和选项。

(1)、参数

所有用户提供的参数及选项都被包含在花括号中。在下面的例子中，这个命令定义了一个 必须的参数: user。

/\*\*

\* 命令行的名称及签名。

\*

\* @var string

\*/

protected $signature = 'email:send {user}';

//

你也可以创建可选参数，并定义参数的默认值。

// 可选参数...

email:send {user?}

// 带有默认值的可选参数...

email:send {user=foo}

(2)、选项

选项，类似于参数，是用户输入的另一种格式。当命令行指定选项时，它们以两个连字符 ( -- ) 作为前缀。有两种类型的选项：接收值和不接收值。不接收值的选项就像是一个布尔值的「开关」。让我们看一下这种类型的选项的例子。

/\*\*

\* 命令行的名称及签名。

\*

\* @var string

\*/

protected $signature = 'email:send {user} {--queue}';

在这个例子中，可以在调用 Artisan 命令时指定 --queue 开关。如果 --queue 开关被传递，该选项的值为 true， 否则为 false。

php artisan email:send 1 --queue

//

///1/ 带值的选项

接下来，让我们来看一个带值的选项。如果想让用户必须为选项指定一个值，则需要在选项名称末尾追加一个等号 = 作为的后缀。

/\*\*

\* 命令行的名称及签名。

\*

\* @var string

\*/

protected $signature = 'email:send {user} {--queue=}';

//

// 在这个例子中， 用户可以传递该选项的值，如下所示。

php artisan email:send 1 --queue=default

//

你也可以通过在选项名称后面指定默认值来设定该选项的默认值。如果用户没有传递选项值，将使用设定的默认值。

email:send {user} {--queue=default}

//

///2/ 选项简写

要在定义选项时指定简写，你可以在选项名称前指定它，并用 | 分隔符将简写与完整选项名称分隔开。

email:send {user} {--Q|queue}

(3)、输入数组

如果你想定义接收数组输入的参数或选项，你可以使用 \* 符号。首先，我们先看一个数组参数的实例：email:send {user\*}。

调用此方法时，该 user 参数的输入参数可按顺序传递给命令行。例如，以下命令会设置 user 的值为 ['foo', 'bar']。

php artisan email:send foo bar

//

当定义接收数组的选项时，传递给命令行的每个选项值都应以选项名称为前缀。

email:send {user} {--id=\*}

php artisan email:send --id=1 --id=2

(4)、输入说明

你可以通过使用冒号来为参数和选项添加说明，并使其将他们隔开。如果你需要一点额外的空间来定义你的命令，可以随意分开在多个行里。

/\*\*

\* 命令行的名称及签名。

\*

\* @var string

\*/

protected $signature = 'email:send

{user : The ID of the user}

{--queue= : Whether the job should be queued}';

--

4、I/O 命令 (Command I/O)

(1)、检索输入 (获取输入)

//

///1/ 参数获取

在命令执行时，显然你需要获取命令接收到的参数和选项的值。你可以用 argument 和 option 方法来达到目的。

/\*\*

\* 执行命令。

\*

\* @return mixed

\*/

public function handle()

{

$userId = $this->argument('user');

//

}

//

如果你想所有的参数以 array 数组获取，可以调用 arguments 方法。

$arguments = $this->arguments();

//

///2/ 选项获取

和获取参数类似，option 方法可以非常容易的获取选项的值。要将所有的选项以数组获取，使用 options 方法。

// 获取一个指定的选项值...

$queueName = $this->option('queue');

// 获取所有的选项值...

$options = $this->options();

如果参数或选项不存在，则返回 null

(2)、交互输入

除了显示输出外，你还可以要求用户在命令执行时提供输入。 ask 方法将提示用户输入并接收，然后用户的输入将会传入你的命令。

/\*\*

\* 执行命令。

\*

\* @return mixed

\*/

public function handle()

{

$name = $this->ask('What is your name?');

}

secret 方法和 ask 方法类似，但当用户在控制台输入时他们的输入内容是不可见的。这个方法适用于需要用户输入像密码这样的敏感信息的时候。

$password = $this->secret('What is the password?');

//

///1/ 请求确认

如果你想要寻求用户确认一些简单的信息，你可以使用 confirm 方法。默认情况下，该方法将返回 false 。但如果用户在回复中输入 y 或者 yes 则会返回 true。

if ($this->confirm('Do you wish to continue?')) {

//

}

//

///2/ 自动补全

anticipate 方法可用于为可能的选择提供自动补全功能。用户仍然可以忽略自动补全的提示，作任意回答。

$name = $this->anticipate('What is your name?', ['Taylor', 'Dayle']);

//

///3/ 多重选择

如果你要给用户提供一些预设的选择，可以使用 choice 方法。你也可以设置默认值的索引，用以应对用户没有选择的情景。

$name = $this->choice('What is your name?', ['Taylor', 'Dayle'], $defaultIndex);

(3)、编写输出

可以使用 line 、 info 、 comment 、 question 和 error 方法来将输出发送到终端。每个方法都使用适当的 ANSI 颜色表明其目的。例如，让我们向用户显示一些普通信息，通常来说，最好使用 info 方法，它会在控制台将输出的内容显示为绿色。

//

///1/ 普通信息 info()

/\*\*

\* 执行命令。

\*

\* @return mixed

\*/

public function handle()

{

$this->info('Display this on the screen');

}

//

///2/ 错误信息 error()

显示错误信息， 使用 error 方法。 错误信息则会显示为红色。

$this->error('Something went wrong!');

//

///3/ 无色输出 line()

如果你想在控制台显示无颜色设置的输出，请使用 line 方法。

$this->line('Display this on the screen');

//

///4/ 表格布局

对于多行列数据的格式化输出，table 方法处理起来更轻松。基于传入的表头和行数据，它会动态计算宽高。

$headers = ['Name', 'Email'];

$users = App\User::all(['name', 'email'])->toArray();

$this->table($headers, $users);

//

///5/ 进度条

对于耗时任务，提示进度非常有必要。使用 output 对象就可以创建、加载以及停止进度条。首先，定义好任务总步数，然后，在每次任务执行时加载进度条。

$users = App\User::all();

$bar = $this->output->createProgressBar(count($users));

$bar->start();

foreach ($users as $user) {

$this->performTask($user);

$bar->advance();

}

$bar->finish();

参阅 Symfony Progress Bar component documentati...( https://symfony.com/doc/current/components/console/helpers/progressbar.html ) 获取更多高级用法。

--

5、注册命令

app/Console/Commands 目录下的命令都会被注册，这是由于控制台内核的 commands 方法调用了 load。实际上，可随意调用 load 来扫描其他目录下的 Artisan 命令。

/\*\*

\* 注册命令

\*

\* @return void

\*/

protected function commands()

{

$this->load(\_\_DIR\_\_.'/Commands');

$this->load(\_\_DIR\_\_.'/MoreCommands');

// ...

}

也可以在 app/Console/Kernel.php 文件的 $commands 属性中手动注册命令的类名。Artisan 启动时，这个属性列出的命令都将由 服务容器(https://learnku.com/docs/laravel/6.x/containe) 解析并通过 Artisan 进行注册。

protected $commands = [

Commands\SendEmails::class

];

--

6、以编程方式执行命令 (程序调用命令)

有时需要在 CLI 之外执行 Artisan 命令，例如，在路由或控制器里触发 Artisan 命令。要实现调用，可以使用 Artisan 门面的 call 方法。call 方法的第一个参数接受命令名，第二个参数接受数组形式的命令参数。退出码将返回。

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

$exitCode = Artisan::call('email:send', [

'user' => 1, '--queue' => 'default'

]);

//

});

//

另外，你可以将整个 Artisan 命令作为字符串传递给 call 方法。

Artisan::call('email:send 1 --queue=default');

//

Artisan 门面的 queue 方法可以将 Artisan 命令队列化，交由 队列工作进程( https://learnku.com/docs/laravel/6.0/queues ) 进行后台处理。使用此方法之前，务必配置好队列以及运行队列监听器。

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

Artisan::queue('email:send', [

'user' => 1, '--queue' => 'default'

]);

//

});

//

你也可以指定 Artisan 命令派发的连接或任务。

Artisan::queue('email:send', [

'user' => 1, '--queue' => 'default'

])->onConnection('redis')->onQueue('commands');

//

///1/ 传递数组值

如果定义了接受数组的选项，可以直接传递数组到该选项。

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

$exitCode = Artisan::call('email:send', [

'user' => 1, '--id' => [5, 13]

]);

});

//

///2/ 传递布尔值

需要指定没有选项值的选项时，例如，migrate:refresh 命令的 --force 选项，就可以传入 true 或 false。

$exitCode = Artisan::call('migrate:refresh', [

'--force' => true,

]);

(1)、从其他命令调用命令 (命令的互相调用)

call 方法可以实现调用其它 Artisan 命令。call 方法接受命令名和数组形式的选项。

/\*\*

\* 执行命令

\*

\* @return mixed

\*/

public function handle()

{

$this->call('email:send', [

'user' => 1, '--queue' => 'default'

]);

//

}

//

如果要抑制控制台命令的所有输出，可以使用 callSilent 方法。callSilent 的使用方法同 call。

$this->callSilent('email:send', [

'user' => 1, '--queue' => 'default'

]);

--

--

二、广播系统

--

https://learnku.com/docs/laravel/6.x/broadcasting/5159

--

1、简介

在现代的 web 应用程序中， WebSockets 被用来实现实时、即时更新的用户接口。当服务器上的数据更新后，更新信息会通过 WebSocket 连接发送到客户端等待处理。相比于不停地轮询应用程序，这是一种更加可靠和高效的选择。

为了帮助你构建这类应用， Laravel 将通过 WebSocket 连接来使「广播」 事件 变得更加轻松。 广播 Laravel 事件允许你在服务端和客户端 JavaScript 应用程序间共享相同的事件名。

{注} 在深入了解事件广播之前，请确认你已阅读所有关于 Laravel 事件和监听器( https://learnku.com/docs/laravel/6.x/events ) 的文档。

(1)、配置

所有关于事件广播的配置都保存在 config/broadcasting.php 配置文件中。 Laravel 自带了几个广播驱动： Pusher( https://pusher.com/channels )、Redis( https://learnku.com/docs/laravel/6.x/redis )，和一个用于本地开发与调试的 log 驱动。另外，还有一个 null 驱动允许你完全关闭广播系统。每一个驱动的示例配置都可以在 config/broadcasting.php 配置文件中找到。

(2)、对驱动的要求

在对事件进行广播之前，你必须先注册 App\Providers\BroadcastServiceProvider 。对于一个新建的 Laravel 应用程序，你只需要在 config/app.php 配置文件的 providers 数组中取消对该提供者的注释即可。该提供者将允许你注册广播授权路由和回调。

//

///1/ CSRF 令牌

Laravel Echo ( https://learnku.com/docs/laravel/6.x/broadcasting/5159#installing-laravel-echo ) 需要访问当前会话的 CSRF 令牌。你应当验证你的应用程序的 head HTML 元素是否定义了包含 CSRF 令牌的 meta 标签。

<meta name="csrf-token" content="{{ csrf\_token() }}">

//

///2/ Pusher

如果你使用 Pusher( https://pusher.com/channels ) 来对事件进行广播，请用 Composer 包管理器来安装 Pusher PHP SDK。

composer require pusher/pusher-php-server "~4.0"

然后，你需要在 config/broadcasting.php 配置文件中配置你的 Pusher 证书。该文件中已经包含了一个 Pusher 示例配置，你可以快速地指定你的 Pusher key 、secret 和 application ID。 config/broadcasting.php 文件的 pusher 配置项同时也允许你指定 Pusher 支持的额外 options ，例如 cluster。

'options' => [

'cluster' => 'eu',

'useTLS' => true

],

当 Channels 和 Laravel Echo 一起使用时，你应该在 resources/js/bootstrap.js 文件中实例化 Echo 对象时指定 pusher 作为所需要的 broadcaster。

import Echo from "laravel-echo";

window.Pusher = require('pusher-js');

window.Echo = new Echo({

broadcaster: 'pusher',

key: 'your-pusher-channels-key'

});

//

///3/ Redis

如果你使用 Redis 广播器，请安装 Predis 库。

composer require predis/predis

Redis 广播器会使用 Redis 的 发布 / 订阅 特性来广播消息；尽管如此，你仍需将它与能够从 Redis 接收消息的 WebSocket 服务器配对使用以便将消息广播到你的 WebSocket 频道上去。

当 Redis 广播器发布一个事件的时候，该事件会被发布到它指定的频道上去，传输的数据是一个采用 JSON 编码的字符串。该字符串包含了事件名、 data 数据和生成该事件 socket ID 的用户（如果可用的话）。

//

///4/ Socket.IO

如果你想将 Redis 广播器 和 Socket.IO 服务器进行配对，你需要在你的应用程序中引入 Socket.IO JavaScript 客户端库。你可以通过 NPM 包管理器进行安装。

npm install --save socket.io-client

然后，你需要在实例化 Echo 时指定 socket.io 连接器和 host。

import Echo from "laravel-echo"

window.io = require('socket.io-client');

window.Echo = new Echo({

broadcaster: 'socket.io',

host: window.location.hostname + ':6001'

});

最后，你需要运行一个与 Laravel 兼容的 Socket.IO 服务器。 Laravel 官方并没有内置 Socket.IO 服务器实现；不过，可以选择一个由社区驱动维护的项目 tlaverdure/laravel-echo-server ( https://github.com/tlaverdure/laravel-echo-server )，目前托管在 GitHub 。

(3)、对队列的要求

在开始广播事件之前，你还需要配置和运行 队列监听器( https://learnku.com/docs/laravel/6.x/queues )。所有的事件广播都是通过队列任务来完成的，因此应用程序的响应时间不会受到明显影响。

--

2、概念综述

Laravel 的事件广播允许你使用基于驱动的 WebSockets 将服务端的 Laravel 事件广播到客户端的 JavaScript 应用程序。当前的 Laravel 自带了 Pusher ( https://pusher.com/channels ) 和 Redis 驱动。通过使用 Laravel Echo ( https://learnku.com/docs/laravel/6.x/broadcasting/5159#installing-laravel-echo ) 的 Javascript 包，我们可以很方便地在客户端消费事件。

事件通过「频道」来广播，这些频道可以被指定为公开或私有的。任何访客都可以不经授权或认证订阅一个公开频道；然而，如果想要订阅一个私有频道，那么该用户必须通过认证，并获得该频道的授权。

(1)、使用示例程序

深入了解事件广播的每个组件之前，让我们先用一个电子商务网站作为例子来概览一下。我们不会讨论配置 Pusher( https://pusher.com/channels ) 或者 Laravel Echo ( https://learnku.com/docs/laravel/6.x/broadcasting/5159#installing-laravel-echo ) 的细节，这些会在本文档的其它章节里详细讨论。

在我们的应用程序中，我们假设有一个允许用户查看订单配送状态的页面。有一个 ShippingStatusUpdated 事件会在配送状态更新时被触发。

event(new ShippingStatusUpdated($update));

//

///1/ ShouldBroadcast 接口

当用户在查看自己的订单时，我们不希望他们必须通过刷新页面才能看到状态更新。我们希望一旦有更新时就主动将更新信息广播到客户端。所以，我们必须标记 ShippingStatusUpdated 事件实现 ShouldBroadcast 接口。这会让 Laravel 在事件被触发时广播该事件。

<?php

namespace App\Events;

use Illuminate\Broadcasting\Channel;

use Illuminate\Queue\SerializesModels;

use Illuminate\Broadcasting\PrivateChannel;

use Illuminate\Broadcasting\PresenceChannel;

use Illuminate\Broadcasting\InteractsWithSockets;

use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class ShippingStatusUpdated implements ShouldBroadcast

{

/\*\*

\* Information about the shipping status update.

\*

\* @var string

\*/

public $update;

}

?>

ShouldBroadcast 接口要求事件定义一个 broadcastOn 方法。该方法负责指定事件被广播到哪些频道。在（通过 Artisan 命令）生成的事件类中，一个空的 broadcastOn 方法已经被预定义好了，所以我们只需要完成其细节即可。我们希望只有订单的创建者能够看到状态的更新，所以我们要把该事件广播到与这个订单绑定的私有频道上去。

/\*\*

\* Get the channels the event should broadcast on.

\*

\* @return \Illuminate\Broadcasting\PrivateChannel

\*/

public function broadcastOn()

{

return new PrivateChannel('order.'.$this->update->order\_id);

}

//

///2/ 授权频道

请记住，只有授权过的用户才可以收听私有频道。我们可以在 routes/channels.php 文件中，定义我们频道的授权规则。 在这个例子中，我们需要去验证任何试图收听 order.1 私有频道的用户，是否是订单实际上的创建者。

Broadcast::channel('order.{orderId}', function ($user, $orderId) {

return $user->id === Order::findOrNew($orderId)->user\_id;

});

channel 方法接收两个参数： 频道的名称和一个通过返回 true 或 false ，来表示用户是否有权收听该频道的回调函数 。

所有授权回调都将当前经过身份验证的用户作为其第一个参数，并将任何其他通配符参数作为其后续参数。 在这个例子中，我们用 {orderId} 占位符来通配表示频道名称 "ID" 的部分。

//

///3/ 监听事件广播

接下来， 剩余的工作就是在我们的 JavaScript 应用程序中监听事件。 我们可以使用 Laravel Echo 来做到这一点。首先，我们将使用 private 方法订阅私有频道。 然后，我们可以使用 listen 方法来监听 ShippingStatusUpdated 事件。 默认情况下，所有事件的公共属性都将包含在广播事件中。

Echo.private(`order.${orderId}`)

.listen('ShippingStatusUpdated', (e) => {

console.log(e.update);

});

--

3、定义广播事件

要通知 Laravel 应该广播给定事件，并在该事件上实现 Illuminate\Contracts\Broadcasting\ShouldBroadcast 接口。此接口已导入到框架生成的所有事件类中，因此你可以轻松地将其添加到任何事件中。

ShouldBroadcast 接口要求你实现一个方法： broadcastOn。 broadcastOn 方法应该返回事件应该广播的频道或频道数组。 该频道应该是 Channel，PrivateChannel，或者 PresenceChannel 的实例。Channel 的实例 代表任何用户可以订阅的公共频道， PrivateChannels 和 PresenceChannels 实列代表需要频道授权( https://learnku.com/docs/laravel/6.x/broadcasting/5159#authorizing-channels )的私有频道。

<?php

namespace App\Events;

use App\User;

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;

/\*\*

\* 新建一个新的事件实例

\*

\* @return void

\*/

public function \_\_construct(User $user)

{

$this->user = $user;

}

/\*\*

\* 获取事件应广播的频道。

\*

\* @return Channel|array

\*/

public function broadcastOn()

{

return new PrivateChannel('user.'.$this->user->id);

}

}

?>

那么， 你仅仅需要像平常那样 触发事件( https://learnku.com/docs/laravel/6.x/events )。事件一旦触发， 一个 队列任务( https://learnku.com/docs/laravel/6.x/queues ) 将通过你指定的广播驱动程序广播该事件。

(1)、广播名称

通常，Laravel 将通过事件类的名称广播该事件，但是， 你也可以通过在事件中定义一个 broadcastAs 方法，来自定义事件名称。

/\*\*

\* 广播事件名称

\*

\* @return string

\*/

public function broadcastAs()

{

return 'server.created';

}

如果你通过 broadcastAs 方法自定义广播名称， 你应该确保使用了一个前导的 . 字符，注册了你的监听器。 这将指示 Echo 不要将应用程序的命名空间添加到事件中。

.listen('.server.created', function (e) {

....

});

(2)、广播数据

当一个事件被广播时， 它的所有 public 属性都会被自动序列化并作为事件的有效负载进行广播，这允许你从 JavaScript 应用程序访问它的任何公共数据。所以，例如，如果你的事件有一个包含 Eloquent 模型的公共 $user 属性， 该事件广播载入内容将如下所示。

{

"user": {

"id": 1,

"name": "Patrick Stewart"

...

}

}

但是，如果你希望对广播载入的内容进行更细粒度的控制，可以在事件中添加 broadcastWith 方法。此方法将返回一个数据数组作为你希望广播事件所载入的内容。

/\*\*

\* 获取广播数据

\*

\* @return array

\*/

public function broadcastWith()

{

return ['id' => $this->user->id];

}

(3)、广播队列

通常， 在你的 queue.php 配置文件中， 每个广播事件都放在指定默认队列连接的默认队列中。你可以在事件类中定义一个 broadcastQueue 属性来自定义广播的队列。此属性应指定广播时你要使用队列的名称。

/\*\*

\* 要放置事件的队列的名称。

\*

\* @var string

\*/

public $broadcastQueue = 'your-queue-name';

假如你要使用 sync 队列驱动代替默认的队列驱动来广播事件，你可以通过继承 ShouldBroadcastNow 接口来替代 ShouldBroadcast 接口。

<?php

use Illuminate\Contracts\Broadcasting\ShouldBroadcastNow;

class ShippingStatusUpdated implements ShouldBroadcastNow

{

//

}

?>

(4)、广播条件

有时候你需要在给定条件为 true 的时候才广播事件，你可以在事件类中增加一个 broadcastWhen 方法去定义这些条件。

/\*\*

\* 确定事件是否要被广播

\*

\* @return bool

\*/

public function broadcastWhen()

{

return $this->value > 100;

}

--

4、授权频道

私人频道往往需要授权验证过的用户才可以实际收听该频道。这是通过使用频道名称向你的 Laravel 应用程序发出 HTTP 请求，并允许你的应用程序确定用户是否可以收听该频道来实现的。使用 Laravel Echo( https://learnku.com/docs/laravel/6.x/broadcasting/5159#installing-laravel-echo ) 时， 将自动进行授权订阅私人频道的 HTTP 请求；但是，你需要定义响应这些请求的正确路由。

(1)、定义授权路由

庆幸的是，Laravel 可以轻松定义响应频道授权请求的路由。在 Laravel 应用程序附带的 BroadcastServiceProvider 中，你将看到对 Broadcast :: routes 方法的调用。 此方法将注册 /broadcasting/auth 路由去处理授权请求。

Broadcast::routes();

Broadcast::routes 方法将自动将其路由放在 web 中间件组中； 但是，如果要自定义指定的属性，可以将路径属性数组传递给方法。

Broadcast::routes($attributes);

//

///1/ 自定义授权端点

通常， Echo 会使用 /broadcasting/auth 端点来授权频道访问。 但是，你可以通过将 authEndpoint 配置选项传递给 Echo 实例来指定自己的授权端点。

window.Echo = new Echo({

broadcaster: 'pusher',

key: 'your-pusher-channels-key',

authEndpoint: '/custom/endpoint/auth'

});

(2)、定义授权回调

接下来，我们需要定义实际执行频道授权的逻辑。这是在你的应用程序附带的 routes/channels.php 文件中完成的 。 在这个文件中，你可以使用 Broadcast::channel 方法注册频道授权回调。

Broadcast::channel('order.{orderId}', function ($user, $orderId) {

return $user->id === Order::findOrNew($orderId)->user\_id;

});

channel 方法接受两个参数：频道的名称和返回 true 或 false 来指示用户是否有权在频道上进行侦听的回调。

所有授权回调都将当前经过身份验证的用户作为其第一个参数，并将任何其他通配符参数作为其后续参数。 在这个例子中，我们用 {orderId} 占位符来通配表示频道名称 "ID" 的部分。

//

///1/ 授权回调模型绑定

就像 HTTP 路由一样， 频道路由可以利用隐式和显式 路由模型绑定.。例如，你可以请求一个实际的 Order 模型实列来替代接收一个字符串或者订单 ID 数字。

use App\Order;

Broadcast::channel('order.{order}', function ($user, Order $order) {

return $user->id === $order->user\_id;

});

//

///2/ 授权回调验证

私有和在线广播频道，通过应用程序默认的认证防护器来认证当前用户。如果用户未经过认证，则会自动拒绝频道授权，并且永远不会执行授权回调。 但是，你可以分配多个自定义防护器，以便在必要时对传入请求进行身份验证。

Broadcast::channel('channel', function() {

// ...

}, ['guards' => ['web', 'admin']]);

(3)、定义频道类

///1/ 创建频道类

如果你的应用程序使用了很多不同的频道， 那么 routes/channels.php 文件将会变得很臃肿。 所以，你可以使用频道类，而不是使用闭包来授权频道。要生成频道类，使用 make:channel

Artisan 命令。 此命令将在 App/Broadcasting 目录，生成一个新的频道类。

php artisan make:channel OrderChannel

//

///2/ 注册频道

接下来，在 routes/channels.php 文件中注册你的频道。

use App\Broadcasting\OrderChannel;

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

//

///3/ 授权逻辑

最后，你可以把频道的授权逻辑，放在频道类的 join 方法中。join 方法将保留在频道授权闭包中的逻辑。 你还可以利用频道模型绑定。

<?php

namespace App\Broadcasting;

use App\User;

use App\Order;

class OrderChannel

{

/\*\*

\*新建一个新的频道实例

\*

\* @return void

\*/

public function \_\_construct()

{

//

}

/\*\*

\* 验证用户对频道的访问权限。

\*

\* @param \App\User $user

\* @param \App\Order $order

\* @return array|bool

\*/

public function join(User $user, Order $order)

{

return $user->id === $order->user\_id;

}

}

?>

{提示} 和很多 Laravel 中其他类一样，频道类会被 服务容器( https://learnku.com/docs/laravel/6.x/container ) 自动解析。 所以，你可以在构造函数中键入频道所需的任何依赖。

--

5、广播事件

一旦你定义了一个事件并用 ShouldBroadcast 接口标记它，你只需要使用 event 函数来触发事件。 事件调度程序将注意到事件标记为 ShouldBroadcast 接口，并将事件排队以进行广播。

event(new ShippingStatusUpdated($update));

(1)、只广播给他人

在构建利用事件广播的应用程序时，你可以使用 broadcast 函数替换 event 函数。 与 event 函数一样，broadcast 函数将事件调度到服务器端侦听器。

broadcast(new ShippingStatusUpdated($update));

但是，broadcast 函数还开放了 toOthers 方法，该方法允许你从广播的收件人中排除当前用户。

broadcast(new ShippingStatusUpdated($update))->toOthers();

为了更好地理解何时可能需要使用 toOthers 方法，让我们设想一个任务列表应用程序，用户可以通过输入任务名称来创建新任务。 要创建任务，你的应用程序可能会向 /task 端点发出请求，该端点广播任务的创建并返回 JSON 格式的新任务。 当你的 JavaScript 应用程序从端点收到响应时，它可能会直接将新任务插入其任务列表中，如下所示。

axios.post('/task', task)

.then((response) => {

this.tasks.push(response.data);

});

但是，请记住，我们还广播任务的创建。 如果你的 JavaScript 应用程序正在收听此事件以便将任务添加到任务列表，则列表中将出现重复的任务：一个来自端点，另一个来自广播。 你可以使用 toOthers 方法来指示广播器不向当前用户广播事件。

{注意} 你的事件必须使用 Illuminate\Broadcasting\InteractsWithSockets trait 才能调用 toOthers 方法。

//

///1/ 配置

初始化 Laravel Echo 实例时，会为连接分配套接字 ID 。 如果你使用 Vue( https://vuejs.org/ ) 和 Axios( https://github.com/mzabriskie/axios )，套接字 ID 将自动作为 X-Socket-ID 标题 附加到每个传出请求。 然后，当你调用 toOthers 方法时，Laravel 将从标头中提取套接字 ID，并指示广播器不要广播到具有该套接字 ID 的任何连接。

如果你不使用 Vue 和 Axios ，则需要手动配置 JavaScript 应用程序以发送 X-Socket-ID 头。 你可以使用 Echo.socketId 方法检索套接字 ID。

var socketId = Echo.socketId();

--

6、接收广播

(1)、安装 Laravel Echo

Laravel Echo 是一个 JavaScript 库，可以轻松订阅频道并收听 Laravel 广播的事件。 你可以通过 NPM 包管理器安装 Echo。 在这个例子中，我们还将安装 pusher-js 包，因为我们将使用 Pusher Channels 广播器。

npm install --save laravel-echo pusher-js

安装 Echo 后，你就可以在应用程序的 JavaScript 中创建一个全新的 Echo 实例。 一个好的实现方式是将它放在 Laravel 框架附带的 resources/js/bootstrap.js 文件的底部。

import Echo from "laravel-echo";

window.Echo = new Echo({

broadcaster: 'pusher',

key: 'your-pusher-channels-key'

});

在创建使用 pusher 连接器的 Echo 实例时，你还可以指定 cluster 是否必须通过 TLS 进行连接（默认情况下，当 forceTLS 为 false 时，如果页面是通过 HTTP 加载的，则会生成非 TLS 连接或者作为 TLS 连接失败时的回调）。

window.Echo = new Echo({

broadcaster: 'pusher',

key: 'your-pusher-channels-key',

cluster: 'eu',

forceTLS: true

});

//

///1/ 使用现有客户端实例

如果你已经有 Echo 使用的 Pusher Channels 或 Socket.io 客户端实例，你可以通过 client 配置选项将它传递给 Echo。

const client = require('pusher-js');

window.Echo = new Echo({

broadcaster: 'pusher',

key: 'your-pusher-channels-key',

client: client

});

--

(2)、对事件进行监听

一旦安装并实例化了 Echo，就可以开始监听事件广播了。 首先，使用 channel 方法检索频道的实例，然后调用 listen 方法来监听指定的事件。

Echo.channel('orders')

.listen('OrderShipped', (e) => {

console.log(e.order.name);

});

如果想在私有频道上收听事件，请使用 private 方法。 可以通过链式调用 listen 方法，在单个频道上侦听多个事件。

Echo.private('orders')

.listen(...)

.listen(...)

.listen(...);

(3)、退出频道

要退出频道，你可以在 Echo 实例上调用 leaveChannel 方法。

Echo.leaveChannel('orders');

//

如果你想推出一个频道以及相关的私有和在线频道，你可以调用 leave 方法。

Echo.leave('orders');

(4)、命名空间

你可能已经在上面的示例中，注意到我们没有为事件类指定完整的命名空间。 这是因为 Echo 会自动假设事件位于 App\Events 命名空间中。 但是，你可以通过传递 namespace 配置选项来实例化 Echo 时配置根命名空间。

window.Echo = new Echo({

broadcaster: 'pusher',

key: 'your-pusher-channels-key',

namespace: 'App.Other.Namespace'

});

或者，你可以在使用 Echo 订阅事件类时使用 . 作为前缀。 这将允许你始终指定完全限定的类名。

Echo.channel('orders')

.listen('.Namespace\\Event\\Class', (e) => {

//

});

--

7、Presence 频道

Presence 频道建立在私有频道安全性的基础上，同时暴露了谁订阅频道的附加特征。 这样可以轻松构建功能强大的协作应用程序功能，例如在其他用户查看同一页面时通知用户。

(1)、授权 Presence 频道

所有在线频道也是私有频道；因此，用户必须 被授权才能访问( https://learnku.com/docs/laravel/6.x/broadcasting/5159#authorizing-channels )。但是，在为 Presence 频道定义授权回调时，如果用户有权加入该频道，则不会返回 true。相反，你应该返回有关用户的数据数组。

授权回调返回的数据将可供 JavaScript 应用程序中的 Presence 频道事件侦听器使用。 如果用户未被授权加入 Presence 频道，则应返回 false 或 null。

Broadcast::channel('chat.{roomId}', function ($user, $roomId) {

if ($user->canJoinRoom($roomId)) {

return ['id' => $user->id, 'name' => $user->name];

}

});

(2)、加入 Presence 频道

加入 Presence 频道，你可以使用 Echo 的 join 方法。 join 方法将返回一个 PresenceChannel 实现，它与 listen 方法一起展示，允许你订阅 here，join 和 leaving 事件。

Echo.join(`chat.${roomId}`)

.here((users) => {

//

})

.joining((user) => {

console.log(user.name);

})

.leaving((user) => {

console.log(user.name);

});

一旦成功加入频道，将立即执行 here 回调，并且将接收包含当前订阅该频道的所有其他用户的用户信息的数组。 当新用户加入频道时，将执行 join 方法，而当用户离开频道时，将执行 leaving 方法。

(3)、广播到 Presence 频道

Presence 频道可以像公共或私私有频道一样接收事件。 使用聊天室的示例，我们可能希望将 NewMessage 事件广播到房间的 Presence 频道。 为此，我们将从事件的 broadcastOn 方法返回一个 PresenceChannel 实例。

/\*\*

\* 获取事件广播的频道。

\*

\* @return Channel|array

\*/

public function broadcastOn()

{

return new PresenceChannel('room.'.$this->message->room\_id);

}

//

与公共或私有事件一样，可以使用广播功能来广播 Presence 频道事件。 与其他事件一样，你可以使用 toOthers 方法排除当前用户接收广播。

broadcast(new NewMessage($message));

broadcast(new NewMessage($message))->toOthers();

//

你可以通过 Echo 的 listen 方法监听 join 事件。

Echo.join(`chat.${roomId}`)

.here(...)

.joining(...)

.leaving(...)

.listen('NewMessage', (e) => {

//

});

--

8、客户端事件

{提示} 使用 Pusher 频道( https://pusher.com/channels )时， 必须在应用程序仪表板( https://dashboard.pusher.com/ )的应用程序设置部分启用客户端事件选项以便发送客户端事件。

有时，你可能希望将事件广播到其他连接的客户端，而根本不需要使用 Laravel 应用程序。这对于诸如键入通知之类的内容特别有用，在这种情况下，你希望提醒应用程序的用户另一个用户正在给定屏幕上键入消息。

//

//// 广播事件

要广播客户端事件，可以使用 Echo 的 whisper 方法。

Echo.private('chat')

.whisper('typing', {

name: this.user.name

});

//

//// 监听事件

要监听客户端事件，可以使用 listenForWhisper 方法。

Echo.private('chat')

.listenForWhisper('typing', (e) => {

console.log(e.name);

});

--

9、消息通知

通过将事件广播与通知( https://learnku.com/docs/laravel/6.x/notifications )配对，你的 JavaScript 应用程序可以在发生时收到新通知，而无需刷新页面。 首先，请务必阅读有关使用广播通知频道( https://learnku.com/docs/laravel/6.x/notifications#broadcast-notifications )的文档。

配置通知以使用广播频道后，你可以使用 Echo 的 notification 方法监听广播事件。 请记住，频道名称应与接收通知的实体的类名相匹配。

Echo.private(`App.User.${userId}`)

.notification((notification) => {

console.log(notification.type);

});

在此示例中，回调将接收通过 广播 频道发送到 App\User 实例的所有通知。 App.User.{id} 频道的频道授权回调包含在 Laravel 框架附带的默认 BroadcastServiceProvider 中。

--

--

三、缓存系统

--

https://learnku.com/docs/laravel/6.x/cache/5160

--

1、配置

Laravel 为各种后端缓存提供了丰富而统一的 API，其配置信息位于 config/cache.php 文件中。在该文件中你可以指定应用默认使用哪个缓存驱动。Laravel 支持当前流行的后端缓存，例如 Memcached ( https://memcached.org/ ) 和 Redis ( https://redis.io/ )。

缓存配置文件还包含各种其他选项，这些选项都记录在文件中，因此请确保阅读这些选项。默认情况下，Laravel 配置为使用 file 缓存驱动，它将序列化的缓存对象存储在文件系统中。对于较大型应用，建议使用更强大的驱动程序，例如 Memcached 或 Redis。你甚至可以为同一个驱动程序配置多个缓存配置。

(1)、驱动的前提条件

///1/ 数据库

当使用 database 缓存驱动时，你需要配置一个表来存放缓存数据。下面是构建缓存数据表结构的 Schema 声明示例。

Schema::create('cache', function ($table) {

$table->string('key')->unique();

$table->text('value');

$table->integer('expiration');

});

提示：你也可以使用 Artisan 命令 php artisan cache:table 来生成合适的迁移。

//

///2/ Memcached

使用 Memcached 驱动需要安装 Memcached PECL 扩展包 。你可以把所有的 Memcached 服务器都列在 config/cache.php ( https://pecl.php.net/package/memcached ) 配置文件中。

'memcached' => [

[

'host' => '127.0.0.1',

'port' => 11211,

'weight' => 100

],

],

你还可以将 host 选项设置为 UNIX socket 路径。如果你这样设置了， port 选项应该设置为 0。

'memcached' => [

[

'host' => '/var/run/memcached/memcached.sock',

'port' => 0,

'weight' => 100

],

],

//

///3/ Redis

在使用 Laravel 的 Redis 缓存之前，你需要通过 PECL 安装 PhpRedis PHP 扩展，或者通过 Composer 安装 predis/predis 包（~1.0）。

如需了解更多关于 Redis 的配置，请参考 Laravel Redis 文档( https://learnku.com/docs/laravel/6.x/redis#configuration )。

--

2、缓存的使用

(1)、获取缓存实例

Illuminate\Contracts\Cache\Factory 和 Illuminate\Contracts\Cache\Repository 契约( https://learnku.com/docs/laravel/6.x/contracts ) 提供了 Laravel 缓存服务的访问机制。Factory 契约为你的应用程序定义了访问所有缓存驱动的机制。 Repository 契约通常是由你的 cache 配置文件指定的默认缓存驱动实现的。

当然，你也可以使用 Cache Facade，我们将在后续的文档中介绍。Cache Facade 为 Laravel 缓存契约底层的实现提供了方便又简洁的方法。

<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

class UserController extends Controller

{

/\*\*

\* 展示应用的所有用户列表

\*

\* @return Response

\*/

public function index()

{

// 缓存：获取数据；

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

}

}

?>

//

///1/ 访问多个缓存存储

使用 Cache Facade，你可以通过 store 方法来访问各种缓存存储。传入 store 方法的键应该对应 cache 配置信息文件中的 stores 配置数组中所列出的一个。

$value = Cache::store('file')->get('foo');

Cache::store('redis')->put('bar', 'baz', 600); // 10 分钟

(2)、从缓存中获取数据

Cache Facade 的 get 方法用于从缓存中获取数据。如果该数据在缓存中不存在，那么该方法将返回 null 。正如你想的那样，你也可以向 get 方法传递第二个参数，用来指定如果查找的数据不存在时你希望返回的默认值。

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

$value = Cache::get('key', 'default');

你甚至可以传递一个 Closure 作为默认值。如果指定的数据在缓存中不存在，将返回 Closure 的结果。传递闭包的方法允许你从数据库或其他外部服务中获取默认值。

$value = Cache::get('key', function () {

return DB::table(...)->get();

});

//

///2/ 检查缓存项是否存在

has 方法可以用于判断缓存项是否存在。如果值为 null，则该方法将会返回 false。

if (Cache::has('key')) {

//

}

//

///3/ 递增与递减值

increment 和 decrement 方法可以用来调整缓存中整数项的值。这两个方法都可以传入第二个可选参数，这个参数用来指明要递增或递减的数量。

Cache::increment('key');

Cache::increment('key', $amount);

Cache::decrement('key');

Cache::decrement('key', $amount);

//

///4/ 获取和存储

有时你可能想从缓存中获取一个数据，而当请求的缓存项不存在时，程序能为你存储一个默认值。例如，你可能想从缓存中获取所有用户，当缓存中不存在这些用户时，程序将从数据库将这些用户取出并放入缓存。你可以使用 Cache::remember 方法来实现。

$value = Cache::remember('users', $seconds, function () {

return DB::table('users')->get();

});

如果缓存中不存在你想要的数据时，则传递给 remember 方法的 闭包 将被执行，然后将其结果返回并放置到缓存中。

你可以使用 rememberForever 方法从缓存中获取数据或者永久存储它。

$value = Cache::rememberForever('users', function () {

return DB::table('users')->get();

});

//

///5/ 获取和删除

如果你需要从缓存中获取到数据之后再删除它，你可以使用 pull 方法。和 get 方法一样，如果缓存不存在，则返回 null。

$value = Cache::pull('key');

(3)、在缓存中存储数据

你可以使用 Cache Facade 的 put 方法将数据存储到缓存中。

Cache::put('key', 'value', $seconds);

//

如果缓存的过期时间没有传递给 put 方法， 则缓存将永久有效。

Cache::put('key', 'value');

//

除了以整数形式传递过期时间的秒数，你还可以传递一个 DateTime 实例来表示该数据的到期时间。

Cache::put('key', 'value', now()->addMinutes(10));

//

///1/ 只存储没有的数据

add 方法将只存储缓存中不存在的数据。如果存储成功，将返回 true ，否则返回 false。

Cache::add('key', 'value', $seconds);

//

///2/ 数据永久存储

forever 方法可用于持久化将数据存储到缓存中。因为这些数据不会过期，所以必须通过 forget 方法从缓存中手动删除它们。

Cache::forever('key', 'value');

提示：如果你使用 Memcached 驱动，当缓存数据达到存储上限时，「永久存储」 的数据可能会被删除。

(4)、从缓存中删除数据

你可以使用 forget 方法从缓存中删除这些数据。

Cache::forget('key');

//

你也可以通过提供零或者负的 TTL 值来删除这些数据。

Cache::put('key', 'value', 0);

Cache::put('key', 'value', -5);

//

你可以使用 flush 方法清空所有的缓存。

Cache::flush();

注意：清空缓存的方法并不会考虑缓存前缀，会将缓存中的所有内容删除。因此在清除与其它应用程序共享的缓存时，请慎重考虑。

(5)、原子锁

注意：要想使用该特性，你的应用必须使用 memcached，dynamodb 或 redis 缓存驱动作为你应用的默认缓存驱动。此外，所有服务器必须与同一中央缓存服务器进行通信。

原子锁允许对分布式锁进行操作而不必担心竞争条件。例如，Laravel Forge ( https://forge.laravel.com/ ) 使用原子锁来确保在一台服务器上每次只有一个远程任务在执行。你可以使用 Cache::lock 方法来创建和管理锁。

use Illuminate\Support\Facades\Cache;

$lock = Cache::lock('foo', 10);

if ($lock->get()) {

// 获取锁定10秒...

//

$lock->release();

}

//

get 方法也可以接收一个闭包。在闭包执行之后，Laravel 将会自动释放锁。

Cache::lock('foo')->get(function () {

// 获取无限期锁并自动释放...

});

//

如果你在请求时锁无法使用，你可以控制 Laravel 等待指定的秒数。如果在指定的时间限制内无法获取锁，则会抛出 Illuminate\Contracts\Cache\LockTimeoutException。

use Illuminate\Contracts\Cache\LockTimeoutException;

$lock = Cache::lock('foo', 10);

try {

$lock->block(5);

// 等待最多5秒后获取的锁...

} catch (LockTimeoutException $e) {

// 无法获取锁...

} finally {

optional($lock)->release();

}

//

Cache::lock('foo', 10)->block(5, function () {

// 等待最多5秒后获取的锁...

});

//

///1/ 管理跨进程锁

有时候，你可能希望在一个进程中获取锁然后在另一个进程中释放它。例如，你可能在一个 Web 请求中获取锁 并且希望在由该请求触发的队列任务结束时释放锁。对于这样的场景，你应该把传递锁的作用域 「owner token」传递给队列任务，使队列任务可以使用给定的 token 重新实例化锁。

// 控制器里面...

$podcast = Podcast::find($id);

$lock = Cache::lock('foo', 120);

if ($result = $lock->get()) {

ProcessPodcast::dispatch($podcast, $lock->owner());

}

// ProcessPodcast Job 里面...

Cache::restoreLock('foo', $this->owner)->release();

//

如果你想在无视当前锁的所有者的情况下释放锁，你可以使用 forceRelease 方法。

Cache::lock('foo')->forceRelease();

(6)、Cache 辅助函数

除了可以使用 Cache 的门面以及 Cache 契约 ( https://learnku.com/docs/laravel/6.x/contracts ) 外，你也可以使用全局辅助函数 Cache 来获取和保存缓存数据。当只使用一个字符串参数调用 Cache 函数时，这将返回给定键对应的值。

$value = cache('key');

//

如果你向函数提供了一组带有过期时间的键值对，那么在这段时间内，它将缓存此数据。

cache(['key' => 'value'], $seconds);

cache(['key' => 'value'], now()->addMinutes(10));

//

当 Cache 函数在没有任何参数的情况下被调用，那么它返回的将是一个实现 Illuminate\Contracts\Cache\Factory 的实例，并且允许你调用其他的缓存方法。

cache()->remember('users', $seconds, function () {

return DB::table('users')->get();

});

Tip：如果在测试中使用全局函数 Cache ，你可以使用 Cache::shouldReceive 方法就像 测试 Facade( https://learnku.com/docs/laravel/6.x/mocking#mocking-facades )。

--

3、缓存标记

{注意} 当使用「file」或者 database 缓存驱动时，缓存标记将不支持。此外，当使用带有「永久」存储的缓存的多个缓存标记时，类似 Memcached 的缓存驱动性能最佳，它会自动清除旧的数据。

(1)、存储被标记的缓存数据 (存储被打上标签的缓存数据)

缓存标签允许你给相关的缓存标签项打上同一个标签以便后续可以清除这些缓存值。你可以通过传入一个被排序的标签数组来访问这些缓存项。例如，我们可以在使用标签的同时使用 put 方法来设置缓存。

Cache::tags(['people', 'artists'])->put('John', $john, $seconds);

Cache::tags(['people', 'authors'])->put('Anne', $anne, $seconds);

(2)、访问被标记的缓存数据 (访问被打上标签的缓存数据)

若要获取一个被打上标签的缓存数据，将相同标签的有序数组传递给 tags 方法，然后调用 get 方法检索你要获取的键。

$john = Cache::tags(['people', 'artists'])->get('John');

$anne = Cache::tags(['people', 'authors'])->get('Anne');

(3)、删除被标记的缓存数据 (移除带有标签的缓存数据)

你可能需要移除单个标签或者一组标签所标记的所有缓存数据，例如，下面这个例子将会移除带有 people, authors 或者两者都有的标签的所有缓存数据，所以 Anne 和 John 将会从缓存中删除。

Cache::tags(['people', 'authors'])->flush();

//

相反，下面的例子只会移除带有标签 authors 的缓存数据，因此 Anne 缓存数据将会被移除， 但是 John 就不会。

Cache::tags('authors')->flush();

--

4、增加自定义缓存驱动

(1)、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 contract ( https://learnku.com/docs/laravel/6.x/contracts ) 契约。因此， MongoDB 缓存实现看起来就像是这样。

//

///1/ MongoDB 缓存启动定义

<?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, $seconds) {}

public function putMany(array $values, $seconds) {}

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 MongoDB 的连接来实现这些方法。 关于如何实现这些方法中的实例，可以通过阅读源代码 Illuminate\Cache\MemcachedStore 来加以理解。 一旦我们完成契约接口的实现，我们就可以通过以下的示例来完成自定义驱动的注册了。

//

///2/ 自定义驱动的注册

Cache::extend('mongo', function ($app) {

return Cache::repository(new MongoStore);

});

Tip：如果你不知道该将缓存驱动的代码放在什么地方，你可以在你的 app 文件夹下创建一个 Extensions 的命名空间。 值得注意的是，Laravel 并没有硬性规定应用程序的结构。因此你可以根据自己的喜好自由的组织你的应用程序。

(2)、注册驱动

在 Laravel 注册一个自定义的缓存驱动，我们需要在 Cache 门面上使用 extend 方法。 对 Cache::extend 的调用可以在新的 Laravel 应用程序中自带的 App\Providers\AppServiceProvider 的 boot 方法中完成，或者你也可以自己创建服务提供者来存放扩展，只是不要忘记在 config/app.php 的 provider 的数组中注册服务提供者。

<?php

namespace App\Providers;

use App\Extensions\MongoStore;

use Illuminate\Support\Facades\Cache;

use Illuminate\Support\ServiceProvider;

class CacheServiceProvider extends ServiceProvider

{

/\*\*

\* 在容器中注册绑定

\*

\* @return void

\*/

public function register()

{

//

}

/\*\*

\* 执行服务注册后的引导

\*

\* @return void

\*/

public function boot()

{

Cache::extend('mongo', function ($app) {

return Cache::repository(new MongoStore);

});

}

}

?>

传递给 extend 方法的第一个参数是驱动程序的名称。 该值对应 config/cache.php 配置文件中的 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 的实例， 也就是 服务容器( https://learnku.com/docs/laravel/6.x/container ) 的一个实例。

一旦你的扩展被注册以后，只需要更新配置文件 config/cache.php 的 driver 选项作为自定义扩展名称即可。

--

5、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( https://learnku.com/docs/laravel/6.x/events )。 通常，你应该将这些事件监听器放在 EventServiceProvider 中。

/\*\*

\* 应用程序的事件监听器映射

\*

\* @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',

],

];

--

--

四、集合 (Collections)

--

https://learnku.com/docs/laravel/6.x/collections/5161

--

1、简介

Illuminate\Support\Collection 类提供了一个更具可读性和更便于处理数组数据的封装。具体例子请查看下面代码。我们使用 collect 辅助函数从数组中创建一个新的集合实例，对其中每一个元素执行 strtoupper 函数之后再删除所有的空元素。

$collection = collect(['taylor', 'abigail', null])->map(function ($name) {

return strtoupper($name);

})

->reject(function ($name) {

return empty($name);

});

正如你所见，Collection 类允许你链式调用其它方法，以达到在底层数组上流畅的执行 map 和 reduce 操作，通常，集合是不可变的，这意味着每一个 Collection 方法都会返回一个新的 Collection 实例。

(1)、创建集合

如上所述， collect 辅助函数会为指定的数组返回一个新的 Illuminate\Support\Collection 实例。因此，我们可以这样轻松的创建一个集合。

$collection = collect([1, 2, 3]);

Tip：通常，Eloquent( https://learnku.com/docs/laravel/6.x/eloquent ) 查询的结果返回的内容都是 Collection 实例。

(2)、扩展集合

集合都是「可宏扩展」(macroable) 的，它允许你在执行时将其它方法添加到 Collection 类。例如，通过下面的代码在 Collection 类中添加一个 toUpper 方法。

use Illuminate\Support\Str;

Collection::macro('toUpper', function () {

return $this->map(function ($value) {

return Str::upper($value);

});

});

$collection = collect(['first', 'second']);

$upper = $collection->toUpper();

// ['FIRST', 'SECOND']

通常，你应该在 服务提供者( https://learnku.com/docs/laravel/6.x/providers ) 内声明集合宏。

--

2、可用方法

接下来的文档内容，我们会探讨 Collection 类的每个方法。请牢记，所有方法都可以通过链式访问的形式优雅的操作数组。而且，几乎所有的方法都会返回一个新的 Collection 实例，允许你在必要时保存集合的原始副本。

(1)、all()

all 方法返回该集合表示的底层数组。

collect([1, 2, 3])->all(); // [1, 2, 3]

(2)、average()

avg 方法的别名。

(3)、avg()

avg 方法返回给定键的 平均值( https://en.wikipedia.org/wiki/Average )。

$average = collect([['foo' => 10], ['foo' => 10], ['foo' => 20], ['foo' => 40]])->avg('foo'); // 20

$average = collect([1, 1, 2, 4])->avg(); // 2

(4)、chunk()

chunk 方法将集合拆成多个给定大小的小集合。

$collection = collect([1, 2, 3, 4, 5, 6, 7]);

$chunks = $collection->chunk(4);

$chunks->toArray(); // [[1, 2, 3, 4], [5, 6, 7]]

这个方法在使用网格系统的 视图( https://learnku.com/docs/laravel/6.x/views ) 中特别适用，例如 Bootstrap ( https://getbootstrap.com/docs/4.1/layout/grid/ )。想象你有一个 Eloquent ( https://learnku.com/docs/laravel/6.x/eloquent ) 模型的集合要在网格中显示。

@foreach ($products->chunk(3) as $chunk)

<div class="row">

@foreach ($chunk as $product)

<div class="col-xs-4">{{ $product->name }}</div>

@endforeach

</div>

@endforeach

(5)、collapse()

collapse 方法将多个数组的集合合并成一个数组的集合。

$collection = collect([[1, 2, 3], [4, 5, 6], [7, 8, 9]]);

$collapsed = $collection->collapse();

$collapsed->all(); // [1, 2, 3, 4, 5, 6, 7, 8, 9]

(6)、combine()

combine 方法可以将一个集合的值作为键，再将另一个数组或集合的值作为值合并成一个集合。

$collection = collect(['name', 'age']);

$combined = $collection->combine(['George', 29]);

$combined->all(); // ['name' => 'George', 'age' => 29]

(7)、collect()

collect 方法返回一个新的 Collection 实例，其中包含当前集合中的项目。

$collectionA = collect([1, 2, 3]);

$collectionB = $collectionA->collect();

$collectionB->all(); // [1, 2, 3]

//

collect 方法主要用于将 惰性集合( https://learnku.com/docs/laravel/6.x/collections/5161#lazy-collections ) 转换为标准 Collection 实例。

$lazyCollection = LazyCollection::make(function () {

yield 1;

yield 2;

yield 3;

});

$collection = $lazyCollection->collect();

get\_class($collection); // 'Illuminate\Support\Collection'

$collection->all(); // [1, 2, 3]

Tip：当您有一个 Enumerable 实例并且需要一个非惰性集合实例时，collect 方法特别有用。由于 collect() 是 Enumerable 契约的一部分，因此您可以安全地使用它来获取 Collection 实例。

(8)、concat()

concat 方法将给定的 数组 或集合值追加到集合的末尾。

$collection = collect(['John Doe']);

$concatenated = $collection->concat(['Jane Doe'])->concat(['name' => 'Johnny Doe']);

$concatenated->all(); // ['John Doe', 'Jane Doe', 'Johnny Doe']

(9)、contains()

contains 方法判断集合是否包含指定的集合项。

$collection = collect(['name' => 'Desk', 'price' => 100]);

$collection->contains('Desk'); // true

$collection->contains('New York'); // false

//

你也可以使用 contains 方法传递一组键 / 值对，可以判断该键 / 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，你也可以用 contains 方法传递一个回调函数来执行自己的真实测试。

$collection = collect([1, 2, 3, 4, 5]);

$collection->contains(function ($value, $key) {

return $value > 5;

}); // false

contains 方法在检查集合项的值时使用「宽松」比较，这意味着具有整数值的字符串将被视为等于相同值的整数。相反 containsStrict 方法则是使用「严格」比较进行过滤。

(10)、containsStrict()

这个方法和 contains 方法类似，但是它却是使用了「严格」比较来比较所有的值。

(11)、count()

count 方法返回这个集合内集合项的总数量。

$collection = collect([1, 2, 3, 4]);

$collection->count(); // 4

(12)、countBy()

countBy 方法计算集合中每个值的出现次数。默认情况下，该方法计算每个元素的出现次数。

$collection = collect([1, 2, 2, 2, 3]);

$counted = $collection->countBy();

$counted->all(); // [1 => 1, 2 => 3, 3 => 1]

//

但是，你也可以传递一个回调函数给 countBy 来计算自定义的值出现的次数。

$collection = collect(['alice@gmail.com', 'bob@yahoo.com', 'carlos@gmail.com']);

$counted = $collection->countBy(function ($email) {

return substr(strrchr($email, "@"), 1);

});

$counted->all(); // ['gmail.com' => 2, 'yahoo.com' => 1]

(13)、crossJoin

crossJoin 方法交叉连接指定数组或集合的值，返回所有可能排列的笛卡尔积。

$collection = collect([1, 2]);

$matrix = $collection->crossJoin(['a', 'b']);

$matrix->all();

/\*

[

[1, 'a'],

[1, 'b'],

[2, 'a'],

[2, 'b'],

]

\*/

$collection = collect([1, 2]);

$matrix = $collection->crossJoin(['a', 'b'], ['I', 'II']);

$matrix->all();

/\*

[

[1, 'a', 'I'],

[1, 'a', 'II'],

[1, 'b', 'I'],

[1, 'b', 'II'],

[2, 'a', 'I'],

[2, 'a', 'II'],

[2, 'b', 'I'],

[2, 'b', 'II'],

]

\*/

(14)、dd()

$collection = collect(['John Doe', 'Jane Doe']);

$collection->dd();

/\*

Collection {

#items: array:2 [

0 => "John Doe"

1 => "Jane Doe"

]

}

\*/

如果你不想中断执行脚本，请使用 dump 方法替代。

(15)、diff()

diff 方法将集合与其它集合或者 PHP 数组 进行值的比较。然后返回原集合中存在而指定集合中不存在的值。

$collection = collect([1, 2, 3, 4, 5]);

$diff = $collection->diff([2, 4, 6, 8]);

$diff->all(); // [1, 3, 5]

(16)、diffAssoc()

diffAssoc 方法与另外一个集合或基于它的键和值的 PHP 数组 进行比较。这个方法将会返回原集合不存在于指定集合的键 / 值对。

$collection = collect([

'color' => 'orange',

'type' => 'fruit',

'remain' => 6

]);

$diff = $collection->diffAssoc([

'color' => 'yellow',

'type' => 'fruit',

'remain' => 3,

'used' => 6,

]);

$diff->all(); // ['color' => 'orange', 'remain' => 6]

(17)、diffKeys()

diffKeys 方法和另外一个集合或 PHP 数组 的键进行比较，然后返回原集合中存在而指定集合中不存在键所对应的键 / 值对。

$collection = collect([

'one' => 10,

'two' => 20,

'three' => 30,

'four' => 40,

'five' => 50,

]);

$diff = $collection->diffKeys([

'two' => 2,

'four' => 4,

'six' => 6,

'eight' => 8,

]);

$diff->all(); // ['one' => 10, 'three' => 30, 'five' => 50]

(18)、dump()

dump 方法用于打印集合项。

$collection = collect(['John Doe', 'Jane Doe']);

$collection->dump();

/\*

Collection {

#items: array:2 [

0 => "John Doe"

1 => "Jane Doe"

]

}

\*/

如果要在打印集合后终止执行脚本，请使用 dd 方法代替。

(19)、duplicates()

duplicates 方法从集合中检索并返回重复的值。

$collection = collect(['a', 'b', 'a', 'c', 'b']);

$collection->duplicates(); // [2 => 'a', 4 => 'b']

//

如果集合包含数组或对象，则可以传递希望检查重复值的属性的键。

$employees = collect([

['email' => 'abigail@example.com', 'position' => 'Developer'],

['email' => 'james@example.com', 'position' => 'Designer'],

['email' => 'victoria@example.com', 'position' => 'Developer'],

])

$employees->duplicates('position'); // [2 => 'Developer']

(20)、duplicatesStrict()

此方法与 duplicates 方法具有相同的签名；但是，所有值都以「严格」的方式进行比较。

(21)、each()

each 方法用于循环集合项并将其传递到回调函数中。

$collection->each(function ($item, $key) {

//

});

//

如果你想中断对集合项的循环，那么就在你的回调函数中返回 false。

$collection->each(function ($item, $key) {

if (/\* some condition \*/) {

return false;

}

});

(22)、eachSpread()

eachSpread 方法用于循环集合项，将每个嵌套集合项的值传递给回调函数。

$collection = collect([['John Doe', 35], ['Jane Doe', 33]]);

$collection->eachSpread(function ($name, $age) {

//

});

//

你可以通过在回调函数里返回 false 来中断循环。

$collection->eachSpread(function ($name, $age) {

return false;

});

(23)、every()

every 方法可用于验证集合中的每一个元素是否通过指定的条件测试。

collect([1, 2, 3, 4])->every(function ($value, $key) {

return $value > 2;

}); // false

//

如果集合为空， every 将返回 true。

$collection = collect([]);

$collection->every(function($value, $key) {

return $value > 2;

}); // true

(24)、except()

except 方法返回集合中除了指定键之外的所有集合项。

$collection = collect(['product\_id' => 1, 'price' => 100, 'discount' => false]);

$filtered = $collection->except(['price', 'discount']);

$filtered->all(); // ['product\_id' => 1]

与 except 对应的是 only ( https://learnku.com/docs/laravel/6.x/collections/5161#method-only ) 方法。

(25)、filter()

filter 方法使用给定的回调函数过滤集合，只保留那些通过指定条件测试的集合项。

$collection = collect([1, 2, 3, 4]);

$filtered = $collection->filter(function ($value, $key) {

return $value > 2;

});

$filtered->all(); // [3, 4]

//

如果没有提供回调函数，集合中所有返回 false 的元素都会被移除。

$collection = collect([1, 2, 3, null, false, '', 0, []]);

$collection->filter()->all(); // [1, 2, 3]

与 filter 对应的是 reject ( https://learnku.com/docs/laravel/6.x/collections/5161#method-reject ) 方法。

(26)、first()

first 方法返回集合中通过指定条件测试的第一个元素。

collect([1, 2, 3, 4])->first(function ($value, $key) {

return $value > 2;

}); // 3

//

你也可以不传入参数调用 first 方法来获取集合中的第一个元素。如果集合为空，则会返回 null。

collect([1, 2, 3, 4])->first(); // 1

(27)、firstWhere()

firstWhere 方法返回集合中含有指定键 / 值对的第一个元素。

$collection = collect([

['name' => 'Regena', 'age' => null],

['name' => 'Linda', 'age' => 14],

['name' => 'Diego', 'age' => 23],

['name' => 'Linda', 'age' => 84],

]);

$collection->firstWhere('name', 'Linda'); // ['name' => 'Linda', 'age' => 14]

//

你也可以使用运算符来调用 firstWhere 方法。

$collection->firstWhere('age', '>=', 18); // ['name' => 'Diego', 'age' => 23]

//

和 where ( https://learnku.com/docs/laravel/6.x/collections/5161#method-where ) 方法一样，你可以将一个参数传递给 firstWhere 方法。在这种情况下， firstWhere 方法将返回指定键的值为「真」的第一个集合项。

$collection->firstWhere('age'); // ['name' => 'Linda', 'age' => 14]

(28)、flatMap()

flatMap 方法遍历集合并将其中的每个值传递到给定的回调函数。可以通过回调函数修改集合项并返回它们，从而形成一个被修改过的新集合。然后，集合转化的数组是同级的。

$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'];

(29)、flatten()

flatten 方法将多维集合转为一维集合。

$collection = collect(['name' => 'taylor', 'languages' => ['php', 'javascript']]);

$flattened = $collection->flatten();

$flattened->all(); // ['taylor', 'php', 'javascript'];

//

你可以选择性地传入「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'],

]

\*/

在这个例子里，调用 flatten 时不传入深度参数的话也会将嵌套数组转成一维的，然后返回 ['iPhone 6S', 'Apple', 'Galaxy S7', 'Samsung'] 。传入深度参数能让你限制设置返回数组的层数。

(30)、flip()

flip 方法将集合的键和对应的值进行互换。

$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$flipped = $collection->flip();

$flipped->all(); // ['taylor' => 'name', 'laravel' => 'framework']

(31)、forget()

forget 方法将通过指定的键来移除集合中对应的内容。

$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$collection->forget('name');

$collection->all(); // ['framework' => 'laravel']

注意：与大多数集合的方法不同的是， forget 不会返回修改后的新集合；它会直接修改原集合。

(32)、forPage()

forPage 方法返回一个含有指定页码数集合项的新集合。这个方法接受页码数作为其第一个参数，每页显示的项数作为其第二个参数。

$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9]);

$chunk = $collection->forPage(2, 3);

$chunk->all(); // [4, 5, 6]

(33)、get()

get 方法返回指定键的集合项，如果该键在集合中不存在，则返回 null。

$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('name'); // taylor

//

你可以任选一个默认值作为第二个参数传递。

$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('foo', 'default-value'); // default-value

//

你甚至可以将一个回调函数作为默认值传递。如果指定的键不存在，就会返回回调函数的结果。

$collection->get('email', function () {

return 'default-value';

}); // default-value

(34)、groupBy()

groupBy 方法根据指定键对集合项进行分组。

$collection = collect([

['account\_id' => 'account-x10', 'product' => 'Chair'],

['account\_id' => 'account-x10', 'product' => 'Bookcase'],

['account\_id' => 'account-x11', 'product' => 'Desk'],

]);

$grouped = $collection->groupBy('account\_id');

$grouped->toArray();

/\*

[

'account-x10' => [

['account\_id' => 'account-x10', 'product' => 'Chair'],

['account\_id' => 'account-x10', 'product' => 'Bookcase'],

],

'account-x11' => [

['account\_id' => 'account-x11', 'product' => 'Desk'],

],

]

\*/

你可以传递一个回调函数用来代替一个字符串的 键。这个回调函数应该返回你希望用来分组的键的值。

$grouped = $collection->groupBy(function ($item, $key) {

return substr($item['account\_id'], -3);

});

$grouped->toArray();

/\*

[

'x10' => [

['account\_id' => 'account-x10', 'product' => 'Chair'],

['account\_id' => 'account-x10', 'product' => 'Bookcase'],

],

'x11' => [

['account\_id' => 'account-x11', 'product' => 'Desk'],

],

]

\*/

可以传递一个数组用于多重分组标准。每一个数组元素将对应多维数组内的相应级别。

$data = new Collection([

10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role\_1', 'Role\_3']],

20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role\_1', 'Role\_2']],

30 => ['user' => 3, 'skill' => 2, 'roles' => ['Role\_1']],

40 => ['user' => 4, 'skill' => 2, 'roles' => ['Role\_2']],

]);

$result = $data->groupBy([

'skill',

function ($item) {

return $item['roles'];

},

], $preserveKeys = true);

/\*

[

1 => [

'Role\_1' => [

10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role\_1', 'Role\_3']],

20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role\_1', 'Role\_2']],

],

'Role\_2' => [

20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role\_1', 'Role\_2']],

],

'Role\_3' => [

10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role\_1', 'Role\_3']],

],

],

2 => [

'Role\_1' => [

30 => ['user' => 3, 'skill' => 2, 'roles' => ['Role\_1']],

],

'Role\_2' => [

40 => ['user' => 4, 'skill' => 2, 'roles' => ['Role\_2']],

],

],

];

\*/

(35)、has()

has 方法判断集合中是否存在指定键。

$collection = collect(['account\_id' => 1, 'product' => 'Desk', 'amount' => 5]);

$collection->has('product'); // true

$collection->has(['product', 'amount']); // true

$collection->has(['amount', 'price']); // false

(36)、implode

implode 方法用于合并集合项。其参数取决于集合项的类型。如果集合包含数组或对象，你应该传递你希望合并的属性的键，以及你希望放在值之间用来「拼接」的字符串。

$collection = collect([

['account\_id' => 1, 'product' => 'Desk'],

['account\_id' => 2, 'product' => 'Chair'],

]);

$collection->implode('product', ', '); // Desk, Chair

如果集合中包含简单的字符串或数值，只需要传入「拼接」用的字符串作为该方法的唯一参数即可。

collect([1, 2, 3, 4, 5])->implode('-'); // '1-2-3-4-5'

(37)、intersect()

intersect 方法从原集合中移除在指定数组 或集合中不存在的任何值。生成的集合将会保留原集合的键。

$collection = collect(['Desk', 'Sofa', 'Chair']);

$intersect = $collection->intersect(['Desk', 'Chair', 'Bookcase']);

$intersect->all(); // [0 => 'Desk', 2 => 'Chair']

(38)、intersectByKeys()

intersectByKeys 方法从原集合中移除在指定 数组 或集合中不存在的任何键。

$collection = collect([

'serial' => 'UX301', 'type' => 'screen', 'year' => 2009

]);

$intersect = $collection->intersectByKeys([

'reference' => 'UX404', 'type' => 'tab', 'year' => 2011

]);

$intersect->all(); // ['type' => 'screen', 'year' => 2009]

(39)、isEmpty()

如果集合为空， isEmpty 方法返回 true，否则返回 false。

collect([])->isEmpty(); // true

(40)、isNotEmpty

如果集合不为空， isNotEmpty 方法返回 true ，否则返回 false。

collect([])->isNotEmpty(); // false

(41)、join()

join 方法将集合中的值用字符串连接。

collect(['a', 'b', 'c'])->join(', '); // 'a, b, c'

collect(['a', 'b', 'c'])->join(', ', ', and '); // 'a, b, and c'

collect(['a', 'b'])->join(', ', ' and '); // 'a and b'

collect(['a'])->join(', ', ' and '); // 'a'

collect([])->join(', ', ' and '); // ''

(42)、keyBy()

keyBy 方法以指定的键作为集合的键。如果多个集合项具有相同的键，则只有最后一个集合项会显示在新集合中。

$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'],

]

\*/

你还可以在这个方法传递一个回调函数。该回调函数返回的值会作为该集合的键。

$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'],

]

\*/

(43)、keys()

keys 方法返回集合中的所有键。

$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']

(44)、last()

last 方法返回集合中通过指定条件测试的最后一个元素。

collect([1, 2, 3, 4])->last(function ($value, $key) {

return $value < 3;

}); // 2

不传入参数的情况下， last 方法会获取集合中的最后一个元素。如果集合为空，那么方法将返回 null。

collect([1, 2, 3, 4])->last(); // 4

(45)、macro()

静态 macro 方法允许你在运行时将方法添加至 Collection 类。关于更多信息，请参阅 扩展集合( https://learnku.com/docs/laravel/6.0/collections/5161#extending-collections ) 的文档。

(46)、make()

静态 make 方法可以创建一个新的集合实例。请参阅 创建集合( https://learnku.com/docs/laravel/6.0/collections/5161#creating-collections ) 部分。

(47)、map()

map 方法遍历集合并将每一个值传入给定的回调函数。该回调函数可以任意修改集合项并返回，从而生成被修改过集合项的新集合。

$collection = collect([1, 2, 3, 4, 5]);

$multiplied = $collection->map(function ($item, $key) {

return $item \* 2;

});

$multiplied->all(); // [2, 4, 6, 8, 10]

注意：与其它大多数集合方法一样，map 会返回一个新的集合实例；它不会修改原集合。如果你想修改原集合，请使用 transform ( https://learnku.com/docs/laravel/6.0/collections/5161#method-transform ) 方法。

(48)、mapInto()

mapInto() 方法可以迭代集合，通过将值传递给构造函数来创建给定类的新实例。

class Currency

{

/\*\*

\* Create a new currency instance.

\*

\* @param string $code

\* @return void

\*/

function \_\_construct(string $code)

{

$this->code = $code;

}

}

$collection = collect(['USD', 'EUR', 'GBP']);

$currencies = $collection->mapInto(Currency::class);

$currencies->all(); // [Currency('USD'), Currency('EUR'), Currency('GBP')]

(49)、mapSpread()

mapSpread 方法可以遍历集合项，将每个嵌套项值给指定的回调函数。该回调函数可以自由修改该集合项并返回，从而生成被修改过集合项的新集合。

$collection = collect([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]);

$chunks = $collection->chunk(2);

$sequence = $chunks->mapSpread(function ($even, $odd) {

return $even + $odd;

});

$sequence->all(); // [1, 5, 9, 13, 17]

(50)、mapToGroups()

mapToGroups 方法通过给定的回调函数对集合项进行分组。该回调函数应该返回一个包含单个键 / 值对的关联数组，从而生成一个分组值的新集合。

$collection = collect([

[

'name' => 'John Doe',

'department' => 'Sales',

],

[

'name' => 'Jane Doe',

'department' => 'Sales',

],

[

'name' => 'Johnny Doe',

'department' => 'Marketing',

]

]);

$grouped = $collection->mapToGroups(function ($item, $key) {

return [$item['department'] => $item['name']];

});

$grouped->toArray();

/\*

[

'Sales' => ['John Doe', 'Jane Doe'],

'Marketing' => ['Johnny Doe'],

]

\*/

$grouped->get('Sales')->all(); // ['John Doe', 'Jane Doe']

(51)、mapWithKeys()

mapWithKeys 方法遍历集合并将每个值传入给定的回调函数。该回调函数将返回一个包含单个键 / 值对的关联数组。

$collection = collect([

[

'name' => 'John',

'department' => 'Sales',

'email' => 'john@example.com'

],

[

'name' => 'Jane',

'department' => 'Marketing',

'email' => 'jane@example.com'

]

]);

$keyed = $collection->mapWithKeys(function ($item) {

return [$item['email'] => $item['name']];

});

$keyed->all();

/\*

[

'john@example.com' => 'John',

'jane@example.com' => 'Jane',

]

\*/

(52)、max()

max 方法返回指定键的最大值。

$max = collect([['foo' => 10], ['foo' => 20]])->max('foo'); // 20

$max = collect([1, 2, 3, 4, 5])->max(); // 5

(53)、median()

median 方法返回指定键的 中间值( https://en.wikipedia.org/wiki/Median )。

$median = collect([['foo' => 10], ['foo' => 10], ['foo' => 20], ['foo' => 40]])->median('foo'); // 15

$median = collect([1, 1, 2, 4])->median(); // 1.5

(54)、merge()

merge 方法将合并指定的数组或集合到原集合。如果给定的集合项的字符串键与原集合中的字符串键相匹配，则指定集合项的值将覆盖原集合的值。

$collection = collect(['product\_id' => 1, 'price' => 100]);

$merged = $collection->merge(['price' => 200, 'discount' => false]);

$merged->all(); // ['product\_id' => 1, 'price' => 200, 'discount' => false]

//

如果给定的集合项为数字，则这些值将会追加在集合的最后。

$collection = collect(['Desk', 'Chair']);

$merged = $collection->merge(['Bookcase', 'Door']);

$merged->all(); // ['Desk', 'Chair', 'Bookcase', 'Door']

(55)、mergeRecursive()

mergeRecursive 方法以递归的形式合并给定的数组或集合到原集合中，如果给定集合项的字符串键与原集合的字符串键一致，则会将给定的集合项的值以递归的形式合并到原集合的相同键中。

$collection = collect(['product\_id' => 1, 'price' => 100]);

$merged = $collection->mergeRecursive(['product\_id' => 2, 'price' => 200, 'discount' => false]);

$merged->all(); // ['product\_id' => [1, 2], 'price' => [100, 200], 'discount' => false]

(56)、min()

min 方法返回指定键的最小值。

$min = collect([['foo' => 10], ['foo' => 20]])->min('foo'); // 10

$min = collect([1, 2, 3, 4, 5])->min(); // 1

(57)、mode()

mode 方法返回指定键的 众数( https://en.wikipedia.org/wiki/Mode\_(statistics) )。

$mode = collect([['foo' => 10], ['foo' => 10], ['foo' => 20], ['foo' => 40]])->mode('foo'); // [10]

$mode = collect([1, 1, 2, 4])->mode(); // [1]

(58)、nth()

nth 方法创建由每隔 n 个元素组成的一个新集合。

$collection = collect(['a', 'b', 'c', 'd', 'e', 'f']);

$collection->nth(4); // ['a', 'e']

//

你也可以设置指定的偏移位置作为第二个参数。

$collection->nth(4, 1); // ['b', 'f']

(59)、only()

only 方法返回集合中所有指定键的集合项。

$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 对应的是 except ( https://learnku.com/docs/laravel/6.0/collections/5161#method-except ) 方法。

(60)、pad()

pad 方法将使用给定的值填充数组，直到数组达到指定的大小。该方法的行为与 array\_pad ( https://secure.php.net/manual/en/function.array-pad.php ) PHP 函数功能类似。

要填充到左侧，你应该使用负值。如果给定大小的绝对值小于或等于数组的长度，则不会发生填充。

$collection = collect(['A', 'B', 'C']);

$filtered = $collection->pad(5, 0);

$filtered->all(); // ['A', 'B', 'C', 0, 0]

$filtered = $collection->pad(-5, 0);

$filtered->all(); // [0, 0, 'A', 'B', 'C']

(61)、partition()

partition 方法可以和 PHP 函数中的 list 结合使用，用来分开通过指定条件的元素以及那些不通过指定条件的元素。

$collection = collect([1, 2, 3, 4, 5, 6]);

list($underThree, $equalOrAboveThree) = $collection->partition(function ($i) {

return $i < 3;

});

$underThree->all(); // [1, 2]

$equalOrAboveThree->all(); // [3, 4, 5, 6]

(62)、pipe()

pipe 方法传入一个回调函数并返回结果。

$collection = collect([1, 2, 3]);

$piped = $collection->pipe(function ($collection) {

return $collection->sum();

}); // 6

(63)、pluck()

pluck 方法可以获取集合中指定键对应的所有值。

$collection = collect([

['product\_id' => 'prod-100', 'name' => 'Desk'],

['product\_id' => 'prod-200', 'name' => 'Chair'],

]);

$plucked = $collection->pluck('name');

$plucked->all(); // ['Desk', 'Chair']

//

你也可以通过传入第二个参数来指定生成集合的键。

$plucked = $collection->pluck('name', 'product\_id');

$plucked->all(); // ['prod-100' => 'Desk', 'prod-200' => 'Chair']

//

如果存在重复的键，则最后一个匹配元素将被插入到弹出的集合中。

$collection = collect([

['brand' => 'Tesla', 'color' => 'red'],

['brand' => 'Pagani', 'color' => 'white'],

['brand' => 'Tesla', 'color' => 'black'],

['brand' => 'Pagani', 'color' => 'orange'],

]);

$plucked = $collection->pluck('color', 'brand');

$plucked->all(); // ['Tesla' => 'black', 'Pagani' => 'orange']

(64)、pop()

pop 从原集合返回并移除最后一个集合项。

$collection = collect([1, 2, 3, 4, 5]);

$collection->pop(); // 5

$collection->all(); // [1, 2, 3, 4]

(65)、prepend()

prepend 方法将指定的值添加到集合的开头。

$collection = collect([1, 2, 3, 4, 5]);

$collection->prepend(0);

$collection->all(); // [0, 1, 2, 3, 4, 5]

//

你也可以传递第二个参数来设置新增加集合项的键。

$collection = collect(['one' => 1, 'two' => 2]);

$collection->prepend(0, 'zero');

$collection->all(); // ['zero' => 0, 'one' => 1, 'two' => 2]

(66)、pull()

pull 方法把指定键对应的值从集合中移除并返回。

$collection = collect(['product\_id' => 'prod-100', 'name' => 'Desk']);

$collection->pull('name'); // 'Desk'

$collection->all(); // ['product\_id' => 'prod-100']

(67)、push()

push 方法把指定的值追加到集合项的末尾。

$collection = collect([1, 2, 3, 4]);

$collection->push(5);

$collection->all(); // [1, 2, 3, 4, 5]

(68)、put()

put 方法在集合内设置给定的键值对。

$collection = collect(['product\_id' => 1, 'name' => 'Desk']);

$collection->put('price', 100);

$collection->all(); // ['product\_id' => 1, 'name' => 'Desk', 'price' => 100]

(69)、random()

random 方法从集合中返回一个随机项。

$collection = collect([1, 2, 3, 4, 5]);

$collection->random(); // 4 - (retrieved randomly)

你可以选择传入一个整数到 random 来指定要获取的随即项的数量。只要你显示传递你希望接收的数量时，则会返回项目的集合。

$random = $collection->random(3);

$random->all(); // [2, 4, 5] - (retrieved randomly)

如果集合的项小于指定的数量，则该方法将抛出 InvalidArgumentException。

(70)、reduce()

reduce 方法将每次迭代的结果传递给下一次迭代直到集合减少为单个值。

$collection = collect([1, 2, 3]);

$total = $collection->reduce(function ($carry, $item) {

return $carry + $item;

}); // 6

第一次迭代时 $carry 的数值为 null； however，你也可以通过传入第二个参数到 reduce 来指定它的初始值。

$collection->reduce(function ($carry, $item) {

return $carry + $item;

}, 4); // 10

(71)、reject()

reject 方法使用指定的回调函数过滤集合。如果回调函数返回 true 就会把对应的集合项从集合中移除。

$collection = collect([1, 2, 3, 4]);

$filtered = $collection->reject(function ($value, $key) {

return $value > 2;

});

$filtered->all(); // [1, 2]

与 reject 方法对应的是 filter 方法。

(72)、replace()

replace 方法的行为类似 merge，但是，不仅可以覆盖匹配到的相同字符串键的集合项，而且也可以覆盖数字键的集合项。

$collection = collect(['Taylor', 'Abigail', 'James']);

$replaced = $collection->replace([1 => 'Victoria', 3 => 'Finn']);

$replaced->all(); // ['Taylor', 'Victoria', 'James', 'Finn']

(73)、replaceRecursive()

replaceRecursive 类似 replace, 但是会以递归的形式将数组替换到具有相同键的集合项中。

$collection = collect(['Taylor', 'Abigail', ['James', 'Victoria', 'Finn']]);

$replaced = $collection->replaceRecursive(['Charlie', 2 => [1 => 'King']]);

$replaced->all(); // ['Charlie', 'Abigail', ['James', 'King', 'Finn']]

(74)、reverse()

reverse 方法用来倒转集合项的顺序，并保留原始的键。

$collection = collect(['a', 'b', 'c', 'd', 'e']);

$reversed = $collection->reverse();

$reversed->all();

/\*

[

4 => 'e',

3 => 'd',

2 => 'c',

1 => 'b',

0 => 'a',

]

\*/

(75)、search()

search 方法在集合中搜索给定的值并返回它的键。如果没有找到，则返回 false。

$collection = collect([2, 4, 6, 8]);

$collection->search(4); // 1

//

使用 「宽松」的方式进行搜索，这意味着具有整数值的字符串会被认为等于相同值的整数。使用 「严格」的方式进行搜索，就传入 true 作为该方法的第二个参数。

$collection->search('4', true); // false

//

或者，你可以通过传递回调函数来搜索通过条件测试的第一个集合项。

$collection->search(function ($item, $key) {

return $item > 5;

}); // 2

(76)、shift()

shift 方法移除并返回集合的第一个集合项。

$collection = collect([1, 2, 3, 4, 5]);

$collection->shift(); // 1

$collection->all(); // [2, 3, 4, 5]

(77)、shuffle()

shuffle 方法随机打乱集合项。

$collection = collect([1, 2, 3, 4, 5]);

$shuffled = $collection->shuffle();

$shuffled->all(); // [3, 2, 5, 1, 4] - (generated randomly)

(78)、skip()

skip 方法返回除了给定的集合项数目的新集合。

$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

$collection = $collection->skip(4);

$collection->all(); // [5, 6, 7, 8, 9, 10]

(79)、slice()

slice 方法返回集合中给定索引开始后面的部分。

$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

$slice = $collection->slice(4);

$slice->all(); // [5, 6, 7, 8, 9, 10]

//

如果你想限制返回内容的大小，可以将你期望的大小作为第二个参数传递到该方法。

$slice = $collection->slice(4, 2);

$slice->all(); // [5, 6]

默认情况下，返回的内容将会保留原始键。如果你不希望保留原始键，你可以使用 values 方法来重新建立索引。

(80)、some()

contains 方法的别名。

(81)、sort()

sort 方法对集合进行排序。排序后的集合会保留原数组的键，所以在这个例子我们将使用 values 方法去把键重置为连续编号的索引。

$collection = collect([5, 3, 1, 2, 4]);

$sorted = $collection->sort();

$sorted->values()->all(); // [1, 2, 3, 4, 5]

如果你有更高级的排序需求，可以通过自己的算法将回调函数传递到 sort 。请参阅 PHP 文档的 uasort ，这是集合的 sort 方法在底层所调用的。

Tip：如果你需要对嵌套数组或对象进行排序，请参照 sortBy( https://learnku.com/docs/laravel/6.0/collections/5161#method-sortby ) 和 sortByDesc( https://learnku.com/docs/laravel/6.0/collections/5161#method-sortbydesc ) 方法。

(82)、sortBy()

sortBy 方法将根据指定键对集合进行排序。排序后的集合会保留原始数组的键，所以在这个例子中我们使用 values 方法将键重置为连续编号的索引。

$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],

]

\*/

//

你也可以传递你自己的回调函数用于决定如何对集合的值进行排序。

$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']],

]

\*/

(83)、sortByDesc()

该方法与 sortBy 方法一样，但是会以相反的顺序来对集合进行排序。

(84)、sortKeys()

sortKeys 方法通过底层关联数组的键来对集合进行排序。

$collection = collect([

'id' => 22345,

'first' => 'John',

'last' => 'Doe',

]);

$sorted = $collection->sortKeys();

$sorted->all();

/\*

[

'first' => 'John',

'id' => 22345,

'last' => 'Doe',

]

\*/

(85)、sortKeysDesc()

该方法与 sortKeys 方法一样，但是会以相反的顺序来对集合进行排序。

(86)、splice()

splice 方法移除并返回指定索引开始的集合项片段。

$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2);

$chunk->all(); // [3, 4, 5]

$collection->all(); // [1, 2]

//

你可以传递第二个参数用以限制被删除内容的大小。

$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2, 1);

$chunk->all(); // [3]

$collection->all(); // [1, 2, 4, 5]

//

此外，你可以传入含有新参数项的第三个参数来代替集合中删除的集合项。

$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]

(87)、split()

split 方法将集合按照给定的值拆分。

$collection = collect([1, 2, 3, 4, 5]);

$groups = $collection->split(3);

$groups->toArray(); // [[1, 2], [3, 4], [5]]

(88)、sum()

sum 方法返回集合内所有项的和。

collect([1, 2, 3, 4, 5])->sum(); // 15

//

如果集合包含嵌套数组或对象，则应该传入一个键来指定要进行求和的值。

$collection = collect([

['name' => 'JavaScript: The Good Parts', 'pages' => 176],

['name' => 'JavaScript: The Definitive Guide', 'pages' => 1096],

]);

$collection->sum('pages'); // 1272

//

另外，你可以传入自己的回调函数来决定要用集合中的哪些值进行求和。

$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

(89)、take()

take 方法返回给定数量项的新集合。

$collection = collect([0, 1, 2, 3, 4, 5]);

$chunk = $collection->take(3);

$chunk->all(); // [0, 1, 2]

//

你也可以传递负整数从集合末尾获取指定数量的项。

$collection = collect([0, 1, 2, 3, 4, 5]);

$chunk = $collection->take(-2);

$chunk->all(); // [4, 5]

(90)、tap()

tap 方法将给定的回调函数传入该集合，允许你在一个特定点「tap」集合，并在不影响集合本身的情况下对集合项执行某些操作。

collect([2, 4, 3, 1, 5])

->sort()

->tap(function ($collection) {

Log::debug('Values after sorting', $collection->values()->toArray());

})

->shift(); // 1

(91)、times()

静态 times 方法通过调用给定次数的回调函数来创建新集合。

$collection = Collection::times(10, function ($number) {

return $number \* 9;

});

$collection->all(); // [9, 18, 27, 36, 45, 54, 63, 72, 81, 90]

//

使用这个方法可以与工厂结合使用创建出 Eloquent ( https://learnku.com/docs/laravel/6.0/eloquent ) 模型。

$categories = Collection::times(3, function ($number) {

return factory(Category::class)->create(['name' => "Category No. $number"]);

});

$categories->all();

/\*

[

['id' => 1, 'name' => 'Category #1'],

['id' => 2, 'name' => 'Category #2'],

['id' => 3, 'name' => 'Category #3'],

]

\*/

(92)、toArray()

toArray 方法将集合转换成 PHP 数组 。如果集合的值是 Eloquent ( https://learnku.com/docs/laravel/6.0/eloquent ) 模型，那也会被转换成数组。

$collection = collect(['name' => 'Desk', 'price' => 200]);

$collection->toArray();

/\*

[

['name' => 'Desk', 'price' => 200],

]

\*/

注意：toArray 也会将所有集合的嵌套对象转换为数组。如果你想获取原数组，可以使用 all ( https://learnku.com/docs/laravel/6.0/collections/3916#method-all ) 方法。

(93)、toJson()

toJson 方法将集合转换成 JSON 字符串。

$collection = collect(['name' => 'Desk', 'price' => 200]);

$collection->toJson(); // '{"name":"Desk", "price":200}'

(94)、transform()

注意：与大多数集合方法不同， transform 会修改集合本身。如果你想创建新集合，可以使用 map ( https://learnku.com/docs/laravel/6.0/collections/3916#method-map ) 方法。

$collection = collect([1, 2, 3, 4, 5]);

$collection->transform(function ($item, $key) {

return $item \* 2;

});

$collection->all(); // [2, 4, 6, 8, 10]

(95)、union()

union 方法将给定的数组添加到集合。如果给定的数组含有与原集合一样的键，则原集合的值不会被改变。

$collection = collect([1 => ['a'], 2 => ['b']]);

$union = $collection->union([3 => ['c'], 1 => ['b']]);

$union->all(); // [1 => ['a'], 2 => ['b'], 3 => ['c']]

(96)、unique()

unique 方法返回集合中所有唯一项。返回的集合保留着原数组的键，所以在这个例子中，我们使用 values 方法把键重置为连续编号的索引。

$collection = collect([1, 1, 2, 2, 3, 4, 2]);

$unique = $collection->unique();

$unique->values()->all(); // [1, 2, 3, 4]

//

在处理嵌套数组或对象时，你可以指定用于确定唯一性的键。

$collection = collect([

['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],

['name' => 'iPhone 5', 'brand' => 'Apple', 'type' => 'phone'],

['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],

['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],

['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],

]);

$unique = $collection->unique('brand');

$unique->values()->all();

/\*

[

['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],

['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],

]

\*/

//

你也可以通过传递自己的回调函数来确定项的唯一性。

$unique = $collection->unique(function ($item) {

return $item['brand'].$item['type'];

});

$unique->values()->all();

/\*

[

['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],

['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],

['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],

['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],

]

\*/

unique 方法在检查项目值时使用「宽松」模式比较，意味着具有整数值的字符串将被视为等于相同值的整数。你可以使用 uniqueStrict 方法做「严格」模式比较。

(97)、uniqueStrict()

这个方法与 unique 方法一样，然而，所有的值是用 「严格」模式来比较的。

(98)、unless()

unless 方法当传入的第一个参数不为 true 的时候，将执行给定的回调函数。

$collection = collect([1, 2, 3]);

$collection->unless(true, function ($collection) {

return $collection->push(4);

});

$collection->unless(false, function ($collection) {

return $collection->push(5);

});

$collection->all(); // [1, 2, 3, 5]

when 方法与 unless 方法对应。

(99)、unlessEmpty()

whenNotEmpty 方法的别名。

(100)、unlessNotEmpty()

whenEmpty 方法的别名。

(101)、unwrap()

静态 unwrap 方法返回集合内部的可用值。

Collection::unwrap(collect('John Doe')); // ['John Doe']

Collection::unwrap(['John Doe']); // ['John Doe']

Collection::unwrap('John Doe'); // 'John Doe'

(102)、values()

values 方法返回键被重置为连续编号的新集合。

$collection = collect([

10 => ['product' => 'Desk', 'price' => 200],

11 => ['product' => 'Desk', 'price' => 200]

]);

$values = $collection->values();

$values->all();

/\*

[

0 => ['product' => 'Desk', 'price' => 200],

1 => ['product' => 'Desk', 'price' => 200],

]

\*/

(103)、when()

when 方法当传入的第一个参数为 true 时，将执行给定的回调函数。

$collection = collect([1, 2, 3]);

$collection->when(true, function ($collection) {

return $collection->push(4);

});

$collection->when(false, function ($collection) {

return $collection->push(5);

});

$collection->all(); // [1, 2, 3, 4]

与 unless 对应的是 when 方法。

(104)、whenEmpty()

whenEmpty 方法当集合为空时，将执行给定的回调函数。

$collection = collect(['michael', 'tom']);

$collection->whenEmpty(function ($collection) {

return $collection->push('adam');

});

$collection->all(); // ['michael', 'tom']

//

$collection = collect();

$collection->whenEmpty(function ($collection) {

return $collection->push('adam');

});

$collection->all(); // ['adam']

//

$collection = collect(['michael', 'tom']);

$collection->whenEmpty(function($collection) {

return $collection->push('adam');

}, function($collection) {

return $collection->push('taylor');

});

$collection->all(); // ['michael', 'tom', 'taylor']

与 whenEmpty 对应的是 whenNotEmpty 方法。

(105)、whenNotEmpty()

whenNotEmpty 方法当集合不为空时，将执行给定的回调函数。

$collection = collect(['michael', 'tom']);

$collection->whenNotEmpty(function ($collection) {

return $collection->push('adam');

});

$collection->all(); // ['michael', 'tom', 'adam']

//

$collection = collect();

$collection->whenNotEmpty(function ($collection) {

return $collection->push('adam');

});

$collection->all(); // []

//

$collection = collect();

$collection->whenNotEmpty(function($collection) {

return $collection->push('adam');

}, function($collection) {

return $collection->push('taylor');

});

$collection->all(); // ['taylor']

与 whenNotEmpty 对应的是 whenEmpty 方法。

(106)、where()

where 方法通过给定的键 / 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],

]

\*/

where 方法在检查集合项值时使用「宽松」模式比较，这意味着具有整数值的字符串会被认为等于相同值的整数。你可以使用 whereStrict 方法进行「严格」模式比较。而且，你还可以将一个比较运算符作为第二个参数传递。

$collection = collect([

['name' => 'Jim', 'deleted\_at' => '2019-01-01 00:00:00'],

['name' => 'Sally', 'deleted\_at' => '2019-01-02 00:00:00'],

['name' => 'Sue', 'deleted\_at' => null],

]);

$filtered = $collection->where('deleted\_at', '!=', null);

$filtered->all();

/\*

[

['name' => 'Jim', 'deleted\_at' => '2019-01-01 00:00:00'],

['name' => 'Sally', 'deleted\_at' => '2019-01-02 00:00:00'],

]

\*/

(107)、whereStrict()

这个方法与 where 方法类似；不同的是会用「严格」的模式比较。

(108)、whereBetween()

whereBetween 方法会用给定的范围对集合进行过滤。

$collection = collect([

['product' => 'Desk', 'price' => 200],

['product' => 'Chair', 'price' => 80],

['product' => 'Bookcase', 'price' => 150],

['product' => 'Pencil', 'price' => 30],

['product' => 'Door', 'price' => 100],

]);

$filtered = $collection->whereBetween('price', [100, 200]);

$filtered->all();

/\*

[

['product' => 'Desk', 'price' => 200],

['product' => 'Bookcase', 'price' => 150],

['product' => 'Door', 'price' => 100],

]

\*/

(109)、whereIn()

whereIn 方法会根据包含给定数组的键 / 值对来过滤集合。

$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 方法使用「宽松」的比较来检查集合项的值，这意味着具有整数值的字符串会被视为等于相同值的整数。你可以使用 whereInStrict 方法进行「严格」模式比较。

(110)、whereInStrict()

这个方法与 whereIn 方法类似；不同的是会使用「严格」模式进行比较。

(111)、whereInstanceOf()

whereInstanceOf 方法根据指定的类来过滤集合。

$collection = collect([

new User,

new User,

new Post,

]);

return $collection->whereInstanceOf(User::class);

(112)、whereNotBetween()

whereNotBetween 方法在指定的范围内过滤集合。

$collection = collect([

['product' => 'Desk', 'price' => 200],

['product' => 'Chair', 'price' => 80],

['product' => 'Bookcase', 'price' => 150],

['product' => 'Pencil', 'price' => 30],

['product' => 'Door', 'price' => 100],

]);

$filtered = $collection->whereNotBetween('price', [100, 200]);

$filtered->all();

/\*

[

['product' => 'Chair', 'price' => 80],

['product' => 'Pencil', 'price' => 30],

]

\*/

(113)、whereNotIn()

whereNotIn 方法根据未包含在指定数组的键 / 值对来对集合进行过滤。

$collection = collect([

['product' => 'Desk', 'price' => 200],

['product' => 'Chair', 'price' => 100],

['product' => 'Bookcase', 'price' => 150],

['product' => 'Door', 'price' => 100],

]);

$filtered = $collection->whereNotIn('price', [150, 200]);

$filtered->all();

/\*

[

['product' => 'Chair', 'price' => 100],

['product' => 'Door', 'price' => 100],

]

\*/

whereNotIn 方法使用「宽松」模式比较来检查集合项的值，这意味着具有整数值的字符串会被视为等同于相同值的整数。你可以使用 whereNotInStrict 方法做「严格」模式比较。

(114)、whereNotInStrict()

这个方法与 whereNotIn 方法类似；不同的是会使用「严格」模式比较。

(115)、wrap()

静态 wrap 方法在适当的情况下将指定的值放在集合中。

$collection = Collection::wrap('John Doe');

$collection->all(); // ['John Doe']

$collection = Collection::wrap(['John Doe']);

$collection->all(); // ['John Doe']

$collection = Collection::wrap(collect('John Doe'));

$collection->all(); // ['John Doe']

(116)、zip()

zip 方法将指定数组的值和相应索引的原集合的值合并在一起。

$collection = collect(['Chair', 'Desk']);

$zipped = $collection->zip([100, 200]);

$zipped->all(); // [['Chair', 100], ['Desk', 200]]

--

3、高阶信息 (高阶消息传递)

集合也提供对「高阶消息传递」的支持，即集合常见操作的快捷方式。支持高阶消息传递的集合方法有： average, avg, contains, each, every, filter, first, flatMap, groupBy, keyBy, map, max, min, partition, reject, some, sortBy, sortByDesc, sum, 和 unique。

每个高阶消息传递都能作为集合实例的动态的属性来访问。例如，使用 each 高阶消息传递在集合中的每个对象上调用一个方法。

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

$users->each->markAsVip();

同样，我们可以使用 sum 高阶消息传递来收集 users 集合中的「投票」总数。

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

return $users->sum->votes;

--

4、惰性集合 (懒集合)

(1)、简介

注意：在进一步学习 Laravel 的懒集合之前，我们先花点时间熟悉一下 PHP generators ( https://www.php.net/manual/en/language.generators.overview.php )。

为了补充已经很强大的 Collection 类， LazyCollection 类利用了 PHP 的 generators ( https://www.php.net/manual/en/language.generators.overview.php ) 来允许你在保持低内存使用率的同时使用非常大的数据集。

例如，假设你的应用需要处理一个千兆字节的日志文件，同时利用 Laravel 的集合方法来解析这个日志文件。与其一次将整个文件读入内存，不如使用懒集合在给定时间仅将文件的一小部分保留在内存中。

use App\LogEntry;

use Illuminate\Support\LazyCollection;

LazyCollection::make(function () {

$handle = fopen('log.txt', 'r');

while (($line = fgets($handle)) !== false) {

yield $line;

}

})->chunk(4)->map(function ($lines) {

return LogEntry::fromLines($lines);

})->each(function (LogEntry $logEntry) {

// 处理日志项...

});

或者，假设你需要遍历 10000 个 Eloquent 模型。当使用传统的 Laravel 集合时，全部 10000 个 Eloquent 模型必须同时加载到内存中。

$users = App\User::all()->filter(function ($user) {

return $user->id > 500;

});

然而，查询构建器的 cursor 方法返回一个 LazyCollection 实例。这允许你对数据库只进行一次查询， 而且一次只能在内存中加载一个 Eloquent 模型。在此例中， filter 回调只有在我们实际逐个迭代每个用户之后才会执行，这样可以大幅度减少内存使用。

$users = App\User::cursor()->filter(function ($user) {

return $user->id > 500;

});

foreach ($users as $user) {

echo $user->id;

}

(2)、创建懒集合

要创建懒集合，应将 PHP 生成器函数传递给集合的 make 方法。

use Illuminate\Support\LazyCollection;

LazyCollection::make(function () {

$handle = fopen('log.txt', 'r');

while (($line = fgets($handle)) !== false) {

yield $line;

}

});

//

(3)、可枚举契约 (Enumerable 契约)

Collection 类中几乎所有的方法在 LazyCollection 类中也能用。这两个类都实现了 Illuminate\Support\Enumerable 契约，该契约定义了以下方法。

//2-1/ all

https://learnku.com/docs/laravel/6.x/collections/5161#method-all

//

//2-2/ average

https://learnku.com/docs/laravel/6.x/collections/5161#method-average

//

//2-3/ avg

https://learnku.com/docs/laravel/6.x/collections/5161#method-avg

//

//2-4/ chunk

https://learnku.com/docs/laravel/6.x/collections/5161#method-chunk

//

//2-5/ collapse

https://learnku.com/docs/laravel/6.x/collections/5161#method-collapse

//

//2-6/ collect

https://learnku.com/docs/laravel/6.x/collections/5161#method-collect

//

//2-7/ combine

https://learnku.com/docs/laravel/6.x/collections/5161#method-combine

//

//2-8/ concat

https://learnku.com/docs/laravel/6.x/collections/5161#method-concat

//

//2-9/ contains

https://learnku.com/docs/laravel/6.x/collections/5161#method-contains

//

//2-10/ containsStrict

https://learnku.com/docs/laravel/6.x/collections/5161#method-containsstrict

//

//2-11/ count

https://learnku.com/docs/laravel/6.x/collections/5161#method-count

//

//2-12/ countBy

https://learnku.com/docs/laravel/6.x/collections/5161#method-countBy

//

//2-13/ crossJoin

https://learnku.com/docs/laravel/6.x/collections/5161#method-crossjoin

//

//2-14/ dd

https://learnku.com/docs/laravel/6.x/collections/5161#method-dd

//

//2-15/ diff

https://learnku.com/docs/laravel/6.x/collections/5161#method-diff

//

//2-16/ diffAssoc

https://learnku.com/docs/laravel/6.x/collections/5161#method-diffassoc

//

//2-17/ diffKeys

https://learnku.com/docs/laravel/6.x/collections/5161#method-diffkeys

//

//2-18/ dump

https://learnku.com/docs/laravel/6.x/collections/5161#method-dump

//

//2-19/ duplicates

https://learnku.com/docs/laravel/6.x/collections/5161#method-duplicates

//

//2-20/ duplicatesStrict

https://learnku.com/docs/laravel/6.x/collections/5161#method-duplicatesstrict

//

//2-21/ each

https://learnku.com/docs/laravel/6.x/collections/5161#method-each

//

//2-22/ eachSpread

https://learnku.com/docs/laravel/6.x/collections/5161#method-eachspread

//

//2-23/ every

https://learnku.com/docs/laravel/6.x/collections/5161#method-every

//

//2-24/ except

https://learnku.com/docs/laravel/6.x/collections/5161#method-except

//

//2-25/ filter

https://learnku.com/docs/laravel/6.x/collections/5161#method-filter

//

//2-26/ first

https://learnku.com/docs/laravel/6.x/collections/5161#method-first

//

//2-27/ firstWhere

https://learnku.com/docs/laravel/6.x/collections/5161#method-first-where

//

//2-28/ flatMap

https://learnku.com/docs/laravel/6.x/collections/5161#method-flatmap

//

//2-29/ flatten

https://learnku.com/docs/laravel/6.x/collections/5161#method-flatten

//

//2-30/ flip

https://learnku.com/docs/laravel/6.x/collections/5161#method-flip

//

//2-31/ forPage

https://learnku.com/docs/laravel/6.x/collections/5161#method-forpage

//

//2-32/ get

https://learnku.com/docs/laravel/6.x/collections/5161#method-get

//

//2-33/ groupBy

https://learnku.com/docs/laravel/6.x/collections/5161#method-groupby

//

//2-34/ has

https://learnku.com/docs/laravel/6.x/collections/5161#method-has

//

//2-35/ implode

https://learnku.com/docs/laravel/6.x/collections/5161#method-implode

//

//2-36/ intersect

https://learnku.com/docs/laravel/6.x/collections/5161#method-intersect

//

//2-37/ intersectByKeys

https://learnku.com/docs/laravel/6.x/collections/5161#method-intersectbykeys

//

//2-38/ isEmpty

https://learnku.com/docs/laravel/6.x/collections/5161#method-isempty

//

//2-39/ isNotEmpty

https://learnku.com/docs/laravel/6.x/collections/5161#method-isnotempty

//

//2-40/ join

https://learnku.com/docs/laravel/6.x/collections/5161#method-join

//

//2-41/ keyBy

https://learnku.com/docs/laravel/6.x/collections/5161#method-keyby

//

//2-42/ keys

https://learnku.com/docs/laravel/6.x/collections/5161#method-keys

//

//2-43/ last

https://learnku.com/docs/laravel/6.x/collections/5161#method-last

//

//2-44/ macro

https://learnku.com/docs/laravel/6.x/collections/5161#method-macro

//

//2-45/ make

https://learnku.com/docs/laravel/6.x/collections/5161#method-make

//

//2-46/ map

https://learnku.com/docs/laravel/6.x/collections/5161#method-map

//

//2-47/ mapInto

https://learnku.com/docs/laravel/6.x/collections/5161#method-mapinto

//

//2-48/ mapSpread

https://learnku.com/docs/laravel/6.x/collections/5161#method-mapspread

//

//2-49/ mapToGroups

https://learnku.com/docs/laravel/6.x/collections/5161#method-maptogroups

//

//2-50/ mapWithKeys

https://learnku.com/docs/laravel/6.x/collections/5161#method-mapwithkeys

//

//2-51/ max

https://learnku.com/docs/laravel/6.x/collections/5161#method-max

//

//2-52/ median

https://learnku.com/docs/laravel/6.x/collections/5161#method-median

//

//2-53/ merge

https://learnku.com/docs/laravel/6.x/collections/5161#method-merge

//

//2-54/ mergeRecursive

https://learnku.com/docs/laravel/6.x/collections/5161#method-mergerecursive

//

//2-55/ min

https://learnku.com/docs/laravel/6.x/collections/5161#method-min

//

//2-56/ mode

https://learnku.com/docs/laravel/6.x/collections/5161#method-mode

//

//2-57/ nth

https://learnku.com/docs/laravel/6.x/collections/5161#method-nth

//

//2-58/ only

https://learnku.com/docs/laravel/6.x/collections/5161#method-only

//

//2-59/ pad

https://learnku.com/docs/laravel/6.x/collections/5161#method-pad

//

//2-60/ partition

https://learnku.com/docs/laravel/6.x/collections/5161#method-partition

//

//2-61/ pipe

https://learnku.com/docs/laravel/6.x/collections/5161#method-pipe

//

//2-62/ pluck

https://learnku.com/docs/laravel/6.x/collections/5161#method-pluck

//

//2-63/ random

https://learnku.com/docs/laravel/6.x/collections/5161#method-random

//

//2-64/ reduce

https://learnku.com/docs/laravel/6.x/collections/5161#method-reduce

//

//2-65/ reject

https://learnku.com/docs/laravel/6.x/collections/5161#method-reject

//

//2-66/ replace

https://learnku.com/docs/laravel/6.x/collections/5161#method-replace

//

//2-67/ replaceRecursive

https://learnku.com/docs/laravel/6.x/collections/5161#method-replacerecursive

//

//2-68/ reverse

https://learnku.com/docs/laravel/6.x/collections/5161#method-reverse

//

//2-69/ search

https://learnku.com/docs/laravel/6.x/collections/5161#method-search

//

//2-70/ shuffle

https://learnku.com/docs/laravel/6.x/collections/5161#method-shuffle

//

//2-71/ skip

https://learnku.com/docs/laravel/6.x/collections/5161#method-skip

//

//2-72/ slice

https://learnku.com/docs/laravel/6.x/collections/5161#method-slice

//

//2-73/ some

https://learnku.com/docs/laravel/6.x/collections/5161#method-some

//

//2-74/ sort

https://learnku.com/docs/laravel/6.x/collections/5161#method-sort

//

//2-75/ sortBy

https://learnku.com/docs/laravel/6.x/collections/5161#method-sortby

//

//2-76/ sortByDesc

https://learnku.com/docs/laravel/6.x/collections/5161#method-sortbydesc

//

//2-77/ sortKeys

https://learnku.com/docs/laravel/6.x/collections/5161#method-sortkeys

//

//2-78/ sortKeysDesc

https://learnku.com/docs/laravel/6.x/collections/5161#method-sortkeysdesc

//

//2-79/ split

https://learnku.com/docs/laravel/6.x/collections/5161#method-split

//

//2-80/ sum

https://learnku.com/docs/laravel/6.x/collections/5161#method-sum

//

//2-81/ take

https://learnku.com/docs/laravel/6.x/collections/5161#method-take

//

//2-82/ tap

https://learnku.com/docs/laravel/6.x/collections/5161#method-tap

//

//2-83/ times

https://learnku.com/docs/laravel/6.x/collections/5161#method-times

//

//2-84/ toArray

https://learnku.com/docs/laravel/6.x/collections/5161#method-toarray

//

//2-85/ toJson

https://learnku.com/docs/laravel/6.x/collections/5161#method-tojson

//

//2-86/ union

https://learnku.com/docs/laravel/6.x/collections/5161#method-union

//

//2-87/ unique

https://learnku.com/docs/laravel/6.x/collections/5161#method-unique

//

//2-88/ uniqueStrict

https://learnku.com/docs/laravel/6.x/collections/5161#method-uniquestrict

//

//2-89/ unless

https://learnku.com/docs/laravel/6.x/collections/5161#method-unless

//

//2-90/ unlessEmpty

https://learnku.com/docs/laravel/6.x/collections/5161#method-unlessempty

//

//2-91/ unlessNotEmpty

https://learnku.com/docs/laravel/6.x/collections/5161#method-unlessnotempty

//

//2-92/ unwrap

https://learnku.com/docs/laravel/6.x/collections/5161#method-unwrap

//

//2-93/ values

https://learnku.com/docs/laravel/6.x/collections/5161#method-values

//

//2-94/ when

https://learnku.com/docs/laravel/6.x/collections/5161#method-when

//

//2-95/ whenEmpty

https://learnku.com/docs/laravel/6.x/collections/5161#method-whenempty

//

//2-96/ whenNotEmpty

https://learnku.com/docs/laravel/6.x/collections/5161#method-whennotempty

//

//2-97/ where

https://learnku.com/docs/laravel/6.x/collections/5161#method-where

//

//2-98/ whereStrict

https://learnku.com/docs/laravel/6.x/collections/5161#method-wherestrict

//

//2-99/ whereBetween

https://learnku.com/docs/laravel/6.x/collections/5161#method-wherebetween

//

//2-100/ whereIn

https://learnku.com/docs/laravel/6.x/collections/5161#method-whereinhttps://learnku.com/docs/laravel/6.x/collections/5161#method-wherein

//

//2-101/ whereInStrict

https://learnku.com/docs/laravel/6.x/collections/5161#method-whereinstrict

//

//2-102/ whereInstanceOf

https://learnku.com/docs/laravel/6.x/collections/5161#method-whereinstanceof

--

//2-103/ whereNotBetween

https://learnku.com/docs/laravel/6.x/collections/5161#method-wherenotbetween

//

//2-104/ whereNotIn

https://learnku.com/docs/laravel/6.x/collections/5161#method-wherenotin

//

//2-105/ whereNotInStrict

https://learnku.com/docs/laravel/6.x/collections/5161#method-wherenotinstrict

//

//2-106/ wrap

https://learnku.com/docs/laravel/6.x/collections/5161#method-wrap

//

//2-107/ zip

https://learnku.com/docs/laravel/6.x/collections/5161#method-zip

注意：LazyCollection 类不提供使集合变动的方法（如 “shift”、“pop”、“prepend” 等）。

(4)、懒集合方法 (惰性集合方法)

除了在 Enumerable 契约中定义的方法外，LazyCollection 类还包含以下方法。

///1/ tapEach()

当 each 方法为集合中每一个集合项调用给定回调时， tapEach 方法仅调用给定回调，因为这些集合项正在逐个从列表中拉出。

$lazyCollection = LazyCollection::times(INF)->tapEach(function ($value) {

dump($value);

});

// 到目前为止，没有任何东西被抛弃

$array = $lazyCollection->take(3)->all();

// 1

// 2

// 3

--

--

五、事件系统 (Events)

--

https://learnku.com/docs/laravel/6.x/events/5162

--

1、简介 (事件系统介绍)

Laravel 的事件提供了一个简单的观察者实现，允许你在应用中订阅和监听各种发生的事件。事件类通常放在 app/Events 目录下，而这些事件类的监听器则放在 app/Listeners 目录下。如果在你的应用中你没有看到这些目录，不用担心，它们会在你使用 Artisan 控制台命令生成事件与监听器的时候自动创建。

事件系统为应用各个方面的解耦提供了非常棒的方法，因为单个事件可以拥有多个互不依赖的监听器。举个例子，你可能希望每次订单发货时向用户推送一个 Slack 通知。你可以简单地发起一个可以被监听器接收并转化为 Slack 通知的 OrderShipped 事件，而不是将订单处理代码和 Slack 通知代码耦合在一起。

--

2、注册事件 & 监听器 (注册事件和监听器)

Laravel 应用中的 EventServiceProvider 为注册所有的事件监听器提供了一个便利的场所。其中， listen 属性包含了所有事件 (键) 以及事件对应的监听器 (值) 的数组。当然，你可以根据应用的需要，添加多个事件到 listen 属性包含的数组中。举个例子，我们来添加一个 OrderShipped 事件。

/\*\*

\* 应用程序的事件监听器映射

\*

\* @var array

\*/

protected $listen = [

'App\Events\OrderShipped' => [

'App\Listeners\SendShipmentNotification',

],

];

(1)、生成事件 & 监听器

当然，手动创建事件和监听器的文件是件麻烦事。而在这里，你只需要将监听器和事件添加到 EventServiceProvider 中，而后使用 event:generate 命令。这个命令会生成在 EventServiceProvider 中列出的所有事件和监听器。当然，已经存在的事件和监听器将保持不变。

php artisan event:generate

(2)、手动注册事件

通常，事件是在 EventServiceProvider 的 $listen 数组中注册；然而，你也可以在 EventServiceProvider 的 boot 方法中手动注册基于闭包的这些事件。

/\*\*

\* 注册应用中的其它事件

\*

\* @return void

\*/

public function boot()

{

parent::boot();

Event::listen('event.name', function ($foo, $bar) {

//

});

}

//

///1/ 通配符事件监听器

你可以在注册监听器时使用 \* 作为通配符参数，这样可以在同一个监听器上捕获多个事件。通配符监听器接收事件名作为其第一个参数，并将整个事件数据数组作为其第二个参数。

Event::listen('event.\*', function ($eventName, array $data) {

//

});

(3)、事件发现

您可以启用自动事件发现，而不是在 EventServiceProvider 的 $listen 数组中手动注册事件和监听器。 启用事件发现后，Laravel 将通过扫描应用程序的 Listeners 目录自动查找并注册您的事件和侦听器。 此外，EventServiceProvider 中列出的任何明确定义的事件仍将被注册。

Laravel 通过使用反射扫描监听器类来查找事件监听器。 当 Laravel 找到以 handle 开头的监听器类方法时，Laravel 会将这些方法注册为方法签名中类型提示的事件的事件监听器。

use App\Events\PodcastProcessed;

class SendPodcastProcessedNotification

{

/\*\*

\* 处理给定的事件。

\*

\* @param \App\Events\PodcastProcessed

\* @return void

\*/

public function handle(PodcastProcessed $event)

{

//

}

}

默认情况下禁用事件发现，但您可以通过覆盖应用程序的 EventServiceProvider 的 shouldDiscoverEvents 方法来启用它。

/\*\*

\* 确定是否应自动发现事件和侦听器。

\*

\* @return bool

\*/

public function shouldDiscoverEvents()

{

return true;

}

默认情况下，将扫描应用程序的 Listeners 目录中的所有监听器。 如果要定义扫描的其他目录，可以覆盖 EventServiceProvider 中的 discoverEventsWithin 方法。

/\*\*

\* 获取应该用于发现事件的监听器目录

\*

\* @return array

\*/

protected function discoverEventsWithin()

{

return [

$this->app->path('Listeners'),

];

}

在生产中，您可能不希望框架在每个请求上扫描所有监听器。 因此，在部署过程中，您应该运行 event：cache Artisan 命令来缓存应用程序的所有事件和监听器的列表。 框架将使用此列表来加速事件注册过程。 event：clear 命令可用于销毁缓存。

提示：event:list 命令可用于显示应用程序注册的所有事件和监听器的列表。

--

3、定义事件

事件类是一个保存与事件相关信息的容器。例如，假设我们生成的 OrderShipped 事件接收一个 Eloquent ORM 对象( https://learnku.com/docs/laravel/6.x/eloquent )。

//

///1/ OrderShipped事件

<?php

namespace App\Events;

use App\Order;

use Illuminate\Queue\SerializesModels;

class OrderShipped

{

use SerializesModels;

public $order;

/\*\*

\* 创建一个事件实例

\*

\* @param \App\Order $order

\* @return void

\*/

public function \_\_construct(Order $order)

{

$this->order = $order;

}

}

?>

如你所见，这个事件类中没有包含其它逻辑。它只是一个购买的 Order 的实例的容器。如果使用 PHP 的 serialize 函数序列化事件对象，事件使用的 SerializesModels trait 将会优雅地序列化任何 Eloquent 模型。

--

4、定义监听器

接下来，让我们看一下例子中事件的监听器。事件监听器在 handle 方法中接收实例。 event:generate 命令会自动加载正确的事件类，并且在 handle 方法中加入事件的类型提示。在 handle 方法中，你可以执行任何必要的响应事件的操作。

//

///1/ OrderShipped事件监听器

<?php

namespace App\Listeners;

use App\Events\OrderShipped;

class SendShipmentNotification

{

/\*\*

\* 创建事件监听器

\*

\* @return void

\*/

public function \_\_construct()

{

//

}

/\*\*

\* 处理事件

\*

\* @param \App\Events\OrderShipped $event

\* @return void

\*/

public function handle(OrderShipped $event)

{

// 使用 $event->order 来访问 order ...

}

}

?>

提示：你的事件监听器也可以在构造函数中加入任何依赖关系的类型提示。所有的事件监听器都是通过 Laravel 的 服务容器( https://learnku.com/docs/laravel/6.x/container )解析的，因此所有的依赖都将会被自动注入。

//

///2/ 停止事件传播

有时，你可以通过在监听器的 handle 方法中返回 false 来阻止事件被其它的监听器获取。

--

5、事件监听器队列 ^2\_3^

如果你的监听器中要执行诸如发送电子邮件或发出 HTTP 请求之类的耗时任务，你可以将任务丢给队列处理。在开始使用队列监听器之前，请确保在你的服务器或者本地开发环境中能够 配置队列( https://learnku.com/docs/laravel/6.x/queues )并启动一个队列监听器。

要指定监听器启动队列，你可以在监听器类中实现 ShouldQueue 接口。由 Artisan 命令 event:generate 生成的监听器已经将此接口导入到当前命名空间中，因此你可以直接使用。

<?php

namespace App\Listeners;

use App\Events\OrderShipped;

use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue

{

//

}

?>

That's it！ 现在，当这个监听器被事件调用时，事件调度器会自动使用 Laravel 的队列系统( https://learnku.com/docs/laravel/6.x/queues )自动排队。如果在队列中执行监听器时没有抛出异常，任务会在执行完成后自动从队列中删除。

//

///1/ 自定义队列连接 & 队列名称

如果你想要自定义事件监听器所使用的队列的连接和名称，你可以在监听器类中定义 $connection，$queue 或 $delay 属性。

<?php

namespace App\Listeners;

use App\Events\OrderShipped;

use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue

{

/\*\*

\* 任务连接名称。

\*

\* @var string|null

\*/

public $connection = 'sqs';

/\*\*

\* 任务发送到的队列的名称.

\*

\* @var string|null

\*/

public $queue = 'listeners';

/\*\*

\* 处理任务的延迟时间.

\*

\* @var int

\*/

public $delay = 60;

}

?>

//

///2/ 条件监听队列

有时，你可能需要根据某些运行时的数据（满足某些条件）对监听器进行排队， 为此，可以在侦听器中添加 shouldQueue 方法，以确定是否应该将监听器排队并同步执行。

<?php

namespace App\Listeners;

use App\Events\OrderPlaced;

use Illuminate\Contracts\Queue\ShouldQueue;

class RewardGiftCard implements ShouldQueue

{

/\*\*

\* 奖励礼品卡给客户

\*

\* @param \App\Events\OrderPlaced $event

\* @return void

\*/

public function handle(OrderPlaced $event)

{

//

}

/\*\*

\* 确定监听器是否应加入队列

\*

\* @param \App\Events\OrderPlaced $event

\* @return bool

\*/

public function shouldQueue(OrderPlaced $event)

{

return $event->order->subtotal >= 5000;

}

}

?>

(1)、手动访问队列

如果你需要手动访问监听器下面队列任务的 delete 和 release 方法，你可以通过使用 Illuminate\Queue\InteractsWithQueue trait 来实现。这个 trait 会默认加载到生成的监听器中，并提供对这些方法的访问。

<?php

namespace App\Listeners;

use App\Events\OrderShipped;

use Illuminate\Queue\InteractsWithQueue;

use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue

{

use InteractsWithQueue;

/\*\*

\* 处理事件.

\*

\* @param \App\Events\OrderShipped $event

\* @return void

\*/

public function handle(OrderShipped $event)

{

if (true) {

$this->release(30);

}

}

}

?>

(2)、处理失败任务

有时事件监听器的队列任务可能会失败。如果监听器的队列任务超过了队列中定义的最大尝试次数，则会在监听器上调用 failed 方法。 failed 方法接收事件实例和导致失败的异常作为参数。

<?php

namespace App\Listeners;

use App\Events\OrderShipped;

use Illuminate\Queue\InteractsWithQueue;

use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue

{

use InteractsWithQueue;

/\*\*

\* 处理事件.

\*

\* @param \App\Events\OrderShipped $event

\* @return void

\*/

public function handle(OrderShipped $event)

{

//

}

/\*\*

\* 处理失败任务

\*

\* @param \App\Events\OrderShipped $event

\* @param \Exception $exception

\* @return void

\*/

public function failed(OrderShipped $event, $exception)

{

//

}

}

?>

--

6、分发事件

如果要分发事件，你可以将事件实例传递给辅助函数 event。该辅助函数将会把事件分发到所有该事件相应的已经注册了的监听器上。event 辅助函数可以全局使用，你可以在应用中的任何位置进行调用。

<?php

namespace App\Http\Controllers;

use App\Order;

use App\Events\OrderShipped;

use App\Http\Controllers\Controller;

class OrderController extends Controller

{

/\*\*

\* 将传递过来的订单发货

\*

\* @param int $orderId

\* @return Response

\*/

public function ship($orderId)

{

$order = Order::findOrFail($orderId);

// Order shipment logic...

event(new OrderShipped($order));

}

}

?>

Tip：在测试时，只需要断言特定事件被分发，而不需要真正地触发监听器。 Laravel 的内置测试辅助函数( https://learnku.com/docs/laravel/6.x/mocking#event-fake ) 可以轻松做到这一点。

--

7、事件订阅者

(1)、编写事件订阅者

事件订阅者是可以在自身内部订阅多个事件的类，即能够在单个类中定义多个事件处理器。订阅者应该定义一个 subscribe 方法，这个方法接收一个事件分发器实例。你可以调用给定事件分发器上的 listen 方法来注册事件监听器。

<?php

namespace App\Listeners;

class UserEventSubscriber

{

/\*\*

\* 处理用户登录事件

\*/

public function handleUserLogin($event) {}

/\*\*

\* 处理用户注销事件

\*/

public function handleUserLogout($event) {}

/\*\*

\* 为订阅者注册监听器.

\*

\* @param \Illuminate\Events\Dispatcher $events

\*/

public function subscribe($events)

{

$events->listen(

'Illuminate\Auth\Events\Login',

'App\Listeners\UserEventSubscriber@handleUserLogin'

);

$events->listen(

'Illuminate\Auth\Events\Logout',

'App\Listeners\UserEventSubscriber@handleUserLogout'

);

}

}

?>

(2)、注册事件订阅者

在编写完订阅者之后，就可以通过事件分发器对订阅者进行注册。你可以在 EventServiceProvider 中的 $subscribe 属性中注册订阅者。例如，让我们将 UserEventSubscriber 添加到数组列表中。

<?php

namespace App\Providers;

use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvider;

class EventServiceProvider extends ServiceProvider

{

/\*\*

\* 应用中事件监听器的映射。

\*

\* @var array

\*/

protected $listen = [

//

];

/\*\*

\* 需要注册的订阅者类。

\*

\* @var array

\*/

protected $subscribe = [

'App\Listeners\UserEventSubscriber',

];

}

?>

--

--

六、文件存储

--

https://learnku.com/docs/laravel/6.x/filesystem/5163

--

1、简介

Laravel 提供了一个强大的文件系统抽象概念，这得益于 Frank de Jonge 强大的 Flysystem ( https://github.com/thephpleague/flysystem ) 扩展包。Laravel 文件系统集成为使用本地文件系统、Amazon S3 和 Rackspace 云存储提供了简单易用的驱动程序。更棒的是，由于每个系统的 API 保持不变，所以在这些存储选项之间切换是非常简单的。

文件系统的配置文件位于 config/filesystems.php。在这个文件中你可以配置所有「磁盘」。每个磁盘代表特定的存储驱动及存储位置。每种支持的驱动程序的示例配置都包含在配置文件中。因此，只需要修改配置即可反映你的存储偏好和凭据。

当然，你可以根据需要配置多个磁盘，甚至你还可以使多个磁盘共用同一个驱动。

--

2、配置

(1)、公共磁盘

public 磁盘适用于要公开访问的文件。默认情况下， public 磁盘使用 local 驱动，并且将这些文件存储在 storage/app/public 目录下。为了使它们能通过网络访问，你需要创建 public/storage 到 storage/app/public 的符号链接。这种方式能把可公开访问文件都保留在同一个目录下，以便在使用零停机时间部署系统如 Envoyer 的时候，就可以轻松地在不同的部署之间共享这些文件。

//

// 你可以使用 Artisan 命令 storage:link 来创建符号链接。

php artisan storage:link

//

当然，一旦一个文件被存储并且已经创建了符号链接，你就可以使用辅助函数 asset 来创建文件的 URL。

echo asset('storage/file.txt');

(2)、本地驱动

使用 local 驱动时，所有文件操作都与 filesystems 配置文件中定义的 root 目录相关。 默认情况下，此值设置为 storage/app 目录。 因此，以下方法会把文件存储在 storage/app/file.txt 中。

Storage::disk('local')->put('file.txt', 'Contents');

//

///1/ 权限

public 可见性 转换为目录的 0755 和文件的 0644。 你可以修改 filesystems 配置文件中的权限映射。

'local' => [

'driver' => 'local',

'root' => storage\_path('app'),

'permissions' => [

'file' => [

'public' => 0664,

'private' => 0600,

],

'dir' => [

'public' => 0775,

'private' => 0700,

],

],

],

(3)、驱动先决条件

///1/ Composer 包

在使用 SFTP、S3 驱动之前，你需要通过 Composer 安装相应的软件包。

SFTP: league/flysystem-sftp ~1.0

Amazon S3: league/flysystem-aws-s3-v3 ~1.0

//

使用缓存适配器是提高性能的一个绝对必要条件。你需要一个额外的包。

CachedAdapter: league/flysystem-cached-adapter ~1.0

//

///2/ S3 驱动配置

S3 驱动配置信息位于你的 config/filesystems.php 配置文件中。该文件包含 S3 驱动程序的示例配置数组。 你可以自由使用你自己的 S3 配置和凭证修改此阵列。 为方便起见，这些环境变量与 AWS CLI 使用的命名约定相匹配。

//

///3/ FTP 驱动配置

Laravel 的文件系统集成能很好的支持 FTP，不过 FTP 的配置示例并没有被包含在框架默认的 filesystems.php 文件中。需要的话可以使用下面的示例配置。

'ftp' => [

'driver' => 'ftp',

'host' => 'ftp.example.com',

'username' => 'your-username',

'password' => 'your-password',

// 可选的 FTP 配置项...

// 'port' => 21,

// 'root' => '',

// 'passive' => true,

// 'ssl' => true,

// 'timeout' => 30,

],

//

///4/ SFTP 驱动器的配置

Laravel 的 Flysystem 集成包与 SFTP 协同得非常好；不过，在该框架的默认配置文件 filesystems.php 中并没有包含示范配置。如果要配置 SFTP 文件系统，可以使用如下示例配置。

'sftp' => [

'driver' => 'sftp',

'host' => 'example.com',

'username' => 'your-username',

'password' => 'your-password',

// 基于 SSH 密钥的身份验证设置...

// 'privateKey' => '/path/to/privateKey',

// 'password' => 'encryption-password',

// 可选的 SFTP 配置...

// 'port' => 22,

// 'root' => '',

// 'timeout' => 30,

],

(4)、缓存

给指定磁盘开启缓存功能，需要在该磁盘的配置项中直接添加 cache 。 cache 选项应该是一个缓存配置的数组，由缓存驱动名称 store （译者注：文档原始描述文字 disk 与示例代码中的 store 不一致，验证代码后的确应该是 store ，故作此修改。）、 单位为秒的过期时间 expire ，以及缓存前缀 prefix 组成。

's3' => [

'driver' => 's3',

// 驱动器其他配置...

'cache' => [

'store' => 'memcached',

'expire' => 600,

'prefix' => 'cache-prefix',

],

],

--

3、获取磁盘实例

Storage 门面可用于与任何已配置的磁盘进行交互。例如，你可以使用门面中的 put 方法将头像存储到默认磁盘。如果你使用 Storage 门面中的任何方法，而一开始并没有使用 disk 方法，那么所调用的方法会自动传递给默认的磁盘。

use Illuminate\Support\Facades\Storage;

Storage::put('avatars/1', $fileContents);

//

如果应用程序要与多个磁盘进行互操作，可使用 Storage 门面中的 disk 方法对特定磁盘上的文件进行操作。

Storage::disk('s3')->put('avatars/1', $fileContents);

4、检索文件

get 方法可以用于检索文件的内容，此方法返回该文件的原始字符串内容。 切记，所有文件路径的指定都应该相对于为磁盘配置的「root」目录。

$contents = Storage::get('file.jpg');

//

exists 方法可以用来判断磁盘上是否存在指定的文件。

$exists = Storage::disk('s3')->exists('file.jpg');

(1)、下载文件

download 方法可用于生成一个响应，强制用户的浏览器在给定路径下载文件。 download 方法接受一个文件名作为该方法的第二个参数，它将确定用户下载文件时看到的文件名。最后，你可以传递一个 HTTP 数组头作为该方法的第三个参数。

return Storage::download('file.jpg');

return Storage::download('file.jpg', $name, $headers);

(2)、文件 URLs

你可以使用 url 方法来获取给定文件的 URL。如果你使用的是 local 驱动，一般只是在给定的路径上加上 /storage 并返回一个相对的 URL 到那个文件。如果使用的是 s3 或者是 rackspace 驱动，会返回完整的远程 URL。

use Illuminate\Support\Facades\Storage;

$url = Storage::url('file.jpg');

//

注意：切记，如果使用的是 local 驱动，则所有想被公开访问的文件都应该放在 storage/app/public 目录下。此外你应该在 public/storage 创建一个符号链接( https://learnku.com/docs/laravel/6.x/filesystem/5163#the-public-disk ) 来指向 storage/app/public 目录。

//

///1/ 临时 URLs

对于使用 s3 驱动来存储文件，可以使用 temporaryUrl 方法创建给定文件的临时 URL。这个方法会接收路径和 DateTime 实例来指定 URL 何时过期。

$url = Storage::temporaryUrl(

'file.jpg', now()->addMinutes(5)

);

如果你需要指定其他 S3请求参数( https://docs.aws.amazon.com/AmazonS3/latest/API/RESTObjectGET.html#RESTObjectGET-requests )，你可以传递一个数组作为 temporaryUrl 方法的第三个参数。

$url = Storage::temporaryUrl(

'file.jpg',

now()->addMinutes(5),

['ResponseContentType' => 'application/octet-stream']

);

//

///2/ 自定义本地 URL 主机

如果要使用 local 驱动为存储在磁盘上的文件预定义主机，可以向磁盘配置数组添加一个 url 选项。

'public' => [

'driver' => 'local',

'root' => storage\_path('app/public'),

'url' => env('APP\_URL').'/storage',

'visibility' => 'public',

],

(3)、文件元数据

除了读写文件外，Laravel 还可以提供有关文件本身的信息，例如，size 方法可用来获取文件的大小（以字节为单位）。

use Illuminate\Support\Facades\Storage;

$size = Storage::size('file.jpg');

//

lastModified 方法返回文件最后一次被修改的 UNIX 时间戳。

$time = Storage::lastModified('file.jpg');

--

5、保存文件

put 方法可用于将原始文件内容保存到磁盘上。你也可以传递 PHP 的 resource 给 put 方法，它将使用文件系统下的底层流支持。强烈建议在处理大文件时使用此方法。

use Illuminate\Support\Facades\Storage;

Storage::put('file.jpg', $contents);

Storage::put('file.jpg', $resource);

//

///1/ 自动流式传输

如果你想 Laravel 自动将给定文件流式传输到你的存储位置，你可以使用 putFile 或 putFileAs 方法。这个方法接收 Illuminate\Http\File 或 Illuminate\Http\UploadedFile 实例，并自动将文件流式传输到你想要传输的位置。

use Illuminate\Http\File;

use Illuminate\Support\Facades\Storage;

// 自动为文件名生成唯一的ID...

Storage::putFile('photos', new File('/path/to/photo'));

// 手动指定文件名...

Storage::putFileAs('photos', new File('/path/to/photo'), 'photo.jpg');

关于 putFile 方法，有些点需要注意。我们指定的是一个目录名而不是文件名。默认情况下， putFile 方法会生成一个唯一的 ID 作为文件名。文件的扩展名根据检测文件的 MIME 类型来确定。 putFile 方法会返回文件路径，以便你可以将文件路径（包括生成的文件名）存储在数据库中。

putFile 和 putFileAs 方法也接受一个方法来指定存储文件的 「可见性」。如果你将文件存储在诸如 S3 的云盘上，并且想让该文件公开访问，则可以使用以下功能。

Storage::putFile('photos', new File('/path/to/photo'), 'public');

//

///2/ 文件数据写入

prepend 和 append 方法允许你在文件的开头或结尾写入数据。

Storage::prepend('file.log', 'Prepended Text');

Storage::append('file.log', 'Appended Text');

//

///3/ 复制 & 移动文件

copy 方法用来复制文件到磁盘上的新位置，而 move 方法用来重命名或移动文件到新位置。

Storage::copy('old/file.jpg', 'new/file.jpg');

Storage::move('old/file.jpg', 'new/file.jpg');

(1)、文件上传

在 web 应用程序中，最常用到的文件存储的场景的地方就是上传头像，照片和文件。Laravel 上传文件的实例方法 store 可以轻松的处理文件上传存储问题。你只需要调用带有文件保存路径作为参数的 store 方法即可。

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Controllers\Controller;

class UserAvatarController extends Controller

{

/\*\*

\* 更新用户头像.

\*

\* @param Request $request

\* @return Response

\*/

public function update(Request $request)

{

$path = $request->file('avatar')->store('avatars');

return $path;

}

}

?>

上例有几个点需要注意。我们指定的是目录名，而不是文件名。默认情况下， store 方法会自动生成唯一的 ID 作为文件名。文件的扩展名将通过检查文件的 MIME 类型来确定。该文件的路径和文件名会被 store 方法返回，以便后续数据库的存储使用。

你也可以使用 Storage facade 上的 putFile 方法达到和上例同样的效果。

$path = Storage::putFile('avatars', $request->file('avatar'));

//

///1/ 指定文件名

如果你不想将文件名自动分配给存储的文件，可以使用 storeAs 方法，该方法接受路径，文件名和 (可选) 磁盘作为其参数。

$path = $request->file('avatar')->storeAs(

'avatars', $request->user()->id

);

你可以使用 Storage facade 上的 putFileAs 方法达到和上例同样的文件操作。

$path = Storage::putFileAs(

'avatars', $request->file('avatar'), $request->user()->id

);

注意：将自动从文件路径中删除不可打印和无效的 unicode 字符。 因此，你可能希望在将文件路径传递给 Laravel 的文件存储方法之前对其进行清理。 使用 League\Flysystem\Util::normalizePath 方法对文件路径进行规范化。

//

///2/ 指定磁盘

默认情况下， store 方法使用默认磁盘。如果你需要指定其他的磁盘，可以传入磁盘名作为 store 方法的第二个参数。

$path = $request->file('avatar')->store(

'avatars/'.$request->user()->id, 's3'

);

(2)、文件可见性

在 Laravel 集成的文件系统中，「可见性」是一个对多个平台的文件权限的抽象概念。文件可以声明为 public 或 private。如果一个文件被声明为 public，意味着其他人可以访问。例如，使用 S3 驱动时，就可以检索声明为 public 的文件。

你可以通过 put 方法设置文件的可见性。

use Illuminate\Support\Facades\Storage;

Storage::put('file.jpg', $contents, 'public');

//

getVisibility 和 setVisibility 方法可以对现存文件的可见性进行查询和设置。

$visibility = Storage::getVisibility('file.jpg');

Storage::setVisibility('file.jpg', 'public')

--

6、删除文件

delete 方法接收一个文件名或数组形式的文件名来删除磁盘上的文件。

use Illuminate\Support\Facades\Storage;

Storage::delete('file.jpg');

Storage::delete(['file.jpg', 'file2.jpg']);

//

必要的话，可以指定磁盘名来删除其下的文件。

use Illuminate\Support\Facades\Storage;

Storage::disk('s3')->delete('folder\_path/file\_name.jpg');

--

7、目录

///1/ 获取目录下的所有的文件

files 方法返回指定目录下的所有文件。如果你想检索指定目录（包括子目录）中所有的文件列表，可以使用 allFiles 方法。

use Illuminate\Support\Facades\Storage;

$files = Storage::files($directory);

$files = Storage::allFiles($directory);

//

///2/ 获取目录下的所有目录

directories 方法返回指定目录下所有目录的数组。此外，你可以使用 allDirectories 方法获取指定目录以及其子目录下所有目录的列表。

$directories = Storage::directories($directory);

// Recursive...

$directories = Storage::allDirectories($directory);

//

///3/ 创建目录

makeDirectory 方法会递归创建目录。

Storage::makeDirectory($directory);

//

///4/ 删除目录

deleteDirectory 方法会删除指定目录及其下所有的文件。

Storage::deleteDirectory($directory);

--

8、自定义文件系统

虽然 Laravel 的文件系统提供了一些开箱即用的驱动，但是它不仅限于这些，还提供了其他文件系统的适配器。通过这些适配器，可以在 Lavarel 应用中创建自定义驱动。

为了设置自定义文件系统，你需要一个 Flysystem 适配器。现在让我们把社区维护的 Dropbox 适配器添加到项目中。

composer require spatie/flysystem-dropbox

//

接下来，你需要创建一个名为 DropboxServiceProvider 的 服务提供者( https://learnku.com/docs/laravel/6.x/providers )。在它的 boot 方法中，使用 Storage facade 的 extend 方法自定义驱动。

<?php

namespace App\Providers;

use Storage;

use League\Flysystem\Filesystem;

use Illuminate\Support\ServiceProvider;

use Spatie\Dropbox\Client as DropboxClient;

use Spatie\FlysystemDropbox\DropboxAdapter;

class DropboxServiceProvider extends ServiceProvider

{

/\*\*

\* 执行注册后引导驱动.

\*

\* @return void

\*/

public function register()

{

//

}

/\*\*

\* 在容器中注册绑定.

\*

\* @return void

\*/

public function boot()

{

Storage::extend('dropbox', function ($app, $config) {

$client = new DropboxClient(

$config['authorization\_token']

);

return new Filesystem(new DropboxAdapter($client));

});

}

}

?>

extend 方法的第一个参数是驱动的名称，第二个参数是闭包，接受 $app 和 $config 变量。这个闭包必须返回 League\Flysystem\Filesystem 的实例。 $config 变量包含了磁盘在 config/filesystems.php 中的位置。

接下来，在 config/app.php 配置文件中注册服务提供者。

'providers' => [

// ...

App\Providers\DropboxServiceProvider::class,

];

当你创建并注册好服务提供者后，就可以通过在 config/filesystems.php 配置文件中添加 dropbox 驱动并使用它了。

--

--

七、辅助函数

--

https://learnku.com/docs/laravel/6.x/helpers/5164

--

1、简介

Laravel 包含各种各样的全局 PHP 「辅助」函数，框架本身也大量的使用了这些功能函数；如果你觉的方便，你可以在你的应用中任意使用这些函数。

--

2、可用方法 (方法列表)

(1)、数组 & 对象

//// Arr::add

如果给定的键在数组中不存在或数组被设置为 null ，那么 Arr::add 函数将会把给定的键值对添加到数组中。

use Illuminate\Support\Arr;

$array = Arr::add(['name' => 'Desk'], 'price', 100); // ['name' => 'Desk', 'price' => 100]

$array = Arr::add(['name' => 'Desk', 'price' => null], 'price', 100); // ['name' => 'Desk', 'price' => 100]

//

//// Arr::collapse

Arr::collapse 函数将多个数组合并为一个数组。

use Illuminate\Support\Arr;

$array = Arr::collapse([[1, 2, 3], [4, 5, 6], [7, 8, 9]]); // [1, 2, 3, 4, 5, 6, 7, 8, 9]

//

//// Arr::divide

Arr::divide 函数返回一个二维数组，一个值包含原始数组的键，另一个值包含原始数组的值。

use Illuminate\Support\Arr;

[$keys, $values] = Arr::divide(['name' => 'Desk']);

// $keys: ['name']

// $values: ['Desk']

//

//// Arr::dot

Arr::dot 函数将多维数组中所有的键平铺到一维数组中，新数组使用「.」符号表示层级包含关系。

use Illuminate\Support\Arr;

$array = ['products' => ['desk' => ['price' => 100]]];

$flattened = Arr::dot($array); // ['products.desk.price' => 100]

//

//// Arr::except

Arr::except 函数从数组中删除指定的键值对。

use Illuminate\Support\Arr;

$array = ['name' => 'Desk', 'price' => 100];

$filtered = Arr::except($array, ['price']); // ['name' => 'Desk']

//

//// Arr::first

Arr::first 函数返回数组中通过真值测试的第一个元素。

use Illuminate\Support\Arr;

$array = [100, 200, 300];

$first = Arr::first($array, function ($value, $key) {

return $value >= 150;

}); // 200

将默认值作为第三个参数传递给该方法， 如果数组中没有值通过真值测试，则返回默认值。

use Illuminate\Support\Arr;

$first = Arr::first($array, $callback, $default);

//

//// Arr::flatten

Arr::flatten 函数将多维数组中数组的值取出平铺为一维数组。

use Illuminate\Support\Arr;

$array = ['name' => 'Joe', 'languages' => ['PHP', 'Ruby']];

$flattened = Arr::flatten($array); // ['Joe', 'PHP', 'Ruby']

//

//// Arr::forget

Arr::forget 函数使用「.」符号从深度嵌套的数组中删除给定的键值对。

use Illuminate\Support\Arr;

$array = ['products' => ['desk' => ['price' => 100]]];

Arr::forget($array, 'products.desk'); // ['products' => []]

//

//// Arr::get

Arr::get 函数使用「.」符号从深度嵌套的数组中根据指定键检索值。

use Illuminate\Support\Arr;

$array = ['products' => ['desk' => ['price' => 100]]];

$price = Arr::get($array, 'products.desk.price'); // 100

Arr::get 函数也接受一个默认值，如果没有找到特定的键，将返回默认值。

use Illuminate\Support\Arr;

$discount = Arr::get($array, 'products.desk.discount', 0); // 0

//

//// Arr::has

Arr::has 函数使用「.」符号查找数组中是否存在指定的一个或多个键。

use Illuminate\Support\Arr;

$array = ['product' => ['name' => 'Desk', 'price' => 100]];

$contains = Arr::has($array, 'product.name'); // 真

$contains = Arr::has($array, ['product.price', 'product.discount']); // 假

//

//// Arr::last

Arr::last 函数返回数组中通过指定测试的最后一个元素。

use Illuminate\Support\Arr;

$array = [100, 200, 300, 110];

$last = Arr::last($array, function ($value, $key) {

return $value >= 150;

}); // 300

将默认值作为第三个参数传递给该方法，如果没有值通过真值测试，则返回该默认值。

use Illuminate\Support\Arr;

$last = Arr::last($array, $callback, $default);

//

//// Arr::only

Arr::only 函数只返回给定数组中指定的键值对。

use Illuminate\Support\Arr;

$array = ['name' => 'Desk', 'price' => 100, 'orders' => 10];

$slice = Arr::only($array, ['name', 'price']); // ['name' => 'Desk', 'price' => 100]

//

//// Arr::pluck

Arr::pluck 函数从数组中检索给定键的所有值。

use Illuminate\Support\Arr;

$array = [

['developer' => ['id' => 1, 'name' => 'Taylor']],

['developer' => ['id' => 2, 'name' => 'Abigail']],

];

$names = Arr::pluck($array, 'developer.name'); // ['Taylor', 'Abigail']

你也可以指定获取的结果的键。

use Illuminate\Support\Arr;

$names = Arr::pluck($array, 'developer.name', 'developer.id'); // [1 => 'Taylor', 2 => 'Abigail']

//

//// Arr::prepend

Arr::prepend 函数将一个值插入到数组的开始位置。

use Illuminate\Support\Arr;

$array = ['one', 'two', 'three', 'four'];

$array = Arr::prepend($array, 'zero'); // ['zero', 'one', 'two', 'three', 'four']

如果需要，你可以指定你插入值的键。

use Illuminate\Support\Arr;

$array = ['price' => 100];

$array = Arr::prepend($array, 'Desk', 'name'); // ['name' => 'Desk', 'price' => 100]

//

//// Arr::pull

Arr::pull 函数从数组中返回指定键的值并删除此键／值对。

use Illuminate\Support\Arr;

$array = ['name' => 'Desk', 'price' => 100];

$name = Arr::pull($array, 'name'); // $name: Desk; // $array: ['price' => 100];

默认值可以作为第三个参数传递给该方法，如果键不存在，则返回该值。

use Illuminate\Support\Arr;

$value = Arr::pull($array, $key, $default);

//

//// Arr::random

Arr::random 函数从数组中随机返回一个值。

use Illuminate\Support\Arr;

$array = [1, 2, 3, 4, 5];

$random = Arr::random($array); // 4 - (随机检索)

你也可以将返回值的数量作为可选的第二个参数传递给该方法，请注意，提供这个参数会返回一个数组，即便是你只需要一项。

use Illuminate\Support\Arr;

$items = Arr::random($array, 2); // [2, 5] - (随机检索)

//

//// Arr::set

Arr::set 函数使用「.」符号在多维数组中设置指定键的值。

use Illuminate\Support\Arr;

$array = ['products' => ['desk' => ['price' => 100]]];

Arr::set($array, 'products.desk.price', 200); // ['products' => ['desk' => ['price' => 200]]]

//

//// Arr::sort

Arr::sort 函数根据数组的值对数组进行排序。

use Illuminate\Support\Arr;

$array = ['Desk', 'Table', 'Chair'];

$sorted = Arr::sort($array); // ['Chair', 'Desk', 'Table']

你也可以根据给定闭包返回的结果对数组进行排序。

use Illuminate\Support\Arr;

$array = [

['name' => 'Desk'],

['name' => 'Table'],

['name' => 'Chair'],

];

$sorted = array\_values(Arr::sort($array, function ($value) {

return $value['name'];

}));

/\*

[

['name' => 'Chair'],

['name' => 'Desk'],

['name' => 'Table'],

]

\*/

//

//// Arr::sortRecursive

Arr::sortRecursive 函数使用 sort 函数对数值子数组进行递归排序，使用 ksort 函数对关联子数组进行递归排序。

use Illuminate\Support\Arr;

$array = [

['Roman', 'Taylor', 'Li'],

['PHP', 'Ruby', 'JavaScript'],

['one' => 1, 'two' => 2, 'three' => 3],

];

$sorted = Arr::sortRecursive($array);

/\*

[

['JavaScript', 'PHP', 'Ruby'],

['one' => 1, 'three' => 3, 'two' => 2],

['Li', 'Roman', 'Taylor'],

]

\*/

//

//// Arr::where

Arr::where 函数使用给定闭包返回的结果过滤数组。

use Illuminate\Support\Arr;

$array = [100, '200', 300, '400', 500];

$filtered = Arr::where($array, function ($value, $key) {

return is\_string($value);

}); // [1 => '200', 3 => '400']

//

//// Arr::wrap

Arr::wrap 函数将给定的值变为一个数组，如果给定的值已经是数组，则不改变。

use Illuminate\Support\Arr;

$string = 'Laravel';

$array = Arr::wrap($string); // ['Laravel']

//

如果给定的值是空，则返回一个空数组。

use Illuminate\Support\Arr;

$nothing = null;

$array = Arr::wrap($nothing); // []

//

//// data\_fill

data\_fill 函数使用「.」符号在多维数组或对象内设置缺省值。

$data = ['products' => ['desk' => ['price' => 100]]];

data\_fill($data, 'products.desk.price', 200); // ['products' => ['desk' => ['price' => 100]]]

data\_fill($data, 'products.desk.discount', 10); // ['products' => ['desk' => ['price' => 100, 'discount' => 10]]]

//

这个函数还接受星号「＊」作为通配符，相应的填充目标。

$data = [

'products' => [

['name' => 'Desk 1', 'price' => 100],

['name' => 'Desk 2'],

],

];

data\_fill($data, 'products.\*.price', 200);

/\*

[

'products' => [

['name' => 'Desk 1', 'price' => 100],

['name' => 'Desk 2', 'price' => 200],

],

]

\*/

//

//// data\_get

data\_get 函数使用「.」符号从多维数组或对象中检索值。

$data = ['products' => ['desk' => ['price' => 100]]];

$price = data\_get($data, 'products.desk.price'); // 100

//

data\_get 函数也可以接收一个默认值，如果找不到指定的键，则返回默认值。

$discount = data\_get($data, 'products.desk.discount', 0); // 0

//

这个函数还接受「＊」作为通配符，它可以匹配数组或对象的任何键。

$data = [

'product-one' => ['name' => 'Desk 1', 'price' => 100],

'product-two' => ['name' => 'Desk 2', 'price' => 150],

];

data\_get($data, '\*.name'); // ['Desk 1', 'Desk 2'];

//

//// data\_set

data\_set 函数使用「.」符号在多维数组或对象中设置一个值。

$data = ['products' => ['desk' => ['price' => 100]]];

data\_set($data, 'products.desk.price', 200); // ['products' => ['desk' => ['price' => 200]]]

//

该函数也可以接收「＊」通配符，相应的在指定键上设置值。

$data = [

'products' => [

['name' => 'Desk 1', 'price' => 100],

['name' => 'Desk 2', 'price' => 150],

],

];

data\_set($data, 'products.\*.price', 200);

/\*

[

'products' => [

['name' => 'Desk 1', 'price' => 200],

['name' => 'Desk 2', 'price' => 200],

],

]

\*/

默认情况下，所有现有值都会被覆盖， 如果你只希望设置不存在的值，你可以选择 false 作为第四个参数传递给该方法。

$data = ['products' => ['desk' => ['price' => 100]]];

data\_set($data, 'products.desk.price', 200, false); // ['products' => ['desk' => ['price' => 100]]]

//

//// head

head 函数返回给定数组中的第一个元素。

$array = [100, 200, 300];

$first = head($array); // 100

//

//// last

last 函数返回给定数组中的最后一个元素。

$array = [100, 200, 300];

$last = last($array); // 300

(2)、路径

//// app\_path

app\_path 函数返回 app 目录的完整路径．你也可以使用 app\_path 函数来设置应用程序 app 目录的完整路径。

$path = app\_path();

$path = app\_path('Http/Controllers/Controller.php');

//

//// base\_path

base\_path 函数返回项目根目录的完整路径．你也可以使用 base\_path 函数设置项目根目录的完整路径。

$path = base\_path();

$path = base\_path('vendor/bin');

//

//// config\_path

config\_path 函数返回 config 目录的完整路径。你也可以使用 config\_path 函数设置应用程序 config 目录中给定文件的完整路径。

$path = config\_path();

$path = config\_path('app.php');

//

//// database\_path

database\_path 函数返回 database 目录的完整路径。你也可以使用 database\_path 函数来设置 database 目录中给定文件的完整路径。

$path = database\_path();

$path = database\_path('factories/UserFactory.php');

//

//// mix

mix 函数返回 版本化 Mix 文件 ( https://learnku.com/docs/laravel/6.x/mix ) 的路径。

$path = mix('css/app.css');

//

//// public\_path

public\_path 函数返回 public 目录的完整路径。你也可以使用 public\_path 函数来生成 public 目录中给定文件的完整路径。

$path = public\_path();

$path = public\_path('css/app.css');

//

//// resource\_path

resource\_path 函数返回 resources 目录的完整路径。你也可以使用 resource\_path 函数来生成资源文件中给定文件的完整路径。

$path = resource\_path();

$path = resource\_path('sass/app.scss');

//

//// storage\_path

storage\_path 函数返回 storage 目录的完整路径。你也可以使用 storage\_path 函数来设置存储目录下指定文件的完整路径。

$path = storage\_path();

$path = storage\_path('app/file.txt');

(3)、字符串

//// \_\_()

函数使用你的 本地化文件 ( https://learnku.com/docs/laravel/6.x/localization ) 来翻译给定的翻译字符串或翻译键。

echo \_\_('Welcome to our application');

echo \_\_('messages.welcome');

如果指定的翻译字符串或翻译键不存在， \_\_ 函数将返回给定的值．所以，按照上面的例子，如果翻译键 messages.welcome 不存在， \_\_ 函数会将其直接返回。

//

//// class\_basename

class\_basename 函数返回被删除了命名空间的指定类的类名。

$class = class\_basename('Foo\Bar\Baz'); // Baz

//

//// e

e 函数将默认值为 true 的 double\_encode 选项值改为 false 来运行 PHP 的 htmlspecialchars 函数。

echo e('<html>foo</html>'); // &lt;html&gt;foo&lt;/html&gt;

//

//// preg\_replace\_array

preg\_replace\_array 函数使用数组顺序替换字符串中的给定模式。

$string = 'The event will take place between :start and :end';

$replaced = preg\_replace\_array('/:[a-z\_]+/', ['8:30', '9:00'], $string); // 活动将在 8:30 至 9:00 之间进行

//

//// Str::after

Str::after 函数返回在字符串中指定值之后的所有内容。

use Illuminate\Support\Str;

$slice = Str::after('This is my name', 'This is'); // ' my name'

//

//// Str::before

Str::before 函数返回在字符串中指定值之前的所有内容。

use Illuminate\Support\Str;

$slice = Str::before('This is my name', 'my name'); // 'This is '

//

//// Str::camel

Str::camel 函数将给定字符串「蛇式」转化为 camelCase「驼峰式」。

use Illuminate\Support\Str;

$converted = Str::camel('foo\_bar'); // fooBar

//

//// Str::contains

Str::contains 函数判断给定的字符串是否包含给定的值（区分大小写）。

use Illuminate\Support\Str;

$contains = Str::contains('This is my name', 'my'); // true

//

你也可以传递一个数组形式的值来判断字符串中是否包含任何值。

use Illuminate\Support\Str;

$contains = Str::contains('This is my name', ['my', 'foo']); // true

//

//// Str::containsAll

Str::containsAll 函数判断给定的字符串中是否包含给定的数组值。

use Illuminate\Support\Str;

$containsAll = Str::containsAll('This is my name', ['my', 'name']); // true

//

//// Str::endsWith

Str::endsWith 函数判断指定的字符串是否以给定的值结尾。

use Illuminate\Support\Str;

$result = Str::endsWith('This is my name', 'name'); // true

//

//// Str::finish

Str::finish 函数将给定的字符串以给定的值结尾返回（如果它尚未以给定值结尾）。

use Illuminate\Support\Str;

$adjusted = Str::finish('this/string', '/'); // this/string/

$adjusted = Str::finish('this/string/', '/'); // this/string/

//

//// Str::is

Str::is 函数判断给定的字符串是否匹配给定的模式。星号 \* 可以用来表示通配符。

use Illuminate\Support\Str;

$matches = Str::is('foo\*', 'foobar'); // true

$matches = Str::is('baz\*', 'foobar'); // false

//

//// Str::kebab

Str::kebab 函数将给定的「驼峰式」字符串转化为 kebab-case「短横式」字符串。

use Illuminate\Support\Str;

$converted = Str::kebab('fooBar'); // foo-bar

//

//// Str::limit

Str::limit 函数按给定的长度截断给定的字符串。

use Illuminate\Support\Str;

$truncated = Str::limit('The quick brown fox jumps over the lazy dog', 20); // The quick brown fox...

//

你也可以传递第三个参数来改变将被追加到最后的字符串。

use Illuminate\Support\Str;

$truncated = Str::limit('The quick brown fox jumps over the lazy dog', 20, ' (...)'); // The quick brown fox (...)

//

//// Str::orderedUuid

Str::orderedUuid 方法高效生成一个可存储在索引数据库列中的「第一时间」 UUID。

use Illuminate\Support\Str;

return (string) Str::orderedUuid();

//

//// Str::plural() 变成复数

Str::plural 函数将字符串转换为复数形式。该函数目前仅支持英文。

use Illuminate\Support\Str;

$plural = Str::plural('car'); // cars

$plural = Str::plural('child'); // children

//

你可以提供一个整数作为函数的第二个参数来检索字符串的单数或复数形式。

use Illuminate\Support\Str;

$plural = Str::plural('child', 2); // children

$plural = Str::plural('child', 1); // child

//

//// Str::random

Str::random 函数生成一个指定长度的随机字符串。这个函数用 PHP 的 random\_bytes 函数。

use Illuminate\Support\Str;

$random = Str::random(40);

//

//// Str::replaceArray

Str::replaceArray 函数使用数组顺序替换字符串中的给定值。

use Illuminate\Support\Str;

$string = 'The event will take place between ? and ?';

$replaced = Str::replaceArray('?', ['8:30', '9:00'], $string); // 活动将在 8:30 至 9:00 之间进行

//

//// Str::replaceFirst

Str::replaceFirst 函数替换字符串中给定值的第一个匹配项。

use Illuminate\Support\Str;

$replaced = Str::replaceFirst('the', 'a', 'the quick brown fox jumps over the lazy dog'); // a quick brown fox jumps over the lazy dog

//

//// Str::replaceLast

Str::replaceLast 函数替换字符串中最后一次出现的给定值。

use Illuminate\Support\Str;

$replaced = Str::replaceLast('the', 'a', 'the quick brown fox jumps over the lazy dog'); // the quick brown fox jumps over a lazy dog

//

//// Str::singular

Str::singular 函数将字符串转换为单数形式。该函数目前仅支持英文。

use Illuminate\Support\Str;

$singular = Str::singular('cars'); // car

$singular = Str::singular('children'); // child

//

//// Str::slug

The Str::slug 函数将给定的字符串生成一个 URL 友好的「slug」。

use Illuminate\Support\Str;

$slug = Str::slug('Laravel 5 Framework', '-'); // laravel-5-framework

//

//// Str::snake

Str::snake 函数将给定的字符串转换为 snake\_case「蛇式」。

use Illuminate\Support\Str;

$converted = Str::snake('fooBar'); // foo\_bar

//

//// Str::start

Str::start 函数将给定值添加到给定字符串的开始位置（如果字符串尚未以给定值开始）。

use Illuminate\Support\Str;

$adjusted = Str::start('this/string', '/'); // /this/string

$adjusted = Str::start('/this/string', '/'); // /this/string

//

//// Str::startsWith

Str::startsWith 函数判断给定的字符串的开头是否是指定值。

use Illuminate\Support\Str;

$result = Str::startsWith('This is my name', 'This'); // true

//

//// Str::studly

Str::studly 函数将给定的字符串转换为「变种驼峰命名」。

use Illuminate\Support\Str;

$converted = Str::studly('foo\_bar'); // FooBar

//

//// Str::title

Str::title 函数将给定的字符串转换为「首字母大写」。

use Illuminate\Support\Str;

$converted = Str::title('a nice title uses the correct case'); // A Nice Title Uses The Correct Case

//

//// Str::uuid

Str::uuid 方法生成一个 UUID（版本 4）。

use Illuminate\Support\Str;

return (string) Str::uuid();

//

//// Str::words

Str::words 函数限制字符串中的单词数。

use Illuminate\Support\Str;

return Str::words('Perfectly balanced, as all things should be.', 3, ' >>>'); // Perfectly balanced, as >>>

//

//// trans

trans 函数使用你的 本地文件转换给定的翻译密钥。

echo trans('messages.welcome');

如果指定的翻译键不存在，则 trans 方法会简单地返回给定的键。所以，就上面的例子而言，如果翻译键不存在， trans 方法会返回 messages.welcome。

//

//// trans\_choice

trans\_choice 函数根据词形变化来翻译给定的翻译键。

echo trans\_choice('messages.notifications', $unreadCount);

如果指定的翻译键不存在， trans\_choice 方法会简单地返回给定的键。所以，按照上面的例子，如果翻译键不存在，trans\_choice 方法会返回 messages.notifications。

(4)、URLs

//// action

action 函数为指定的控制器动作生成一个 URL 。你不需要传递完整的控制器命名空间。只需要传递相对于 App\Http\Controllers 命名空间控制器类名称。

$url = action('HomeController@index');

$url = action([HomeController::class, 'index']);

//

如果该方法接受路由参数，则可以将它们作为方法的第二个参数传递。

$url = action('UserController@profile', ['id' => 1]);

//

//// asset

asset 函数使用当前请求的协议（ HTTP 或 HTTPS ）为资源文件生成 URL。

$url = asset('img/photo.jpg');

您可以通过 ASSET\_URL 在 .env 文件中设置变量来配置资产 URL 主机。如果您在 Amazon S3 等外部服务上托管资产，这将非常有用。

// ASSET\_URL=http://example.com/assets

$url = asset('img/photo.jpg'); // http://example.com/assets/img/photo.jpg

//

//// route

route 函数为给定的命名路由生成一个 URL。

$url = route('routeName');

如果路由接受参数，则可以将它们作为方法的第二个参数传递。

$url = route('routeName', ['id' => 1]);

默认情况下， route 函数生成的是绝对 URL 。如果你想生成一个相对 URL ，你可以传递 false 作为第三个参数。

$url = route('routeName', ['id' => 1], false);

//

//// secure\_asset

secure\_asset 函数使用 HTTPS 协议为资源文件生成 URL。

$url = secure\_asset('img/photo.jpg');

//

//// secure\_url

secure\_url 函数为给定的路径生成一个标准的 HTTPS URL。

$url = secure\_url('user/profile');

$url = secure\_url('user/profile', [1]);

//

//// url

url 函数生成给定路径的标准 URL。

$url = url('user/profile');

$url = url('user/profile', [1]);

//

如果没有提供路径，则返回 Illuminate\Routing\UrlGenerator 实例。

$current = url()->current();

$full = url()->full();

$previous = url()->previous();

(5)、其他

//// abort

abort 函数抛出 异常处理程序( https://learnku.com/docs/laravel/6.x/errors#the-exception-handler ) 所呈现的 HTTP 异常( https://learnku.com/docs/laravel/6.x/errors#http-exceptions )。

abort(403);

//

你也可以提供异常响应文本和自定义响应头。

abort(403, 'Unauthorized.', $headers);

//

//// abort\_if

当给定的布尔表达式的值为 true 时，abort\_if 函数抛出一个 HTTP 异常。

abort\_if(! Auth::user()->isAdmin(), 403);

和 abort 方法一样，你也可以提供异常响应文本作为第三个参数，并提供一个自定义响应头数据作为第四个参数。

//

//// abort\_unless

当给定的表达式的值为 false 时，abort\_unless 函数抛出一个 HTTP 异常。

abort\_unless(Auth::user()->isAdmin(), 403);

和 abort 方法一样，你也可以提供异常的响应文本作为第三个参数，并提供一个自定义响应头数组作为第四个参数。

//

//// app

app 函数返回 服务容器( https://learnku.com/docs/laravel/6.x/container ) 实例。

$container = app();

你可以传递一个类或接口名称来从容器中解析它。

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

//

//// auth

auth 函数返回一个 认证( https://learnku.com/docs/laravel/6.x/authentication ) 实例。你可以使用它来替代 Auth facade。

$user = auth()->user();

如果需要，你可以指定你想要访问的认证实例。

$user = auth('admin')->user();

//

//// back

back 函数生成一个 重定向 HTTP 响应( https://learnku.com/docs/laravel/6.x/responses#redirects ) 到用户之前的位置。

return back($status = 302, $headers = [], $fallback = false);

return back();

//

//// bcrypt

bcrypt 函数 哈希( https://learnku.com/docs/laravel/6.x/hashing ) 使用 Bcrypt 对给定的值进行散列。你可以使用它替代 Hash facade。

$password = bcrypt('my-secret-password');

//

//// blank

blank 函数判断给定的值是否为空。

// true

blank('');

blank(' ');

blank(null);

blank(collect());

//

// false

blank(0);

blank(true);

blank(false);

如果想使用与 blank 函数相反的方法，请看 filled ( https://learnku.com/docs/laravel/5.8/helpers/3919#method-filled ) 方法。

//

//// broadcast

broadcast 函数将 广播 ( https://learnku.com/docs/laravel/6.x/broadcasting ) 给定的 事件 ( https://learnku.com/docs/laravel/6.x/events ) 到它的监听器。

broadcast(new UserRegistered($user));

//

//// cache

cache 函数可以从 缓存 ( https://learnku.com/docs/laravel/6.x/cache ) 中获取值。如果缓存中给定的键不存在，将返回一个可选的默认值。

$value = cache('key');

$value = cache('key', 'default');

你可以通过向函数添加键值对数组来设置缓存项。与此同时，你还应该传递有效的分钟数或者缓存的持续时间来设置缓存过期时间。

cache(['key' => 'value'], 300);

cache(['key' => 'value'], now()->addSeconds(10));

//

//// class\_uses\_recursive

class\_uses\_recursive 函数返回一个类使用的所有 traits , 包括它所有父类使用的 traits。

$traits = class\_uses\_recursive(App\User::class);

//

//// collect

collect 函数根据给定的值创建一个 collection ( https://learnku.com/docs/laravel/6.x/collections ) 实例。

$collection = collect(['taylor', 'abigail']);

//

//// config

config 函数获取 configuration ( https://learnku.com/docs/laravel/6.x/configuration ) 变量的值。可以使用「点」语法访问配置的值，其中包括文件的名称和访问的选项，如果访问的配置选项不存在，你可以指定一个默认值并且返回这个默认值。

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

$value = config('app.timezone', $default);

//

你也可以在运行时通过传递一个键／值对数组来设置配置变量。

config(['app.debug' => true]);

//

//// cookie

cookie 函数创建一个新的 cookie ( https://learnku.com/docs/laravel/6.x/requests#cookies ) 实例。

$cookie = cookie('name', 'value', $minutes);

//

//// csrf\_field

csrf\_field 函数生成一个包含 CSRF 令牌值的 HTML 输入表单字段 hidden。例如，使用 Blade 语法 ( https://learnku.com/docs/laravel/6.x/blade )。

{{ csrf\_field() }}

//

//// csrf\_token

csrf\_token 函数获取当前 CSRF 令牌的值。

$token = csrf\_token();

//

//// dd

dd 函数打印输出给定的变量并且结束脚本运行。

dd($value);

dd($value1, $value2, $value3, ...);

如果你不停止执行脚本，那么可以使用 dump 函数。

//

//// decrypt

decrypt 函数可以使用 Laravel 的 加密解密机制( https://learnku.com/docs/laravel/6.x/encryption )。

$decrypted = decrypt($encrypted\_value);

//

//// dispatch

dispatch 函数将给定的 任务 ( https://learnku.com/docs/laravel/6.x/queues#creating-jobs ) 推送到 Laravel 任务队列( https://learnku.com/docs/laravel/6.x/queues )。

dispatch(new App\Jobs\SendEmails);

//

//// dispatch\_now

dispatch\_now 函数立即运行给定的 任务 ( https://learnku.com/docs/laravel/6.x/queues#creating-jobs ) 并从 handle 方法返回值。

$result = dispatch\_now(new App\Jobs\SendEmails);

//

//// dump

dump 打印给定的变量。

dump($value);

dump($value1, $value2, $value3, ...);

如果你想要在打印后停止执行脚本，可以使用 dd 函数。

{提示} 你可以使用 Artisan 中的 dump-server 命令拦截所有的 dump 调用并把它们显示在控制台窗口而不是浏览器中。

//

//// encrypt

encrypt 函数使用 Laravel 的 加解密机制( https://learnku.com/docs/laravel/6.x/encryption ) 对给定的值进行加密。

$encrypted = encrypt($unencrypted\_value);

//

//// env

env 函数可以获取 环境变量 ( https://learnku.com/docs/laravel/6.x/configuration#environment-configuration ) 配置的值或者返回默认值。

$env = env('APP\_ENV');

$env = env('APP\_ENV', 'production'); // 返回 'production' 如果 APP\_ENV 未设置的话...

{注意} 如果你在部署过程中执行了 config:cache 命令 ，那么你应该确保只从配置文件中调用 env 函数。一旦配置被缓存，.env 文件将不再次加载，所有对 env 函数的调用将返回 null。

//

//// event

event 函数向监听器派发给定 事件( https://learnku.com/docs/laravel/6.x/events )。

event(new UserRegistered($user));

//

//// factory

factory 函数根据给定的类、名称和数量创建模型工厂构造器。它能够被用于 测试 ( https://learnku.com/docs/laravel/6.x/database-testing#writing-factories ) 或 数据填充( https://learnku.com/docs/laravel/6.x/seeding#using-model-factories )。

$user = factory(App\User::class)->make();

//

//// filled

filled 函数返回是否不为「空」。

// true

filled(0);

filled(true);

filled(false);

//

// false

filled('');

filled(' ');

filled(null);

filled(collect());

对于作用与 filled 相反的方法，可以查看 blank 方法。

//

//// info

info 函数将信息写入 log ( https://learnku.com/docs/laravel/6.x/logging )。

info('Some helpful information!');

可以将上下文数据数组传递给此函数。

info('User login attempt failed.', ['id' => $user->id]);

//

//// logger

logger 函数可以被用于将 debug 级别的消息写入 log ( https://learnku.com/docs/laravel/6.x/logging )。

logger('Debug message');

可以将上下文数据数组传递给此函数。

logger('User has logged in.', ['id' => $user->id]);

如果不带参数调用此函数，它将返回 logger ( https://learnku.com/docs/laravel/6.x/errors#logging ) 实例。

logger()->error('You are not allowed here.');

//

//// method\_field

method\_field 函数生成包含模仿表单 HTTP 动作的 HTML hidden 域。下面的例子使用了 Blade 语法( https://learnku.com/docs/laravel/6.x/blade )。

<form method="POST">

{{ method\_field('DELETE') }}

</form>

//

//// now

now 函数为当前时间创建一个新的 Illuminate\Support\Carbon 实例。

$now = now();

//

//// old

old 函数 获取 ( https://learnku.com/docs/laravel/6.x/requests#retrieving-input ) 刷入 session 的 旧的输入值( https://learnku.com/docs/laravel/6.x/requests#old-input )。

$value = old('value');

$value = old('value', 'default');

//

//// optional

optional 函数接受任何参数，并允许你访问该对象上的属性或调用其方法。如果给定对象为 null，属性或方法将返回 null 而不是引发错误。

return optional($user->address)->street;

{!! old('name', optional($user)->name) !!}

optional 函数也接受闭包作为其第二个参数。如果第一个参数提供的值不是 null，闭包将被调用。

return optional(User::find($id), function ($user) {

return new DummyUser;

});

//

//// policy

policy 方法为给定的类获取 policy ( https://learnku.com/docs/laravel/6.x/authorization#creating-policies ) 实例。

$policy = policy(App\User::class);

//

//// redirect

redirect 函数返回 重定向 HTTP 响应 ( https://learnku.com/docs/laravel/6.x/responses#redirects )，如果不带参数调用则返回重定向器实例。

return redirect($to = null, $status = 302, $headers = [], $secure = null);

return redirect('/home');

return redirect()->route('route.name');

//

//// report

report 函数使用 异常处理器 ( https://learnku.com/docs/laravel/6.x/errors#the-exception-handler ) 的 report 方法报告异常。

report($e);

//

//// request

request 函数返回当前 请求 ( https://learnku.com/docs/laravel/6.x/requests ) 实例，或者获取一个输入项。

$request = request();

$value = request('key', $default);

//

//// rescue

rescue 函数执行给定的闭包，并且捕获其执行过程中引发的任何异常。捕获的所有异常都将传递给 异常处理器 ( https://learnku.com/docs/laravel/6.x/errors#the-exception-handler ) 的 report 方法；然后继续处理此次请求。

return rescue(function () {

return $this->method();

});

还可以为其传递第二个参数。这个参数将作为执行闭包引发异常时的 「默认」值。

return rescue(function () {

return $this->method();

}, false);

return rescue(function () {

return $this->method();

}, function () {

return $this->failure();

});

//

//// resolve

resolve 函数使用 服务容器 ( https://learnku.com/docs/laravel/6.x/container ) 解析给定名称的类或接口的实例。

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

//

//// response

response 函数创建 响应 ( https://learnku.com/docs/laravel/6.x/responses ) 实例，或者获得响应工厂的实例。

return response('Hello World', 200, $headers);

return response()->json(['foo' => 'bar'], 200, $headers);

//

//// retry

retry 函数尝试执行给定的回调，直到达到给定的最大尝试阈值。如果回调没有抛出异常，回调返回值将被返回。如果回调抛出异常，将自动重试。达到最大尝试次数，将抛出异常。

return retry(5, function () {

// 在尝试之间休息100毫秒时尝试5次...

}, 100);

//

//// session

session 函数用于获取或设置 session ( https://learnku.com/docs/laravel/6.x/session ) 值。

$value = session('key');

可以向该函数传递键值对数组来设置 session 值。

session(['chairs' => 7, 'instruments' => 3]);

不带参数调用此函数，则返回存储在 session 中的值。

$value = session()->get('key');

session()->put('key', $value);

//

//// tap

tap 函数接受两个参数： 任意 $value 和闭包。 $value 将被传递给闭包，并被 tap 函数返回。与闭包的返回值无关。

$user = tap(User::first(), function ($user) {

$user->name = 'taylor';

$user->save();

});

如果没有向 tap 函数传递闭包，可以调用给定 $value 的任意方法。调用此方法的返回值永远是 $value，无论方法在其定义中返回什么。例如，Eloquent update 方法指定返回一个整数。但是，我们可以通过 tap 函数链式调用 update 方法强制其返回模型自身。

$user = tap($user)->update([

'name' => $name,

'email' => $email,

]);

要向类中添加 tap 方法，可以将 Illuminate\Support\Traits\Tappable 特征添加到类中。 此特征的 tap 方法接受闭包作为其唯一参数。 对象实例本身将传递给闭包，然后由 tap 方法返回。

return $user->tap(function ($user) {

//

});

//

//// throw\_if

在给定的布尔表达式结果为 true 时，throw\_if 函数抛出给定的异常。

throw\_if(! Auth::user()->isAdmin(), AuthorizationException::class);

throw\_if(

! Auth::user()->isAdmin(),

AuthorizationException::class,

'You are not allowed to access this page'

);

//

//// throw\_unless

在给定的布尔表达式结果为 false 时，throw\_unless 函数抛出给定的异常。

throw\_unless(Auth::user()->isAdmin(), AuthorizationException::class);

throw\_unless(

Auth::user()->isAdmin(),

AuthorizationException::class,

'You are not allowed to access this page'

);

//

//// today

today 函数根据当前日期创建新的 Illuminate\Support\Carbon 实例。

$today = today();

//

//// trait\_uses\_recursive

trait\_uses\_recursive 返回被 trait 使用的全部 trait。

$traits = trait\_uses\_recursive(\Illuminate\Notifications\Notifiable::class);

//

//// transform

transform 函数执行基于（非 空 ( https://learnku.com/docs/laravel/6.x/helpers/5164#method-blank ) ）给定值的 闭包，并返回 闭包 的结果。

$callback = function ($value) {

return $value \* 2;

};

$result = transform(5, $callback); // 10

还可以传递一个默认值或 闭包 作为该函数的第三个参数。如果给定的值为空时，返回该值。

$result = transform(null, $callback, 'The value is blank'); // 此值为空

//

//// validator

validator 函数根据指定的参数创建一个新的 验证器 ( https://learnku.com/docs/laravel/6.x/validation ) 实例。方便起见可以用它来代替 Validator facade。

$validator = validator($data, $rules, $messages);

//

//// value

value 函数返回给定值。如果传递 闭包 给此函数，将执行 闭包 并返回闭包调用的结果。

$result = value(true); // 真

$result = value(function () {

return false;

}); // 假

//

//// view

view 函数获取一个 view ( https://learnku.com/docs/laravel/6.x/views ) 实例。

return view('auth.login');

//

//// with

with 函数返回给定的值。如果传递了一个 Closure 给第二个参数，那么会返回 Closure 执行的结果。

$callback = function ($value) {

return (is\_numeric($value)) ? $value \* 2 : 0;

};

$result = with(5, $callback); // 10

$result = with(null, $callback); // 0

$result = with(5, null); // 5

--

--

八、邮件发送 (Mail)

--

https://learnku.com/docs/laravel/6.x/mail/5165

--

1、简介

Laravel 基于 SwiftMailer ( https://swiftmailer.symfony.com/ ) 库提供了一套干净、清爽的邮件 API。Laravel 为 SMTP、Mailgun、Postmark、Amazon SES，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。

(1)、邮件驱动预备知识

基于 API 的驱动程序（如 Mailgun 和 Postmark）通常比 SMTP 服务器更简单，更快捷。 如果可以的话，尽可能使用其中一个驱动程序。 所有 API 驱动程序都需要 Guzzle HTTP 库，可以通过 Composer 包管理器安装。

// 安装GuzzleHTTP

composer require guzzlehttp/guzzle

//

///1/ Mailgun 驱动

要使用 Mailgun 驱动，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',

],

如果您没有使用 “US” Mailgun ( https://documentation.mailgun.com/en/latest/api-intro.html#mailgun-regions ) 地区 ，您可以在 services 配置文件中定义您所在地区的终端。

'mailgun' => [

'domain' => 'your-mailgun-domain',

'secret' => 'your-mailgun-key',

'endpoint' => 'api.eu.mailgun.net',

],

//

///2/ Postmark 驱动程序

要使用 Postmark 驱动程序，请通过 Composer 安装 Postmark 的 SwiftMailer。

composer require wildbit/swiftmailer-postmark

接下来，安装 Guzzle 并将 config/mail.php 配置文件中的 driver 选项设置为 postmark。 最后，验证您的 config/services.php 配置文件包含以下选项。

'postmark' => [

'token' => 'your-postmark-token',

],

//

///3/ SES 驱动

要使用 Amazon SES 驱动，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

],

如果你在执行 SES 时需要包含 附加选项 ( https://docs.aws.amazon.com/aws-sdk-php/v3/api/api-email-2010-12-01.html#sendrawemail ) SendRawEmail 请求，可以在 ses 配置中定义 options 数组。

'ses' => [

'key' => 'your-ses-key',

'secret' => 'your-ses-secret',

'region' => 'ses-region', // e.g. us-east-1

'options' => [

'ConfigurationSetName' => 'MyConfigurationSet',

'Tags' => [

[

'Name' => 'foo',

'Value' => 'bar',

],

],

],

],

--

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。

(1)、配置发件人

///1/ 使用 from 方法

我们来看一下邮件发件人的配置，或者，换句话说，邮件来自于谁。有两种方式来配置发送者，第一种方式是在可邮寄类的 build 方法方法中调用 from 方法。

/\*\*

\* 构建消息

\*

\* @return $this

\*/

public function build()

{

return $this->from('example@example.com')

->view('emails.orders.shipped');

}

//

///2/ 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， 如果在 mailable 类中未指定其他 from 地址，则将使用此地址。

'from' => ['address' => 'example@example.com', 'name' => 'App Name'],

此外，您可以在 config/mail.php 配置文件中定义全局 reply\_to 地址。

'reply\_to' => ['address' => 'example@example.com', 'name' => 'App Name'],

(2)、配置视图

你可以在可邮寄类的 build 方法中使用 view 方法来指定渲染邮件内容时使用哪个视图模板，由于每封邮件通常使用 Blade 模板 ( https://learnku.com/docs/laravel/6.x/blade ) 来渲染内容，所以你可以在构建邮件 HTML 时使用 Blade 模板引擎提供的所有功能。

/\*\*

\* 构建消息

\*

\* @return $this

\*/

public function build()

{

return $this->view('emails.orders.shipped');

}

Tip：你可以创建一个 resources/views/emails 目录来存放所有邮件模板，当然，你也可以将邮件模板放到 resources/views 目录下任意其它位置。

//

///1/ 纯文本邮件

如果你想要定义一个纯文本格式的邮件，可以使用 text 方法。和 view 方法一样， text 方法接收一个用于渲染邮件内容的模板名，你既可以定义纯文本消息也可以定义 HTML 消息。

/\*\*

\* 构建消息

\*

\* @return $this

\*/

public function build()

{

return $this->view('emails.orders.shipped')

->text('emails.orders.shipped\_plain');

}

(3)、视图数据

///1/ 通过公共属性

通常，我们需要传递一些数据到渲染邮件的 HTML 视图以供使用。有两种方式将数据传递到视图，首先，您的 mailable 类中定义的任何公共属性将自动传递给视图。 因此，您可以将数据传递到 mailable 类的构造函数，并将该数据设置为类的公共属性。

<?php

namespace App\Mail;

use App\Order;

use Illuminate\Bus\Queueable;

use Illuminate\Mail\Mailable;

use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable

{

use Queueable, SerializesModels;

/\*\*

\* 订单实例.

\*

\* @var Order

\*/

public $order;

/\*\*

\* 创建一个新的消息实例.

\*

\* @return void

\*/

public function \_\_construct(Order $order)

{

$this->order = $order;

}

/\*\*

\* 构建消息.

\*

\* @return $this

\*/

public function build()

{

return $this->view('emails.orders.shipped');

}

}

?>

数据被设置给公共属性后，将会在视图中自动生效，所以你可以像在 Blade 模板中访问其它数据一样访问它们。

<div>

Price: {{ $order->price }}

</div>

//

///2/ 通过 with 方法

如果你想要在数据发送到模板之前自定义邮件数据的格式，可以通过 with 方法手动传递数据到视图。一般情况下，你还是需要通过可邮寄类的构造器传递数据，不过，这次你需要设置数据为 protected 或 private 属性，这样，这些数据就不会在视图中自动生效。然后，当调用 with 方法时，传递数组数据到该方法以便数据在视图模板中生效。

<?php

namespace App\Mail;

use App\Order;

use Illuminate\Bus\Queueable;

use Illuminate\Mail\Mailable;

use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable

{

use Queueable, SerializesModels;

/\*\*

\* 订单实例.

\*

\* @var Order

\*/

protected $order;

/\*\*

\* 创建一个新的消息实例.

\*

\* @return void

\*/

public function \_\_construct(Order $order)

{

$this->order = $order;

}

/\*\*

\* 构建消息.

\*

\* @return $this

\*/

public function build()

{

return $this->view('emails.orders.shipped')

->with([

'orderName' => $this->order->name,

'orderPrice' => $this->order->price,

]);

}

}

?>

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>

(4)、附件

要在邮件中加入附件，在 build 方法中使用 attach 方法。attach 方法接受文件的绝对路径作为它的第一个参数。

/\*\*

\* 构建消息

\*

\* @return $this

\*/

public function build()

{

return $this->view('emails.orders.shipped')

->attach('/path/to/file');

}

附加文件到消息时，你也可以传递 数组 给 attach 方法作为第二个参数，以指定显示名称和 / 或是 MIME 类型。

/\*\*

\* 构建消息

\*

\* @return $this

\*/

public function build()

{

return $this->view('emails.orders.shipped')

->attach('/path/to/file', [

'as' => 'name.pdf',

'mime' => 'application/pdf',

]);

}

//

///1/ 从磁盘中添加附件

如果您已在文件存储( https://learnku.com/docs/laravel/6.x/filesystem )上存储了一个文件，则可以使用 attachFromStorage 方法将其附加到电子邮件中。

/\*\*

\* 构建消息

\*

\* @return $this

\*/

public function build()

{

return $this->view('email.orders.shipped')

->attachFromStorage('/path/to/file');

}

如有必要，您可以使用 attachFromStorage 方法的第二个和第三个参数指定文件的附件名称和其他选项。

/\*\*

\* 构建消息

\*

\* @return $this

\*/

public function build()

{

return $this->view('email.orders.shipped')

->attachFromStorage('/path/to/file', 'name.pdf', [

'mime' => 'application/pdf'

]);

}

如果需要指定默认磁盘以外的存储磁盘，可以使用 attachFromStorageDisk 方法。

/\*\*

\* 构建消息

\*

\* @return $this

\*/

public function build()

{

return $this->view('email.orders.shipped')

->attachFromStorageDisk('s3', '/path/to/file');

}

//

///2/ 原始数据附件

attachData 可以使用字节数据作为附件。例如，你可以使用这个方法将内存中生成而没有保存到磁盘中的 PDF 附加到邮件中。attachData 方法第一个参数接收原始字节数据，第二个参数为文件名，第三个参数接受一个数组以指定其他参数。

/\*\*

\* 构建消息

\*

\* @return $this

\*/

public function build()

{

return $this->view('emails.orders.shipped')

->attachData($this->pdf, 'name.pdf', [

'mime' => 'application/pdf',

]);

}

(5)、内联附件

在邮件中嵌入内联图片通常都很麻烦；不过，Laravel 提供了向邮件中附加图片并获取适当的 CID 的简便方法。可以使用邮件模板中 $message 变量的 embed 方法来嵌入内联图片。Laravel 自动使 $message 变量在全部邮件模板中可用，不需要担心如何手动传递它。

<body>

Here is an image:

<img src="{{ $message->embed($pathToImage) }}">

</body>

注意：$message 在文本消息中不可用，因为文本消息不能使用内联附件。

//

///1/ 嵌入原始数据附件

如果已经有了希望嵌入邮件模板的原始数据串，可以使用 $message 变量的 embedData 方法。

<body>

Here is an image from raw data:

<img src="{{ $message->embedData($data, $name) }}">

</body>

(6)、自定义 SwiftMailer 消息

Mailable 基类的 withSwiftMessage 方法允许你注册一个回调，它将在发送消息之前被调用，原始的 SwiftMailer 消息将作为该回调的参数。

/\*\*

\* 构建消息

\*

\* @return $this

\*/

public function build()

{

$this->view('emails.orders.shipped');

$this->withSwiftMessage(function ($message) {

$message->getHeaders()

->addTextHeader('Custom-Header', 'HeaderValue');

});

}

--

4、Markdown 邮件类 (Markdown 格式的 Mailables 类)

Markdown 格式 mailable 消息允许你从预构建模板和 mailable 类中的邮件通知组件获益。由于消息使用 Markdown 书写，Laravel 能够渲染出美观的、响应式的 HTML 模板消息，还能自动生成文本副本。

(1)、生成 Markdown 邮件类 ( 生成 Markdown 格式的 Mailables 类 )

要生成一个适用 Markdown 模板的 mailable，可以使用带有 --markdown 选项的 make:mail Artisan 命令。

// 生成Markdown模板的mailable

php artisan make:mail OrderShipped --markdown=emails.orders.shipped

然后，在它的 build 方法中配置 mailable，调用 markdown 方法代替 view 方法。 markdown 方法接受 Markdown 模板名和一个可选的在模板中可用的数组。

/\*\*

\* 构建消息

\*

\* @return $this

\*/

public function build()

{

return $this->from('example@example.com')

->markdown('emails.orders.shipped');

}

(2)、编写 Markdown 消息 (编写 Markdown 消息)

Markdown mailable 使用 Blade 组件和 Markdown 语法组合，让你可以更方便地利用 Laravel 预制组件构建邮件消息。

~~~ s ~~~

@component('mail::message')

# 订单已发货

Your order has been shipped!

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

View Order

@endcomponent

Thanks,<br>

{{ config('app.name') }}

@endcomponent

~~~ e ~~~

Tip：编写 Markdown 邮件时不要使用额外的缩进。Markdown 解析器将把缩进内容渲染成代码块。

//

///1/ 按钮组件

按钮组件渲染一个居中按钮链接。此组件接受两个参数， url 和可选的 color。颜色选项支持 primary、 success 和 error。你可以随心所欲地向消息添加任意数量的按钮组件。

@component('mail::button', ['url' => $url, 'color' => 'success'])

View Order

@endcomponent

//

///2/ 面板组件

面板组件在面板内渲染给定的文字块，面板与其他消息的背景色略有不同。能让你绘制一个警示文字块。

@component('mail::panel')

This is the panel content.

@endcomponent

//

///3/ 表格组件

表格组件允许你将 Markdown 表格转换成 HTML 表格。此组件接受 Markdown 表格作为其内容。列对齐支持默认的 Markdown 表格对齐语法。

@component('mail::table')

| Laravel | Table | Example |

| ------------- |:-------------:| --------:|

| Col 2 is | Centered | $10 |

| Col 3 is | Right-Aligned | $20 |

@endcomponent

(3)、自定义组件

可以将所有 Markdown 邮件组件导出到自己的应用，用作自定义组件的模板。若要导出这些组件，使用带有 laravel-mail 资产标签的 vendor:publish Artisan 命令。

// 导出laravel-mail资产

php artisan vendor:publish --tag=laravel-mail

此命令将 Markdown 邮件组件导出到 resources/views/vendor/mail 目录。 mail 目录包含 html 和 text 子目录，分别包含各自对应的可用组件描述。可以按照自己的意愿自定义这些组件。

//

///1/ 自定义 CSS

组件导出以后，resources/views/vendor/mail/html/themes 目录有一个 default.css 文件。可以自此文件中自定义 CSS，这些样式将自动内联到 Markdown 邮件消息的 HTML 表示中。

如果想为 Laravel 的 Markdown 组件构建一个全新的主题，您可以在 html/themes 目录中新建一个 CSS 文件。 命名并保存 CSS 文件后，并更新 mail 配置文件的 theme 选项以匹配新主题的名称。

要为单个 mailable 自定义主题，可以将 mailable 类的 $theme 属性设置为发送 mailable 时应使用的主题名称。

--

5、发送邮件

若要发送邮件，使用 Mail facade ( https://learnku.com/docs/laravel/6.x/facades ) 的 to 方法。 to 方法接受 邮件地址、用户实例或用户集合。如果传递一个对象或者对象集合，mailer 在设置收件人时将自动使用它们的 email 和 name 属性，因此请确保对象的这些属性可用。一旦指定了收件人，就可以将 mailable 类实例传递给 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

{

/\*\*

\* 发送给定的订单。

\*

\* @param Request $request

\* @param int $orderId

\* @return Response

\*/

public function ship(Request $request, $orderId)

{

$order = Order::findOrFail($orderId);

// 发送订单...

Mail::to($request->user())->send(new OrderShipped($order));

}

}

?>

在发送消息时不止可以指定收件人。还可以通过链式调用「to」、「cc」、「bcc」一次性指定抄送和密送收件人。

Mail::to($request->user())

->cc($moreUsers)

->bcc($evenMoreUsers)

->send(new OrderShipped($order));

--

6、渲染可邮寄类 (渲染 Mailable)

有时可能希望捕获 mailable 的 HTML 内容，而不发送它。可以调用 mailable 的 render 方法实现此目的。此方法返回 mailable 渲染计算后的字符串。

$invoice = App\Invoice::find(1);

return (new App\Mail\InvoicePaid($invoice))->render();

//

(1)、在浏览器中预览 Mailable (在浏览器中预览邮件)

设计 mailable 模板时，像 Blade 模板一样在浏览器中预览和渲染 mailable 是很方便的。这种情况下，Laravel 允许你在路由闭包或控制其中直接返回任意 mailable。返回的 mailable 将在浏览器中渲染和显示，你可以快速预览设计效果，而不需要将其发送到真实的邮件地址。

Route::get('mailable', function () {

$invoice = App\Invoice::find(1);

return new App\Mail\InvoicePaid($invoice);

});

(2)、邮件队列

//

///1/ 将邮件消息加入队列

由于发送邮件消息可能大幅度延长应用的响应时间，许多开发者选择将邮件消息加入队列放在后台发送。Laravel 使用内置的 统一队列API ( https://learnku.com/docs/laravel/6.x/queues ) 简化了这一工作。若要将邮件消息加入队列，可以在指定消息的接收者后，使用 Mail facade 的 queue 方法。

Mail::to($request->user())

->cc($moreUsers)

->bcc($evenMoreUsers)

->queue(new OrderShipped($order));

此方法自动将作业推送到队列中以便消息在后台发送。使用此特性之前，需要 配置队列( https://learnku.com/docs/laravel/6.x/queues )。

//

///2/ 延迟消息队列

想要延迟发送队列化的邮件消息，可以使用 later 方法。later 方法的第一个参数的第一个参数是标示消息何时发送的 DateTime 实例。

$when = now()->addMinutes(10);

Mail::to($request->user())

->cc($moreUsers)

->bcc($evenMoreUsers)

->later($when, new OrderShipped($order));

//

///3/ 推送到指定队列

由于所有使用 make:mail 命令生成的 mailable 类都是用了 Illuminate\Bus\Queueable trait，因此你可以在任何 mailable 类实例上调用 onQueue 和 onConnection 方法来指定消息的连接和队列名。

$message = (new OrderShipped($order))

->onConnection('sqs')

->onQueue('emails');

Mail::to($request->user())

->cc($moreUsers)

->bcc($evenMoreUsers)

->queue($message);

//

///4/ 默认队列

如果一个 mailable 类终是要队列化，可以在此类上实现 ShouldQueue 契约。这样一来，即使你在发送时调用了 send 方法， mailable 也将被序列化。

use Illuminate\Contracts\Queue\ShouldQueue;

class OrderShipped extends Mailable implements ShouldQueue

{

//

}

--

7、本地化可邮寄类 (本地化 Mailable)

Laravel 允许你使用有别于当前语言的区域设置发送 mailable，即使被加入到队列中也保留该区域设置。

为达到此目的， Mail facade 提供了 locale 方法设置目标语言。应用在格式化 mailable 是将切换到该区域设置，并在格式化完成后恢复到原来的区域设置。

Mail::to($request->user())->locale('es')->send(

new OrderShipped($order)

);

(1)、用户首选区域设置

有时候，应用存储每个用户的首选区域设置。通过在一个或多个模型上实现 HasLocalePreference 契约，可以通知 Laravel 在发送邮件时使用预存的区域设置。

use Illuminate\Contracts\Translation\HasLocalePreference;

class User extends Model implements HasLocalePreference

{

/\*\*

\* 获取用户首选区域设置。

\*

\* @return string

\*/

public function preferredLocale()

{

return $this->locale;

}

}

一旦实现了此接口，Laravel 在向此模型发送 mailable 和通知时，将自动使用首选区域设置。因此在使用此接口时不需要调用 locale 方法。

Mail::to($request->user())->send(new OrderShipped($order));

--

8、邮件 & 本地开发

在开发发送邮件的应用时，你也许不想真的向实时邮件地址发送邮件。Laravel 为本地开发期间提供了几个 「禁用」真实发送的途径。

(1)、日志驱动

log 邮件驱动采取将邮件消息写入日志取代发送邮件，已备查看。应用环境配置的更多消息，请查阅 配置文档( https://learnku.com/docs/laravel/6.x/configuration#environment-configuration )。

(2)、通用配置

Laravel 为通过框架发送的邮件提供了指定常用收件人的其他解决方案。通过此方法，应用生成的邮件都将发送到指定地址，以取代发送消息时指定的真实地址。可以借助 config/mail.php 配置文件的 to 选项实现此目的。

'to' => [

'address' => 'example@example.com',

'name' => 'Example'

],

(3)、Mailtrap

最后，你可以使用 Mailtrap 服务和 smtp 驱动发送邮件消息到 「虚拟」邮箱，这样就可以在真实的邮件客户端查看邮件消息。此方法的好处是允许你在 Mailtrap 的消息阅览器中实际查看最终的邮件。

--

9、事件

Laravel 在处理邮件消息发送时触发两个事件。MessageSending 事件在消息发送前触发，MessageSent 事件则在消息发送后触发。切记，这些事件是在邮件被 发送 时触发，而不是在队列化的时候。可以在 EventServiceProvider 中注册此事件的侦听器。

/\*\*

\* 为应用映射事件侦听器。

\*

\* @var array

\*/

protected $listen = [

'Illuminate\Mail\Events\MessageSending' => [

'App\Listeners\LogSendingMessage',

],

'Illuminate\Mail\Events\MessageSent' => [

'App\Listeners\LogSentMessage',

],

];

--

--

九、消息通知

--

https://learnku.com/docs/laravel/6.x/notifications/5166

--

1、简介

除了支持 发送邮件 ( https://learnku.com/docs/laravel/6.x/mail ) 之外，Laravel 还支持通过多种频道发送通知，包括邮件、短信 (通过 Nexmo https://www.nexmo.com/ )，以及 Slack ( https://slack.com/ )。通知还能存储到数据库以便后续在 Web 页面中显示。

通常，通知都是简短、有信息量的消息，用于通知用户应用中发生了什么。举例来说，如果你是在编写一个在线交易的应用，你应该会通过邮件和短信频道类给用户发送一条 「账单支付」 的通知。

--

2、创建通知

Laravel 中的一条通知就是一个类 (通常存放在 app/Notifications 文件夹下)。看不到的话不要担心， 运行下 make:notification 命令就能创建了。

// 创建通知

php artisan make:notification InvoicePaid

这条命令会在 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 方法以及一个或多个消息构建的方法（比如 toMail 或者 toDatabase)，它们会针对指定的渠道把通知转换为对应的消息。

--

3、发送通知

(1)、使用 Notifiable Trait

通知可以通过两种方法发送: Notifiable trait 的 notify 方法或 Notification facade ( https://learnku.com/docs/laravel/6.x/facades )。首先，让我们来探讨下使用 trait。

<?php

namespace App;

use Illuminate\Notifications\Notifiable;

use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable

{

use Notifiable;

}

?>

默认的 App\User 模型中使用了这个 trait，它包含着一个可以用来发送通知的方法： notify。notify 方法需要一个通知实例做参数。

use App\Notifications\InvoicePaid;

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

提示：请记住，你可以在任意模型中使用 Illuminate\Notifications\Notifiable trait。而不仅仅是在 User 模型中。

(2)、使用 Notification Facade

另外，你可以通过 Notification facade ( https://learnku.com/docs/laravel/6.x/facades ) 来发送通知，它主要用在当你给多个可接收通知的实体发送的时候，比如给用户集合发送通知。使用 Facade 发送通知的话，要把可以接收通知和通知的实例传递给 send 方法。

// 发送通知

Notification::send($users, new InvoicePaid($invoice));

(3)、指定发送频道 (发送指定频道)

每个通知类都会有个 via 方法，它决定了通知会在哪个频道上发送。开箱即用的频道有 mail，database，broadcast，nexmo，和 slack。

提示：如果你想使用其他的频道，比如 Telegram 或者 Pusher，你可以去看下社区驱动的 Laravel 通知频道网站( http://laravel-notification-channels.com/ )。

via 方法接收一个 $notifiable 实例，这个实例将是通知实际发送到的类的实例。你可以用 $notifiable 来决定这个通知用哪些频道来发送。

/\*\*

\* 获取通知发送频道。

\*

\* @param mixed $notifiable

\* @return array

\*/

public function via($notifiable)

{

return $notifiable->prefers\_sms ? ['nexmo'] : ['mail', 'database'];

}

(4)、通知队列化

注：使用通知队列前需要配置队列并 开启一个队列任务( https://learnku.com/docs/laravel/6.x/queues )。

发送通知可能是耗时的，尤其是通道需要调用额外的 API 来传输通知。为了加速应用的响应时间，可以将通知推送到队列中异步发送，而要实现推送通知到队列，可以让对应通知类实现 ShouldQueue 接口并使用 Queueable trait。如果通知类是通过 make:notification 命令生成的，那么该接口和 trait 已经默认导入，你可以快速将它们添加到通知类。

<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;

use Illuminate\Notifications\Notification;

use Illuminate\Contracts\Queue\ShouldQueue;

class InvoicePaid extends Notification implements ShouldQueue

{

use Queueable;

// ...

}

?>

ShouldQueue 接口被添加到通知类以后，你可以像之前一样正常发送通知，Laravel 会自动检测到 ShouldQueue 接口然后将通知推送到队列。

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

//

如果你想要延迟通知的发送，可以在通知实例后加上 delay 方法。

$when = now()->addMinutes(10);

$user->notify((new InvoicePaid($invoice))->delay($when));

(5)、按需通知

有时候你可能需要发送通知给某个用户，但是该用户不存在于应用的用户系统中，要实现这一目的，我们使用 Notification::route 方法， 方法在发送通知之前指定特别的通知路由。

Notification::route('mail', 'taylor@example.com')

->route('nexmo', '5555555555')

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

--

4、邮件通知

(1)、格式化邮件信息

如果通知支持以邮件方式发送，你需要在通知类上定义一个 toMail 方法。该方法会接收一个 $notifiable 实体并返回 Illuminate\Notifications\Messages\MailMessage 实例。邮件消息可以包含多行文本以及对「动作的调用」。 让我们来看一个 toMail 方法的示例。

/\*\*

\* 获取通知对应的邮件。

\*

\* @param mixed $notifiable

\* @return \Illuminate\Notifications\Messages\MailMessage

\*/

public function toMail($notifiable)

{

$url = url('/invoice/'.$this->invoice->id);

return (new MailMessage)

->greeting('Hello!')

->line('One of your invoices has been paid!')

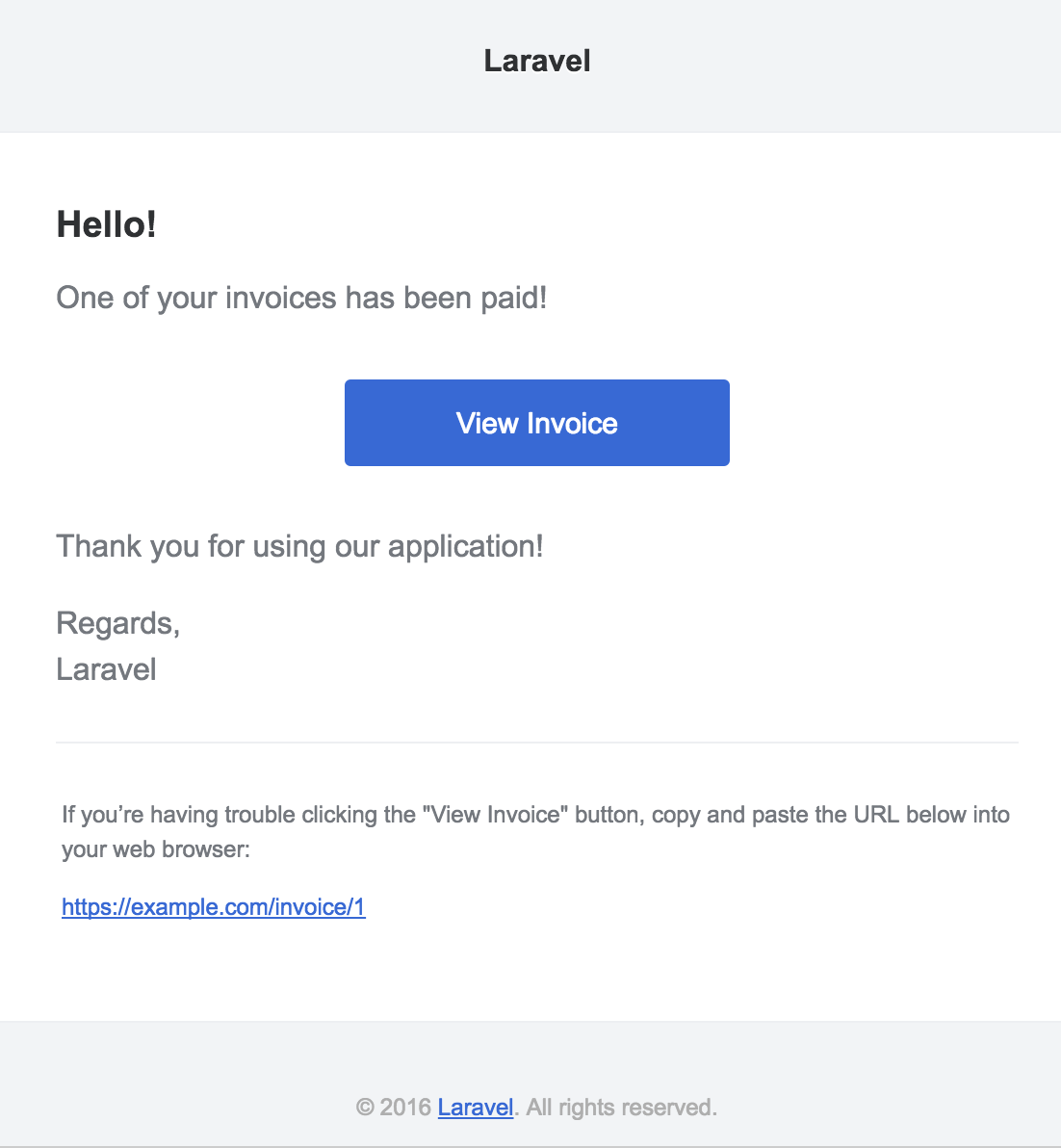
->action('View Invoice', $url)

->line('Thank you for using our application!');

}

提示：请注意我们在 toMail 方法中使用了 $this->invoice->id。你可以传递任何通知生成消息所需要的数据到通知的构造器。

在这个例子中，我们注册了一条问候、一行文本、对动作的调用以及另一行文本。MailMessage 对象提供的这些方法让格式化短小的事务邮件变得简单快捷。mail 通道会将消息组件转化为美观的、响应式的、带有纯文本副本的 HTML 邮件模板。下面是一个通过 mail 通道生成的邮件示例。



( 图 https://laravel.com/img/docs/notification-example.png )

提示： 发送邮件通知时，确保在配置文件 config/app.php 中设置了 name 的值。这个值将会用在邮件通知消息的头部和尾部。

//

///1/ 其他通知格式化选项

除了在通知类中定义多行文本之外，你还可以使用 view 方法来指定一个自定义的、用于渲染通知邮件的模板。

/\*\*

\* 获取通知邮件。

\*

\* @param mixed $notifiable

\* @return \Illuminate\Notifications\Messages\MailMessage

\*/

public function toMail($notifiable)

{

return (new MailMessage)->view(

'emails.name', ['invoice' => $this->invoice]

);

}

此外，你可以从 toMail 方法中返回一个 可邮寄对象( https://learnku.com/docs/laravel/6.x/mail )。

use App\Mail\InvoicePaid as Mailable;

/\*\*

\* Get the mail representation of the notification.

\*

\* @param mixed $notifiable

\* @return Mailable

\*/

public function toMail($notifiable)

{

return (new Mailable($this->invoice))->to($this->user->email);

}

//

///2/ 错误消息

一些通知会告知用户错误信息，例如失败的订单支付。你可以在构建消息的时候调用 error 方法来指示该邮件消息表示错误信息。在邮件消息中使用 error 方法时，动作按钮将会变成红色。

/\*\*

\* 获取通知邮件。

\*

\* @param mixed $notifiable

\* @return \Illuminate\Notifications\Message

\*/

public function toMail($notifiable)

{

return (new MailMessage)

->error()

->subject('Notification Subject')

->line('...');

}

(2)、自定义接收者 (自定义接收人)

通过 mail 通道发送通知时，通知系统会自动在被通知实体上查找 email 属性，你可以通过在该实体上定义一个 routeNotificationForMail 来自定义使用哪个邮箱地址发送通知。

<?php

namespace App;

use Illuminate\Notifications\Notifiable;

use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable

{

use Notifiable;

/\*\*

\* 邮件通道通知的路由。

\*

\* @param \Illuminate\Notifications\Notification $notification

\* @return string

\*/

public function routeNotificationForMail($notification)

{

return $this->email\_address;

}

}

?>

(3)、自定义主题

默认情况下，邮件的主题就是格式为 「标题风格」 的通知类名，因此，如果通知类被命名为 InvoicePaid，邮件的主题就是 Invoice Paid，如果你想要为消息指定明确的主题，可以在构建消息的时候调用 subject 方法。

/\*\*

\* 获取通知的邮件表示。

\*

\* @param mixed $notifiable

\* @return \Illuminate\Notifications\Messages\MailMessage

\*/

public function toMail($notifiable)

{

return (new MailMessage)

->subject('Notification Subject')

->line('...');

}

(4)、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。

// 发布通知扩展包

php artisan vendor:publish --tag=laravel-notifications

--

5、Markdown 邮件通知

Markdown 邮件通知允许你利用邮件通知的预置模板，从而让你可以自由编写更长、更具个性化的消息。因为这些消息以 Markdown 格式编写，Laravel 还可以为它们渲染出高颜值、响应式的 HTML 模板，同时自动生成纯文本的副本。

(1)、生成消息

要生成带有相应 Markdown 模板的通知，可以在使用 Artisan 命令 make:notification 时带上 --markdown 选项。

// 生成Markdown模板通知消息

php artisan make:notification InvoicePaid --markdown=mail.invoice.paid

//

和其他邮件通知一样，使用 Markdown 模板的通知类也要定义一个 toMail 方法。不过，你可以使用 markdown 方法取代构造通知的 line 和 action 方法来指定要使用的 Markdown 模板名称。

/\*\*

\* 获取通知的邮件表示。

\*

\* @param mixed $notifiable

\* @return \Illuminate\Notifications\Messages\MailMessage

\*/

public function toMail($notifiable)

{

$url = url('/invoice/'.$this->invoice->id);

return (new MailMessage)

->subject('Invoice Paid')

->markdown('mail.invoice.paid', ['url' => $url]);

}

(2)、写消息

Markdown 邮件通知联合使用了 Blade 组件和 Markdown 语法，从而让你在不脱离 Laravel 预置组件的情况下轻松构建通知。

~~~ s ~~~

@component('mail::message')

# Invoice Paid

Your invoice has been paid!

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

View Invoice

@endcomponent

Thanks,<br>

{{ config('app.name') }}

@endcomponent

~~~ e ~~~

//

///1/ 按钮组件

按钮组件渲染一个居中的按钮链接。该组件接收两个参数， url 和可选的 color，支持的颜色有 blue， green 和 red。你可以添加任意数量的按钮组件到消息中。

@component('mail::button', ['url' => $url, 'color' => 'green'])

View Invoice

@endcomponent

//

///2/ 面板组件

面板组件将给定的文字区块渲染到一个面板中，并且有一个淡淡的背景色与周围的消息区分开。适用于需要引起注意的文字区块。

@component('mail::panel')

This is the panel content.

@endcomponent

//

///3/ 表格组件

表格组件允许你将一个 Markdown 表格转化为 HTML 表格。该组件接收 Markdown 表格作为其内容。表格列对齐支持使用默认的 Markdown 表格列对齐语法。

@component('mail::table')

| Laravel | Table | Example |

| ------------- |:-------------:| --------:|

| Col 2 is | Centered | $10 |

| Col 3 is | Right-Aligned | $20 |

@endcomponent

(3)、自定义组件

你可以导出所有 Markdown 通知组件到应用中进行自定义，要导出组件，使用 Artisan 命令 vendor:publish 来发布 laravel-mail 资源标签。

// 发布Markdown通知组件

php artisan vendor:publish --tag=laravel-mail

该命令会发布 Markdown 邮件通知组件到 resources/views/vendor/mail 目录。 mail 目录包含 html 和 markdown 目录，每个子目录中又包含各自的所有有效组件。你可以按照自己的喜好自由编辑这些组件。

//

///1/ 自定义 CSS

导出组件之后， resources/views/vendor/mail/html/themes 目录将会包含一个默认的 default.css 文件，你可以在这个文件中自定义 CSS，这样 Markdown 通知的 HTML 样式就会自动调整。

{注：} 如果你想要为 Markdown 组件构建全新的主题，只需在 html/themes 目录中编写一个新的 CSS 文件并修改 mail 配置文件的 theme 选项即可。

--

6、数据库通知

(1)、先决条件 (预备知识)

database 通知通道会在数据表中存储通知信息，该表包含诸如通知类型以及用于描述通知的自定义 JSON 数据之类的信息。

你可以在用户界面中查询这个数据表来展示通知，不过，在此之前，需要创建数据表来保存信息，你可以使用 notifications:table 命令来生成迁移文件然后以此生成相应数据表。

//

//// 通知库表生成

php artisan notifications:table

php artisan migrate

(2)、格式化数据库通知

如果要将通知消息存入到数据库表中，就需要为通知类定义 toDatabase 或 toArray 方法。此方法接收 $notifiable 实体作为参数并返回原生 PHP 数组。返回的数据将被编码为 JSON 格式并存储到 notifications 表的 data 列。来看一个 toArray 方法示例。

/\*\*

\* 获取通知的数组表示。

\*

\* @param mixed $notifiable

\* @return array

\*/

public function toArray($notifiable)

{

return [

'invoice\_id' => $this->invoice->id,

'amount' => $this->invoice->amount,

];

}

//

///1/ toDatabase Vs. toArray

toArray 方法还可以使用 broadcast 通道来判断哪些数据被广播到 JavaScript 客户端。如果想要用两个不同的数组来表示 database 和 broadcast 通道，你需要定义 toDatabase 方法来替代 toArray 方法。

(3)、访问通知

一旦通知存入数据库，就需要用适当的方法在通知实体里访问它们。 在 Lareval 的默认 App\User 模型带有的 Illuminate\Notifications\Notifiable trait 里，它的 notifications Eloquent 关联方法能返回实体通知。要获取通知，可以像其它 Eloquent 关联方法一样访问此方法。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;

}

若要只获取 「未读」通知，可以使用 unreadNotifications 关联方法。同样地，这些通知按照 created\_at 时间戳排序。

$user = App\User::find(1);

foreach ($user->unreadNotifications as $notification) {

echo $notification->type;

}

Tip：若要从 JavaScript 客户端访问通知，需要为应用定义一个通知控制器，它返回可通知实体的通知，比如当前用户。可以从 JavaScript 客户端向该控制器 URI 发送 HTTP 请求。

(4)、标记已读通知 (标记通知已读)

通常，在用户阅览一条通知之后，你会想将其标识为「已读」。 Illuminate\Notifications\Notifiable trait 提供了 markAsRead 方法，它更新数据库中通知记录的 read\_at 列。

$user = App\User::find(1);

foreach ($user->unreadNotifications as $notification) {

$notification->markAsRead();

}

可以在通知控制集合上直接使用 markAsRead 方法代替循环调用通知。

$user->unreadNotifications->markAsRead();

//

还可以使用指更新将所有的通知标为已读，而不从数据库中读取它们。

$user = App\User::find(1);

$user->unreadNotifications()->update(['read\_at' => now()]);

//

可以使用 delete 方法从表中整个删除通知。

$user->notifications()->delete();

--

7、广播通知

(1)、先决条件 (预备知识)

广播通知前，你需要配置和熟悉 Laravel 的 事件广播 ( https://learnku.com/docs/laravel/6.x/broadcasting ) 服务。事件广播提供了在 JavaScript 客户端响应服务端触发 Laravel 事件的方法。

(2)、格式化广播通知

broadcast 通道使用 Laravel 的 event broadcasting ( https://learnku.com/docs/laravel/6.x/broadcasting ) 服务广播通知，它允许 JavaScript 客户端实时捕获通知。如果通知支持广播，你就需要在通知类上定义 toBroadcast 方法。此方法接受 $notifiable 实体作为参数，并返回 BroadcastMessage 实例。返回的数据将被编码为 JSON 并广播给 JavaScript 客户端。我们来看一个 toBroadcast 方法示例。

use Illuminate\Notifications\Messages\BroadcastMessage;

/\*\*

\* 获取通知的可广播表示。

\*

\* @param mixed $notifiable

\* @return BroadcastMessage

\*/

public function toBroadcast($notifiable)

{

return new BroadcastMessage([

'invoice\_id' => $this->invoice->id,

'amount' => $this->invoice->amount,

]);

}

//

///1/ 广播队列配置

所有的广播通知都被放入广播队列。想要配置用于广播操作的队列连接或者队列名称，需要使用 BroadcastMessage 的 onConnection 和 onQueue 方法。

return (new BroadcastMessage($data))

->onConnection('sqs')

->onQueue('broadcasts');

Tip：除了指定的数据，广播通知还包含 type 域，它包括通知类的类名。

(3)、监听通知

通知将会以格式为 {notifiable}.{id} 的形式在私有频道上广播，因此，如果你要发送通知到 ID 为 1 的 App\User 实例，那么该通知将会在私有频道 App.User.1 上进行广播，如果使用了 LaravelEcho ( https://learnku.com/docs/laravel/6.x/broadcasting )，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);

});

//

///1/ 自定义通知通道

如果你想要自定义被通知实体在某个通道上接收广播通知，可以在被通知实体上定义一个 receivesBroadcastNotificationsOn 方法。

<?php

namespace App;

use Illuminate\Notifications\Notifiable;

use Illuminate\Broadcasting\PrivateChannel;

use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable

{

use Notifiable;

/\*\*

\* 用户接收广播通知的通道。

\*

\* @return string

\*/

public function receivesBroadcastNotificationsOn()

{

return 'users.'.$this->id;

}

}

?>

--

8、短信通知 (短信（SMS）通知)

(1)、先决条件 (预备知识)

Laravel 基于 Nexmo ( https://www.nexmo.com/ ) 发送短信通知，在使用 Nexmo 发送通知前，需要安装对应 Composer 依赖包 laravel/nexmo-notification-channel。

// 安装Nexmo依赖包

composer require laravel/nexmo-notification-channel

//

下一步，你需要在配置文件 config/services.php 中进行相应配置。你可以参考以下示例配置。

'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。

(2)、格式化短信通知

如果通知支持以短信方式发送，那么你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');

}

//

///1/ Unicode 内容

如果你的短信消息包含 Unicode 字符，需要在构造 NexmoMessage 实例时调用 unicode 方法。

/\*\*

\* Get the Nexmo / SMS representation of the notification.

\*

\* @param mixed $notifiable

\* @return NexmoMessage

\*/

public function toNexmo($notifiable)

{

return (new NexmoMessage)

->content('Your unicode message')

->unicode();

}

(3)、自定义「From」电话号码 (自定义 “发送” 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');

}

(4)、短信通知路由

使用 nexmo 通道发送通知的时候，通知系统会自动在被通知实体上查找 phone\_number 属性。如果你想要自定义通知被发送到的手机号码，可以在该实体上定义一个 routeNotificationForNexmo 方法。

<?php

namespace App;

use Illuminate\Notifications\Notifiable;

use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable

{

use Notifiable;

/\*\*

\* Route notifications for the Nexmo channel.

\*

\* @param \Illuminate\Notifications\Notification $notification

\* @return string

\*/

public function routeNotificationForNexmo($notification)

{

return $this->phone;

}

}

?>

--

9、Slack 通知

(1)、先决条件 (预备知识)

在通过 Slack 发送通知前，必须通过 Composer 安装 Slack 通知通道。

// 安装Slack依赖包

composer require laravel/slack-notification-channel

//

此外，你还要为 Slack 组配置一个 “Incoming Webhook ( https://api.slack.com/incoming-webhooks )” 集成。该集成会在你进行 Slack通知路由( https://learnku.com/docs/laravel/6.x/notifications/5166#routing-slack-notifications ) 的时候提供一个 URL。

(2)、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。



( 图 https://laravel.com/assets/img/basic-slack-notification.png )

//

///1/ 自定义发送者和接收者

你可以使用 from 和 to 方法自定义发送者和接收者， from 方法接收一个用户名和 emoji 标识，而 to 方法接收通道或用户名。

/\*\*

\* Get the Slack representation of the notification.

\*

\* @param mixed $notifiable

\* @return SlackMessage

\*/

public function toSlack($notifiable)

{

return (new SlackMessage)

->from('Ghost', ':ghost:')

->to('#other')

->content('This will be sent to #other');

}

还可以使用图片作为 logo 用以取代 emoji。

/\*\*

\* Get the Slack representation of the notification.

\*

\* @param mixed $notifiable

\* @return SlackMessage

\*/

public function toSlack($notifiable)

{

return (new SlackMessage)

->from('Laravel')

->image('https://laravel.com/favicon.png')

->content('This will display the Laravel logo next to the message');

}

(3)、Slack 附件

你还可以添加 “附件” 到 Slack 消息。相对简单文本消息，附件可以提供更加丰富的格式选择。在这个例子中，我们会发送一个在应用程序中出现的异常错误通知，包含查看更多异常细节的链接。

/\*\*

\* 获取 Slack 形式的通知。

\*

\* @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) {

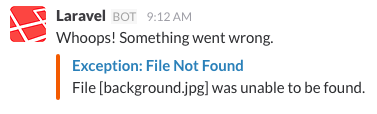
$attachment->title('Exception: File Not Found', $url)

->content('File [background.jpg] was not found.');

});

}

上述代码会生成如下 Slack 消息。



( 图 https://laravel.com/assets/img/basic-slack-attachment.png )

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。

/\*\*

\* 获取 Slack 形式的通知。

\*

\* @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',

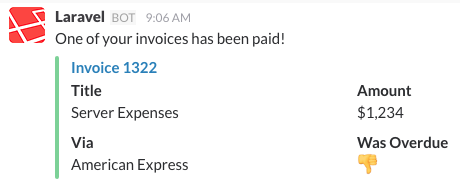
'Was Overdue' => ':-1:',

]);

});

}

上述代码会生成如下 Slack 消息。



( 图 https://laravel.com/assets/img/slack-fields-attachment.png )

//

///1/ Markdown 附件内容

如果一些附件字段包含 Markdown，可以使用 markdown 方法来构建 Slack 用以解析并显示以 Markdown 格式编写的附件字段，该方法支持的值包括 pretext、 text 或 fields。想要了解更多关于 Slack 格式化的信息，查看 Slack API 文档 ( https://api.slack.com/docs/message-formatting#message\_formatting )。

/\*\*

\* 获取 Slack 形式的通知。

\*

\* @param mixed $notifiable

\* @return SlackMessage

\*/

public function toSlack($notifiable)

{

$url = url('/exceptions/'.$this->exception->id);

return (new SlackMessage)

->error()

->content('Whoops! Something went wrong.')

->attachment(function ($attachment) use ($url) {

$attachment->title('Exception: File Not Found', $url)

->content('File [background.jpg] was \*not found\*.')

->markdown(['text']);

});

}

(4)、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.

\*

\* @param \Illuminate\Notifications\Notification $notification

\* @return string

\*/

public function routeNotificationForSlack($notification)

{

return 'https://hooks.slack.com/services/...';

}

}

?>

--

10、本地化通知

Laravel 允许您以当前语言环境之外的其他语言发送通知，并且会在通知队列时记住该语言环境。

要实现这一点， Illuminate\Notifications\Notification 类提供了一个 locale 方法来设置所需的语言。在格式化通知时，应用程序将更改为此语言设置，然后在格式化完成后还原为以前的语言设置。

// 本地化通知

$user->notify((new InvoicePaid($invoice))->locale('es'));

//

多重通知的本地化也可通过 Notification Facade 实现。

Notification::locale('es')->send($users, new InvoicePaid($invoice));

(1)、用户首选语言区域设置

有些情况下，应用程序保存了每个用户的首选语言区域设置。通过在模型上实现 HasLocalePreference 契约，可以指定 Laravel 在发送通知时使用用户保存的首选语言设置。

use Illuminate\Contracts\Translation\HasLocalePreference;

class User extends Model implements HasLocalePreference

{

/\*\*

\* 获取用户的首选区域。

\*

\* @return string

\*/

public function preferredLocale()

{

return $this->locale;

}

}

实现接口后，Laravel 将在向模型发送通知和邮件时自动使用首选区域设置。因此，使用此接口时不需要调用 locale 方法。

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

--

11、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。

/\*\*

\* 事件监听器的 map 映射

\*

\* @var array

\*/

protected $listen = [

'Illuminate\Notifications\Events\NotificationSent' => [

'App\Listeners\LogNotification',

],

];

{提示} 在 EventServiceProvider 中注册监听器之后，使用 Artisan 命令 event:generate 可以快速生成监听器类。

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。

/\*\*

\* 处理事件。

\*

\* @param NotificationSent $event

\* @return void

\*/

public function handle(NotificationSent $event)

{

// $event->channel

// $event->notifiable

// $event->notification

// $event->response

}

--

12、自定义频道

Laravel 为我们提供了多种通知通道，但是尝试编写自定义通道驱动以通过其他通道发送通知，也很简单。首先定义一个包含 send 方法的类，该方法接收两个参数： $notifiable 和 $notification。

<?php

namespace App\Channels;

use Illuminate\Notifications\Notification;

class VoiceChannel

{

/\*\*

\* 发送指定的通知.

\*

\* @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...

}

}

?>

一旦通知通道类被定义后，就可以在应用中通过 via 方法返回类名。

<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;

use App\Channels\VoiceChannel;

use App\Channels\Messages\VoiceMessage;

use Illuminate\Notifications\Notification;

use Illuminate\Contracts\Queue\ShouldQueue;

class InvoicePaid extends Notification

{

use Queueable;

/\*\*

\* 获取通知通道.

\*

\* @param mixed $notifiable

\* @return array|string

\*/

public function via($notifiable)

{

return [VoiceChannel::class];

}

/\*\*

\* 获取语音形式的通知.

\*

\* @param mixed $notifiable

\* @return VoiceMessage

\*/

public function toVoice($notifiable)

{

// ...

}

}

?>

--

--

十、扩展包开发

--

https://learnku.com/docs/laravel/6.x/packages/5167

--

1、简介

扩展包是向 Laravel 中添加功能的主要方式。扩展包可以包含很多有用的功能，例如时间处理扩展包 Carbon ( https://github.com/briannesbitt/Carbon )，亦或者是提供完整 BDD 测试框架的扩展包 Behat ( https://github.com/Behat/Behat )。

当然，扩展包有很多种类型。有些扩展包是独立运行的，意味着他们可以在任意的 PHP 框架中使用。Carbon 和 Behat 就是这样的独立扩展包。要在 Laravel 中使用这种扩展包只需要在 composer.json 文件中引入他们即可。

另一方面，有些扩展包只能在 Laravel 中使用。这些扩展包可能包含专门用来增强 Laravel 应用的路由、控制器、视图和配置的文件。这份指南主要介绍 Laravel 扩展包的开发。

(1)、Facades 注解

当开发 Laravel 应用时，通常使用契约（contracts）或 facades 没有什么区别，因为他们都提供基本相同的可测试能力。然而，在进行扩展包开发的时候，扩展包并不能访问 Laravel 提供的所有测试辅助函数。如果你想像在 Laravel 应用中一样编写扩展包的测试用例，你可以使用扩展包 Orchestral Testbench ( https://github.com/orchestral/testbench )。

--

2、发现扩展包

在 Laravel 应用的 config/app.php 配置文件中，providers 选项定义了能够被 Laravel 加载的服务提供者列表。当有人安装你的扩展包时，你需要将你的服务提供者包含到这个列表中。你可以将服务提供者定义到扩展包的 composer.json 文件中的 extra 部分，而不是让用户手动将你的服务提供者添加到列表中。除了服务提供者，还可以列出所有你想注册的 facades ( https://learnku.com/docs/laravel/6.x/facades )。

"extra": {

"laravel": {

"providers": [

"Barryvdh\\Debugbar\\ServiceProvider"

],

"aliases": {

"Debugbar": "Barryvdh\\Debugbar\\Facade"

}

}

},

(1)、选择性的发现扩展包

如果你是扩展包的用户，想要禁止一个扩展包被发现，你可以在应用的 composer.json 文件中的 extra 部分列出这个扩展包。

"extra": {

"laravel": {

"dont-discover": [

"barryvdh/laravel-debugbar"

]

}

},

你也可以通过在应用的 dont-discover 指令中使用 \* 字符，禁用扩展包发现功能。

"extra": {

"laravel": {

"dont-discover": [

"\*"

]

}

},

--

3、服务提供者

服务提供者 ( https://learnku.com/docs/laravel/6.x/providers ) 让你的扩展包与 Laravel 联系在一起。服务提供者负责将一些东西绑定到 Laravel 的 服务容器 ( https://learnku.com/docs/laravel/6.x/container ) 中，并且告诉 Laravel 从哪里加载扩展包的资源文件，例如视图、配置文件、语言包等。

服务提供者继承了 Illuminate\Support\ServiceProvider 类，并包含了两个方法：register 和 boot。基类 ServiceProvider 位于名为 illuminate/support 的 Composer 扩展包中，你必须将它加入到你的扩展包依赖。想要了解更多关于服务提供者的结构和用途，请查阅 它的文档 ( https://learnku.com/docs/laravel/6.x/providers )。

--

4、资源文件

(1)、配置

有时，你需要将扩展包配置文件发布到应用本身的 config 目录中。这样使用扩展包的用户就可以轻松的重写默认配置项。想要发布扩展包配置文件，只需要在服务提供者的 boot 方法中调用 publishes 方法即可。

/\*\*

\* 在注册后启动服务

\*

\* @return void

\*/

public function boot()

{

$this->publishes([

\_\_DIR\_\_.'/path/to/config/courier.php' => config\_path('courier.php'),

]);

}

现在，当扩展包的用户执行 Laravel 的 vendor:publish 命令，扩展包文件就会被复制到指定的目录中。当然，一旦你的配置文件被发布，就可以如同其他配置一样被访问。

$value = config('courier.option');

{注意} 你不应该在配置文件中定义闭包函数。因为当用户执行 config:cache Artisan 命令时，配置文件将不能被正确的序列化。

//

///1/ 扩展包默认配置

你也可以将扩展包默认配置与应用的副本配置合并在一起。这样扩展包用户就可以在副本配置文件中定义他们想要覆盖的配置选项。想要合并配置，只需要在服务提供者的 register 方法中调用 mergeConfigFrom 方法即可。

/\*\*

\* 在容器中注册绑定

\*

\* @return void

\*/

public function register()

{

$this->mergeConfigFrom(

\_\_DIR\_\_.'/path/to/config/courier.php', 'courier'

);

}

{注意} 此方法只合并配置数组的第一维。如果扩展包用户定义了多维配置数组，缺少的选项将不会被合并。

(2)、路由

如果你的扩展包中包含路由文件，你需要使用 loadRoutesFrom 方法加载他们。此方法将自动判断应用的路由是否已被缓存，如果路由已缓存，将不会加载你的路由文件。

/\*\*

\* 在注册后启动服务

\*

\* @return void

\*/

public function boot()

{

$this->loadRoutesFrom(\_\_DIR\_\_.'/routes.php');

}

(3)、数据库迁移

如果你的扩展包中包含 数据库迁移 ( https://learnku.com/docs/laravel/6.x/migrations )，你需要使用 loadMigrationsFrom 方法告知 Laravel 如何加载他们。loadMigrationsFrom 方法只需要扩展包迁移文件路径作为唯一参数。

/\*\*

\* 在注册后启动服务

\*

\* @return void

\*/

public function boot()

{

$this->loadMigrationsFrom(\_\_DIR\_\_.'/path/to/migrations');

}

一旦你的扩展包迁移文件被注册，当运行 php artisan migrate 命令时他们就会被自动执行。你不需要将他们导入到应用的 database/migrations 目录中。

(4)、语言包

如果你的扩展包中包含 语言包文件 ( https://learnku.com/docs/laravel/6.x/localization )，你需要使用 loadTranslationsFrom 方法告知 Laravel 如何加载他们。例如，如果你的扩展包名为 courier，你需要将下面的内容加入到服务提供者的 boot 方法中。

/\*\*

\* 在注册后启动服务

\*

\* @return void

\*/

public function boot()

{

$this->loadTranslationsFrom(\_\_DIR\_\_.'/path/to/translations', 'courier');

}

扩展包翻译约定使用 package::file.line 语法进行引用。因此，你可以按照下面的方式来加载 courier 扩展包中的 messages 文件的 welcome 行。

echo trans('courier::messages.welcome');

//

///1/ 发布语言包

如果你想要将扩展包中的语言包发布到应用的 resources/lang/vendor 目录，可以使用服务提供者的 publishes 方法。publishes 方法接收一个包含语言包路径和对应发布位置的数组。例如，发布 courier 扩展包的语言包文件，操作如下。

/\*\*

\* 在注册后启动服务

\*

\* @return void

\*/

public function boot()

{

$this->loadTranslationsFrom(\_\_DIR\_\_.'/path/to/translations', 'courier');

$this->publishes([

\_\_DIR\_\_.'/path/to/translations' => resource\_path('lang/vendor/courier'),

]);

}

现在，当扩展包的用户执行 Laravel 的 vendor:publish Artisan 命令，语言包将会被发布到指定的目录中。

(5)、视图

想要在 Laravel 中注册你的扩展包的 视图 ( https://learnku.com/docs/laravel/6.x/views )， 需要告知 Laravel 视图文件的位置。你可以使用服务提供者的 loadViewsFrom 方法来实现。loadViewsFrom 方法允许接收两个参数：视图模板路径和扩展包名。例如，如果你的扩展包名为 courier，你需要将下面的内容加入到服务提供者的 boot 方法中。

/\*\*

\* 在注册后启动服务

\*

\* @return void

\*/

public function boot()

{

$this->loadViewsFrom(\_\_DIR\_\_.'/path/to/views', 'courier');

}

扩展包视图约定使用 package::view 语法进行引用。因此，一旦视图路径在服务提供者中注册成功，你可以使用下面的方式来加载 courier 扩展包中的 admin 视图。

Route::get('admin', function () {

return view('courier::admin');

});

//

///1/ 重写扩展包视图

当你使用 loadViewsFrom 方法时，Laravel 实际上在两个位置注册视图：应用的 resources/views/vendor 目录和你的自定义目录。所以，还以 courier 扩展包为例，Laravel 首先会检查开发人员是否在 resources/views/vendor/courier 中提供了一个自定义版本的视图。然后，如果视图尚未被定义，Laravel 将会搜索在 loadViewsFrom 中定义的视图目录。这种方法可以让用户很简单的自定义或重写扩展包的视图。

//

///2/ 发布视图

如果你希望将你的视图发布到应用的 resources/views/vendor 目录中，可以使用服务提供者的 publishes 方法。publishes 方法接收一个包含视图路径和对应发布位置的数组。

/\*\*

\* 在注册后启动服务

\*

\* @return void

\*/

public function boot()

{

$this->loadViewsFrom(\_\_DIR\_\_.'/path/to/views', 'courier');

$this->publishes([

\_\_DIR\_\_.'/path/to/views' => resource\_path('views/vendor/courier'),

]);

}

现在，当扩展包的用户执行 Laravel 的 vendor:publish Artisan 命令，视图将会被发布到自定的目录中。

--

5、命令

想要在 Laravel 中注册扩展包的 Artisan 命令，需要使用 commands 方法。此方法接收一个命令类的数组。一旦这些命令注册成功，可以使用 Artisan 命令行 ( https://learnku.com/docs/laravel/6.x/artisan ) 执行他们。

/\*\*

\* 在注册后启动服务

\*

\* @return void

\*/

public function boot()

{

if ($this->app->runningInConsole()) {

$this->commands([

FooCommand::class,

BarCommand::class,

]);

}

}

--

6、公共资源文件

你的扩展包中可能存在 JavaScript、CSS 和图片之类的资源文件。想要发布这些资源文件到应用的 public 目录，可以使用服务提供者的 publishes 方法。在下面的例子中，我们也可以添加一个 public 资源分类标签，可用于相关发布资源的分类。

/\*\*

\* 在注册后启动服务

\*

\* @return void

\*/

public function boot()

{

$this->publishes([

\_\_DIR\_\_.'/path/to/assets' => public\_path('vendor/courier'),

], 'public');

}

在，当扩展包的用户执行 vendor:publish 命令，你的资源文件将会被复制到指定的目录中。由于每次更新扩展包时通常都需要覆盖资源文件，因此需要使用 --force 标签。

php artisan vendor:publish --tag=public --force

--

7、发布群组文件

你可能想要分别打包发布扩展包资源文件或资源。举个例子，你想让用户单独发布扩展包中的配置文件，而不是被强制发布扩展包中的所有资源文件。你可以通过调用扩展包服务提供者中的 publishes 方法给不同文件打上「标签」。例如，让我们使用扩展包服务提供者中的 boot 方法来定义两个发布群组。

/\*\*

\* 在注册后启动服务

\*

\* @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');

}

现在你的用户就可以通过执行 vendor:publish 命令，根据定义的标签来发布不同的群组文件。

php artisan vendor:publish --tag=config

--

--

十一、队列

--

https://learnku.com/docs/laravel/6.x/queues/5168

--

1、简介

Tip：Laravel 现在为你的 Redis 队列 提供了 Horizon，一个漂亮的仪表盘和配置系统。查看完整的 Horizon documentation( https://learnku.com/docs/laravel/6.x/horizon ) 文档 了解更多信息。

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 请求和响应的时间。

队列配置文件存放在 config/queue.php 文件中。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( https://kr.github.io/beanstalkd/ ), Amazon SQS( https://aws.amazon.com/sqs/ ), Redis( https://redis.io/ )，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。

(1)、连接 Vs. 队列

在开始使用 Laravel 队列前，弄明白 「连接」 和 「队列」 的区别是很重要的。在你的 config/queue.php 配置文件里，有一个 connections 配置选项。这个选项给 Amazon SQS，Beanstalk，或者 Redis 这样的后端服务定义了一个特有的连接。不管是哪一种，一个给定的连接可能会有多个 「队列」，而 「队列」 可以被认为是不同的栈或者大量的队列任务。

要注意的是，queue 配置文件中每个连接的配置示例中都包含一个 queue 属性。队列任务被发给指定连接的时候会被分发到 queue 属性和指定连接相同的队列中。换句话说，如果你分发任务的时候没有定义分配到哪个队列，那么它就会被放到连接配置中 queue 属性所定义的队列中。

// 这个任务将被分发到默认队列...

Job::dispatch();

// 这个任务将被发送到「emails」队列...

Job::dispatch()->onQueue('emails');

有些应用可能不需要把任务发到不同的队列，而只发到一个简单的队列中就行了。但是把任务推到不同的队列仍然是非常有用的，因为 Laravel 队列处理器允许你定义队列的优先级，所以你能给不同的队列划分不同的优先级或者区分不同任务的不同处理方式了。比如说，如果你把任务推到 high 队列中，你就能让队列处理器优先处理这些任务了。

//

///1/ 启动队列处理器，配置队列任务优先级

php artisan queue:work --queue=high,default

(2)、驱动的必要设置

//

///1/ Database

为了使用 database 队列驱动，你需要一张数据表来存储任务。运行 queue:table Artisan 命令来创建这张表的迁移文件。当迁移文件创建好后，你就可以使用 migrate 命令来进行迁移。

php artisan queue:table

php artisan migrate

//

///2/ Redis

为了使用 redis 队列驱动，你需要在 config/database.php 配置文件中配置 Redis 的数据库连接。

//

///3/ Redis 集群

如果你的 Redis 队列驱动使用了 Redis 集群，你的队列名必须包含一个 key hash tag( https://redis.io/topics/cluster-spec#keys-hash-tags )。这是为了确保所有的 Redis 键对于一个队列都被放在同一哈希中。

'redis' => [

'driver' => 'redis',

'connection' => 'default',

'queue' => '{default}',

'retry\_after' => 90,

],

//

///4/ 阻塞

当使用 Redis 队列时，你可以用 block\_for 配置项来具体说明驱动应该在将任务重新放入 Redis 数据库以及处理器轮询之前阻塞多久。

基于你的队列加载来调整这个值比把新任务放入 Redis 数据库轮询要更有效率的多。例如，你可以将这个值设置为 5 来表明这个驱动应该在等待任务可用时阻塞 5 秒。

'redis' => [

'driver' => 'redis',

'connection' => 'default',

'queue' => 'default',

'retry\_after' => 90,

'block\_for' => 5,

],

//

///4/ 其它队列驱动的依赖扩展包

在使用列表里的队列服务前，必须安装以下依赖扩展包。

- Amazon SQS: `aws/aws-sdk-php ~3.0` - Beanstalkd: `pda/pheanstalk ~4.0` - Redis: `predis/predis ~1.0`

--

2、创建任务

(1)、生成任务类

在你的应用程序中，队列的任务类都默认放在 app/Jobs 目录下。如果这个目录不存在，那当你运行 make:job Artisan 命令时目录就会被自动创建。你可以用以下的 Artisan 命令来生成一个新的队列任务。

//

///1/ 生成任务类

php artisan make:job ProcessPodcast

生成的类继承了 Illuminate\Contracts\Queue\ShouldQueue 接口，这意味着这个任务将会被推送到队列中，而不是同步执行。

(2)、任务类结构

任务类的结构很简单，一般来说只会包含一个让队列用来调用此任务的 handle 方法。我们来看一个示例的任务类。这个示例里，假设我们管理着一个播客发布服务，在发布之前需要处理上传播客文件。

//

///1/ 播客发布服务，在发布之前需要处理上传播客文件。

<?php

namespace App\Jobs;

use App\Podcast;

use App\AudioProcessor;

use Illuminate\Bus\Queueable;

use Illuminate\Queue\SerializesModels;

use Illuminate\Queue\InteractsWithQueue;

use Illuminate\Contracts\Queue\ShouldQueue;

use Illuminate\Foundation\Bus\Dispatchable;

class ProcessPodcast implements ShouldQueue

{

use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

protected $podcast;

/\*\*

\* 新建一个任务实例

\*

\* @param Podcast $podcast

\* @return void

\*/

public function \_\_construct(Podcast $podcast)

{

$this->podcast = $podcast;

}

/\*\*

\* 执行任务

\*

\* @param AudioProcessor $processor

\* @return void

\*/

public function handle(AudioProcessor $processor)

{

// 处理上传播客...

}

}

?>

注意，在这个例子中，我们在任务类的构造器中直接传递了一个 Eloquent 模型( https://learnku.com/docs/laravel/6.x/eloquent )。因为我们在任务类里引用了 SerializesModels 这个 trait，使得 Eloquent 模型在处理任务时可以被优雅地序列化和反序列化。如果你的队列任务类在构造器中接收了一个 Eloquent 模型，那么只有可识别出该模型的属性会被序列化到队列里。当任务被实际运行时，队列系统便会自动从数据库中重新取回完整的模型。这整个过程对你的应用程序来说是完全透明的，这样可以避免在序列化完整的 Eloquent 模式实例时所带来的一些问题。

在队列处理任务时，会调用 handle 方法，而这里我们也可以通过 handle 方法的参数类型提示，让 Laravel 的 服务容器( https://learnku.com/docs/laravel/6.x/container ) 自动注入依赖对象。

如果你想完全控制容器如何将依赖对象注入至 handle 方法，可以使用容器的 bindMethod 方法。bindMethod 方法接受一个任务和容器的回调。虽然可以直接在回调中可以调用 handle 方法，但建议应该从 service provider( https://learnku.com/docs/laravel/6.x/providers ) 调用为佳。

use App\Jobs\ProcessPodcast;

$this->app->bindMethod(ProcessPodcast::class.'@handle', function ($job, $app) {

return $job->handle($app->make(AudioProcessor::class));

});

注意：像图片内容这种二进制数据，在放入队列任务之前必须使用 base64\_encode 方法转换一下。否则，当这项任务放置到队列中时，可能无法正确序列化为 JSON。

(3)、任务中间件

任务中间件允许您针对队列任务的执行包装自定义逻辑，从而减少任务本身的样板代码。 例如，考虑下面的 handle 方法，该方法利用了 Laravel 的 Redis 速率限制功能，使得每 5 秒只允许处理一个任务。

/\*\*

\* 执行队列.

\*

\* @return void

\*/

public function handle()

{

Redis::throttle('key')->block(0)->allow(1)->every(5)->then(function () {

info('Lock obtained...');

// 处理队列...

}, function () {

// 无法获取锁…

return $this->release(5);

});

}

虽然这段代码有效，但 handle 方法的结构变得有噪声，因为它与 Redis 速率限制逻辑混杂在一起。此外，对于我们要进行速率限制的任何其他任务，必须复制此速率限制逻辑。

和 handle 方法中的速率限制不同，我们可以定义一个处理速率限制的任务中间件。Laravel 没有任务中间件的默认位置，因此欢迎您将应用程序中间件放在应用程序的任何位置。在此示例中，我们将中间件放在 app/Jobs/Middleware 目录中。

//

///1/ 任务中间件 (app/Jobs/Middleware)

<?php

namespace App\Jobs\Middleware;

use Illuminate\Support\Facades\Redis;

class RateLimited

{

/\*\*

\* 处理队列中的任务.

\*

\* @param mixed $job

\* @param callable $next

\* @return mixed

\*/

public function handle($job, $next)

{

Redis::throttle('key')

->block(0)->allow(1)->every(5)

->then(function () use ($job, $next) {

// 锁定…

$next($job);

}, function () use ($job) {

// 无法获取锁…

$job->release(5);

});

}

}

?>

正如您所看到的，就像 路由中间件( https://learnku.com/docs/laravel/6.x/middleware )，任务中间件接收正在处理的任务以及应该调用以继续处理任务的回调。

//

///2/ 使用任务中间件

创建任务中间件后，可以通过从任务的 middleware 方法返回它们来将它们附加到任务。此方法不存在于由 make:job Artisan 命令构建的任务上，因此您需要将其添加到自己的任务类定义中。

use App\Jobs\Middleware\RateLimited;

/\*\*

\* 得到任务应该经过的中间人。

\*

\* @return array

\*/

public function middleware()

{

return [new RateLimited];

}

--

3、分发任务

一旦你写完了你的任务类你就可以使用它自带的 dispatch 方法分发它。传递给 dispatch 方法的参数将会被传递给任务的构造函数。

<?php

namespace App\Http\Controllers;

use App\Jobs\ProcessPodcast;

use Illuminate\Http\Request;

use App\Http\Controllers\Controller;

class PodcastController extends Controller

{

/\*\*

\* 存储一个新的播客节目。

\*

\* @param Request $request

\* @return Response

\*/

public function store(Request $request)

{

// 创建播客...

ProcessPodcast::dispatch($podcast);

}

}

?>

(1)、延迟分发

如果你想延迟你的队列任务的执行，你可以在分发任务的时候使用 delay 方法。例如，让我们详细说明一个十分钟之后才会执行的任务。

<?php

namespace App\Http\Controllers;

use App\Jobs\ProcessPodcast;

use Illuminate\Http\Request;

use App\Http\Controllers\Controller;

class PodcastController extends Controller

{

/\*\*

\* 存储一个新的播客节目.

\*

\* @param Request $request

\* @return Response

\*/

public function store(Request $request)

{

// 创建播客...

ProcessPodcast::dispatch($podcast)

->delay(now()->addMinutes(10));

}

}

?>

注意：Amazon SQS 队列服务最大延迟 15 分钟的时间。

(2)、同步调度

如果您想立即（同步）执行队列任务，可以使用 dispatchNow 方法。 使用此方法时，队列任务将不会排队，并立即在当前进程中运行。

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Jobs\ProcessPodcast;

use App\Http\Controllers\Controller;

class PodcastController extends Controller

{

/\*\*

\* 存储一个新的播客节目。

\*

\* @param Request $request

\* @return Response

\*/

public function store(Request $request)

{

// 创建播客...

ProcessPodcast::dispatchNow($podcast);

}

}

?>

(3)、工作链 (任务链)

任务链允许你具体定义一个按序列执行队列任务的列表。一旦序列中的任务失败了，剩余的工作将不会执行。要运行一个任务链，你可以对可分发的任务使用 withChain 方法。

ProcessPodcast::withChain([

new OptimizePodcast,

new ReleasePodcast

])->dispatch();

注意：使用 $this->delete() 方法删除队列任务不会阻止任务链任务执行。只有当任务链中的任务执行失败时，任务链才会停止执行。

//

///1/ 链连接和队列

如果你想定义用于任务链的默认连接和队列，你可以使用 allOnConnection 和 allOnQueue 方法。 这些方法指定了所需队列的连接和队列 —— 除非队列任务被明确指定给了不同的连接 / 队列。

ProcessPodcast::withChain([

new OptimizePodcast,

new ReleasePodcast

])->dispatch()->allOnConnection('redis')->allOnQueue('podcasts');

(4)、自定义队列 & 连接 (自定义连接 & 队列)

//

///1/ 分发任务到指定队列

通过将任务分发到不同队列，你可以将你的队列任务「分类」，甚至指定给不同队列分配的任务数量。记住，这不是推送任务到你定义的队列配置文件的不同的连接里，而是一个单一的连接。要指定队列，在分发任务时使用 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

{

/\*\*

\* 存储一个新的播客节目.

\*

\* @param Request $request

\* @return Response

\*/

public function store(Request $request)

{

// 创建播客...

ProcessPodcast::dispatch($podcast)->onQueue('processing');

}

}

?>

//

///2/ 分发任务到指定连接

如果你在多队列连接中工作，你可以指定将任务分发到哪个连接。要指定连接，在分发任务时使用 onConnection 方法。

<?php

namespace App\Http\Controllers;

use App\Jobs\ProcessPodcast;

use Illuminate\Http\Request;

use App\Http\Controllers\Controller;

class PodcastController extends Controller

{

/\*\*

\* 存储一个新播客节目。

\*

\* @param Request $request

\* @return Response

\*/

public function store(Request $request)

{

// 创建播客...

ProcessPodcast::dispatch($podcast)->onConnection('sqs');

}

}

?>

当然，你可以链式调用 onConnection 和 onQueue 方法来指定连接和队列。

ProcessPodcast::dispatch($podcast)

->onConnection('sqs')

->onQueue('processing');

或者，可以将 connection 指定为任务类的属性。

<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue

{

/\*\*

\* 应该处理任务的队列连接.

\*

\* @var string

\*/

public $connection = 'sqs';

}

(5)、指定任务最大尝试次数 / 超时值

//

///1/ 最大尝试次数

在一个任务重指定最大尝试次数可以通过 Artisan 命令的 --tries 选项 指定。

php artisan queue:work --tries=3

你可能想通过任务类自身对最大任务尝试次数进行一个更颗粒化的处理。如果最大尝试次数是在任务类中定义的，它将优先于命令行中的值提供。

<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue

{

/\*\*

\* 任务可以尝试的最大次数。

\*

\* @var int

\*/

public $tries = 5;

}

?>

//

///2/ 基于时间的尝试

作为另外一个选择来定义任务在失败前会尝试多少次，你可以定义一个任务超时时间。这样的话，在给定的时间范围内，任务可以无限次尝试。要定义一个任务的超时时间，在你的任务类中新增一个 retryUntil 方法。

/\*\*

\* 定义任务超时时间

\*

\* @return \DateTime

\*/

public function retryUntil()

{

return now()->addSeconds(5);

}

Tip：你也可以在你的队列事件监听器中使用 retryUntil 方法。

//

///3/ 超时

注意：timeout 特性对于 PHP 7.1+ 和 pcntl PHP 扩展进行了优化。

同样的，任务执行最大秒数的数值可以通过 Artisan 命令行的 --timeout 选项指定。

php artisan queue:work --timeout=30

然而，你可能也想在任务类自身定义一个超时时间。如果在任务类中指定，优先级将会高于命令行。

<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue

{

/\*\*

\* 任务可以执行的最大秒数 (超时时间)。

\*

\* @var int

\*/

public $timeout = 120;

}

?>

(6)、访问限制 (频率限制)

注意：这个特性要求你的应用可以使用 Redis 服务器( https://learnku.com/docs/laravel/6.x/redis )。

如果你的应用使用了 Redis，你可以通过时间或并发限制你的队列任务。当你的队列任务通过同样有速率限制的 API 使用时，这个特性将很有帮助。

例如，使用 throttle 方法，你可以限制一个给定类型的任务每 60 秒只执行 10 次。如果没有获得锁，一般情况下你应该将任务放回队列以使其可以被稍后重试。

Redis::throttle('key')->allow(10)->every(60)->then(function () {

// 任务逻辑...

}, function () {

// 无法获得锁...

return $this->release(10);

});

Tip：在上述的例子里，key 可以是任何你想要限制频率的任务类型的唯一识别字符串。例如，使用构件基于任务类名的 key，或它操作的 Eloquent 模型的 ID。

注意：将受限制的作业释放回队列，仍然会增加工作的总数 attempts。

或者，你可以指定一个任务可以同时执行的最大数量。在如下情况时这会很有用处：当一个队列中的任务正在修改资源时，一次只能被一个任务修改。例如，使用 funnel 方法，你可以限制一个给定类型的任务一次只能执行一个处理器。

Redis::funnel('key')->limit(1)->then(function () {

// 任务逻辑...

}, function () {

// 无法获得锁...

return $this->release(10);

});

Tip：当使用频率限制时，任务执行成功的尝试的次数可能会难以确定。所以，将频率限制与 时间限制( https://learnku.com/docs/laravel/6.x/queues/5168#time-based-attempts ) 组合是很有作用的。

(7)、错误处理

如果在任务执行的时候出现异常，任务会被自动释放到队列中以再次尝试。任务将会一直被释放直到达到应用允许的最大重试次数。最大重试的数值由 queue:work Artisan 命令的 --tries 选项定义，或者在任务类中定义。更多执行队列处理器的信息可以 在以下找到( https://learnku.com/docs/laravel/6.x/queues/5168#running-the-queue-worker )。

--

4、排队闭包

你也可以直接调用闭包，而不是将任务类调度到队列中。这对于需要执行的快速、简单的任务非常有用。

$podcast = App\Podcast::find(1);

dispatch(function () use ($podcast) {

$podcast->publish();

});

将闭包分派给队列时，闭包的代码内容将以加密方式签名，因此无法在传输过程中对其进行修改。

--

5、运行队列处理器

Laravel 包含了一个队列处理器以将推送到队列中的任务执行。你可以使用 queue:work Artisan 命令运行处理器。 注意一旦 queue:work 命令开始执行，它会一直运行直到它被手动停止或终端被关闭。

//

///1/ 运行队列处理器

php artisan queue:work

Tip：要使 queue:work 进程一直在后台运行，你应该使用进程管理器比如 Supervisor( https://learnku.com/docs/laravel/6.x/queues/5168#supervisor-configuration ) 来确保队列处理器不会停止运行。

记住，队列处理器是一个常驻的进程并且在内存中保存着已经启动的应用状态。因此，它们并不会在启动后注意到你代码的更改。所以，在你的重新部署过程中，请记得 重启你的队列处理器( https://learnku.com/docs/laravel/6.x/queues/5168#queue-workers-and-deployment )。

//

///2/ 指定连接 & 队列

你也可以具体说明队列处理器应该使用哪个队列连接。 传递给 work 的连接名应该与你的 config/queue.php 配置文件中定义的连接之一相符。

php artisan queue:work redis //指定连接;

//

你甚至可以自定义你的队列处理器使其只执行连接中指定的队列。例如，如果你的所有邮件都由 redis 连接的 emails 队列处理，你可以使用如下的命令启动一个仅执行此队列的处理器。

php artisan queue:work redis --queue=emails //指定连接和队列;

//

///3/ 执行单一任务

--once 选项用于使队列处理器只处理队列中的单一任务。

php artisan queue:work --once

//

///4/ 处理所有队列的任务然后退出

--stop-when-empty 选项可用于处理队列处理器处理所有作业然后优雅地退出。如果您希望在队列为空后关闭容器，则在 Docker 容器中运行 Laravel 队列时，此选项很有用。

php artisan queue:work --stop-when-empty

//

///5/ 资源注意事项

后台驻留的队列处理器不会在执行完每个任务后「重启」框架。因此，你应该在每个任务完成后释放任何占用过大的资源。例如，如果你正在用 GD 库执行图像处理，你应该在完成后使用 imagedestroy 释放内存。

(1)、队列优先级

有时你可能想确定队列执行的优先顺序。例如在 config/queue.php 中你可以将 redis 连接的 queue 队列的优先级从 default 设置为 low。然而， 偶尔你也想像如下方式将一个任务推送到 high 队列。

dispatch((new Job)->onQueue('high'));

要运行一个处理器来确认 low 队列中的任务在全部的 high 队列任务完成后才继续执行，你可以传递一个逗号分隔的队列名列表作为 work 命令的参数。

php artisan queue:work --queue=high,low

(2)、队列处理器 & 部署

因为队列处理器是常驻进程，他们在重启前不会应用你代码的更改。因此，部署使用队列处理器的应用最简单的方法是在部署进程中重启队列处理器。你可以平滑地重启所有队列处理器通过使用 queue:restart 方法。

php artisan queue:restart

这个命令将会引导所有的队列处理器在完成当前任务后平滑「中止」，这样不会有丢失的任务。由于在执行 queue:restart 后队列处理器将会中止，所以你应该运行一个进程管理器例如 Supervisor ( https://learnku.com/docs/laravel/6.x/queues/5168#supervisor-configuration ) 来自动重启队列处理器。

Tip：队列使用 缓存 ( https://learnku.com/docs/laravel/6.x/cache ) 存储重启信号，所以你应该确定在使用这个功能之前配置好缓存驱动。

(3)、任务过期 & 超时

//

///1/ 任务过期

在你的 config/queue.php 配置文件中，每个队列连接都定义了一个 retry\_after 选项。这个选项指定了队列连接在重试一个任务前应该等它执行多久。例如，如果 retry\_after 的值设置为 90 ，那么任务在执行了 90 秒后将会被放回队列而不是删除它。一般情况下，你应该将 retry\_after 的值设置为你认为你的任务可能会执行需要最长时间的值。

注意：只有在 Amazon SQS 中不存在 retry\_after 这个值。 SQS 将会以 AWS 控制台配置的 默认可见超时值( https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/AboutVT.html ) 作为重试任务的依据。

//

///2/ 处理器超时

queue:work Artisan 命令包含一个 --timeout 选项。 --timeout 选项指定了 Laravel 的队列主进程在中止一个执行任务的子进程之前需要等到多久。有时一个子进程可能会因为各种原因「冻结」，比如一个外部的 HTTP 请求失去响应。 --timeout 选项会移除那些超过指定时间被冻结的进程。

php artisan queue:work --timeout=60

retry\_after 配置项和 --timeout 命令行配置并不同，但将它们同时使用可以确保任务不会丢失并且任务只会成功执行一次。

注意：--timeout 的值应该比你在 retry\_after 中配置的值至少短几秒。这会确保处理器永远会在一个任务被重试之前中止。如果你的 --timeout 值比 retry\_after 的值长的话，你的任务可能会被执行两次。

//

///3/ 队列进程睡眠时间

当任务在队列中可用时，处理器将会一直无间隔地处理任务。 然而， sleep 选项定义了如果没有新任务的时候处理器将会「睡眠」多长时间。在处理器睡眠时，它不会处理任何新任务 —— 任务将会在队列处理器再次启动后执行。

php artisan queue:work --sleep=3

--

6、Supervisor 配置

(1)、安装 Supervisor

Supervisor 是 Linux 操作系统下中的一个进程监控器，它可以在 queue:work 挂掉时自动重启之。在 Ubuntu 上安装 Supervisor，你可以使用如下命令。

// Ubuntu安装

sudo apt-get install supervisor

{小提醒} 如果觉得配置 Supervisor 难于登天，可以考虑使用 Laravel Forge( https://forge.laravel.com/ )，它将自动为你的 Laravel 项目安装和配置 Supervisor。

(2)、配置 Supervisor

Supervisor 的配置文件通常位于 /etc/supervisor/conf.d 目录下。在该目录中，你可以创建任意数量的配置文件，用来控制 supervisor 将如何监控你的进程。例如，创建一个 laravel-worker.conf 文件使之启动和监控一个 queue:work 进程。

[program:laravel-worker]

process\_name=%(program\_name)s\_%(process\_num)02d

command=php /home/forge/app.com/artisan queue:work sqs --sleep=3 --tries=3

autostart=true

autorestart=true

user=forge

numprocs=8

redirect\_stderr=true

stdout\_logfile=/home/forge/app.com/worker.log

//

在这个例子中，numprocs 指令将指定 Supervisor 运行 8 个 queue:work 进程并对其进行监控，如果它们挂掉就自动重启它们。你应该更改 command 选项中的 queue：work sqs 部分以表示你所需的队列连接。

(3)、启动 Supervisor

配置文件创建完毕后，你就可以使用如下命令更新 Supervisor 配置并启动进程了。

sudo supervisorctl reread

sudo supervisorctl update

sudo supervisorctl start laravel-worker:\*

获取关于 Supervisor 的更多信息，可以查阅 Supervisor 文档( http://supervisord.org/index.html )。

--

7、处理失败的任务

有时你的队列化任务会执行失败。放平心态，好事多磨。 Laravel 包含了一种方便的方法来指定任务应该尝试的最大次数。如果一个任务已经到达了最大尝试次数，它就会被插入到 failed\_jobs 数据库表中。要创建 failed\_jobs 数据库迁移表，你可以使用 queue:failed-table 命令。

//

//// 迁移任务失败库表

php artisan queue:failed-table

php artisan migrate

然后，当你运行 queue worker( https://learnku.com/docs/laravel/6.x/queues/5168#running-the-queue-worker )，你应该使用 queue：work 命令中的 --tries 开关指定应尝试运行任务的最大次数。 如果没有为 --tries 选项指定值，则将死循环尝试运行任务。

php artisan queue:work redis --tries=3

(1)、清除失败任务 (任务失败后清理)

你可以直接在任务类中定义 failed 方法，允许你在任务失败时执行针对于该任务的清理工作。 这是向用户发送警报或恢复任务执行的任何操作的绝佳位置。导致任务失败的 Exception 将被传递给 failed 方法。

<?php

namespace App\Jobs;

use Exception;

use App\Podcast;

use App\AudioProcessor;

use Illuminate\Bus\Queueable;

use Illuminate\Queue\SerializesModels;

use Illuminate\Queue\InteractsWithQueue;

use Illuminate\Contracts\Queue\ShouldQueue;

class ProcessPodcast implements ShouldQueue

{

use InteractsWithQueue, Queueable, SerializesModels;

protected $podcast;

/\*\*

\* 创建任务实例

\*

\* @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)

{

// 上传播客……

}

/\*\*

\* 任务失败的处理过程

\*

\* @param Exception $exception

\* @return void

\*/

public function failed(Exception $exception)

{

// 给用户发送任务失败的通知，等等……

}

}

?>

(2)、任务失败事件

如果你想在任务失败时注册一个可调用的事件，你可以使用 Queue::failing 方法。该事件是通过 email 或 Slack 通知你团队的绝佳时机。例如，我们可以在 Laravel 中的 AppServiceProvider 中附加一个回调事件。

<?php

namespace App\Providers;

use Illuminate\Support\Facades\Queue;

use Illuminate\Queue\Events\JobFailed;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider

{

/\*\*

\* 启动任意服务。

\*

\* @return void

\*/

public function boot()

{

Queue::failing(function (JobFailed $event) {

// $event->connectionName

// $event->job

// $event->exception

});

}

/\*\*

\* 注册服务提供者。

\*

\* @return void

\*/

public function register()

{

//

}

}

?>

(3)、重试失败的任务

//

//// 查看失败任务

要想查看所有被放入 failed\_jobs 数据表中的任务，你可以使用 Artisan 命令 queue:failed。

php artisan queue:failed

//

//// 指定重试任务

queue:failed 命令会列出任务 ID ，队列，以及失败的时间。任务 ID 可能会被用于重试失败的任务。例如，要重试一个任务 ID 为 5 的任务，使用如下命令。

php artisan queue:retry 5

//

//// 重试所有任务

要重试所有失败的任务，执行 queue:retry 命令，将 all 作为 ID 传入。

php artisan queue:retry all

//

//// 删除指定任务

如果你想删除一个失败的任务，使用 queue:forget 命令。

php artisan queue:forget 5

//

//// 删除所有任务

要清空所有失败的任务，使用 queue:flush 命令。

php artisan queue:flush

(4)、忽略缺失的模型

在向任务中注入 Eloquent 模型时，模型被放入队列前将被自动序列化并在执行任务时还原。但是，如果在任务等待执行时删除了模型，任务可能会失败并抛出 ModelNotFoundException。

为了方便，你可以选择设置任务的 deleteWhenMissingModels 属性为 true 来自动地删除缺失模型的任务。

/\*\*

\* 如果模型缺失即删除任务。

\*

\* @var bool

\*/

public $deleteWhenMissingModels = true;

--

8、任务事件

通过在 Queue facade( https://learnku.com/docs/laravel/6.x/facades ) 中使用 before 和 after 方法，你可以指定一个队列任务被执行前后的回调。这些回调是添加额外的日志或增加统计的绝好时机。通常，你应该在 服务提供者( https://learnku.com/docs/laravel/6.x/providers )中调用这些方法。例如，我们可以使用 Laravel 的 AppServiceProvider。

<?php

namespace App\Providers;

use Illuminate\Support\Facades\Queue;

use Illuminate\Support\ServiceProvider;

use Illuminate\Queue\Events\JobProcessed;

use Illuminate\Queue\Events\JobProcessing;

class AppServiceProvider extends ServiceProvider

{

/\*\*

\* 引导启动任意应用服务。

\*

\* @return void

\*/

public function boot()

{

Queue::before(function (JobProcessing $event) {

// $event->connectionName

// $event->job

// $event->job->payload()

});

Queue::after(function (JobProcessed $event) {

// $event->connectionName

// $event->job

// $event->job->payload()

});

}

/\*\*

\* 注册服务提供者。

\*

\* @return void

\*/

public function register()

{

//

}

}

?>

在 Queue facade( https://learnku.com/docs/laravel/6.x/facades ) 使用 looping 方法可以在处理器尝试获取任务之前执行回调。例如，你也许想用一个闭包来回滚之前失败的任务尚未关闭的事务。

Queue::looping(function () {

while (DB::transactionLevel() > 0) {

DB::rollBack();

}

});

--

--

十二、任务调度

--

https://learnku.com/docs/laravel/6.x/scheduling/5169

--

1、简介

过去，你可能需要在服务器上为每一个调度任务去创建 Cron 入口。但是这种方式很快会变得不友好，因为这些任务调度不在源代码中，并且你每次都需要通过 SSH 链接登录到服务器中才能增加 Cron 入口。

Laravel 命令行调度器允许你在 Laravel 中对命令调度进行清晰流畅的定义。且使用这个任务调度器时，你只需要在你的服务器上创建单个 Cron 入口。你的任务调度在 app/Console/Kernel.php 的 schedule 方法中进行定义。为了帮助你更好的入门，这个方法中有个简单的例子。

(1)、启动调度器

当使用这个调度器时，你只需要把下面的 Cron 入口添加到你的服务器中。如果你不知道怎么在服务器中添加 Cron 入口，可以考虑使用一些服务来管理 Cron 入口，比如 Laravel Forge ( https://forge.laravel.com/ )。

\* \* \* \* \* cd /path-to-your-project && php artisan schedule:run >> /dev/null 2>&1

\* \* \* \* \*

- - - - -

| | | | |

| | | | +----- 星期中星期几 (0 - 7) (星期天 为0)

| | | +---------- 月份 (1 - 12)

| | +--------------- 一个月中的第几天 (1 - 31)

| +-------------------- 小时 (0 - 23)

+------------------------- 分钟 (0 - 59)

这个 Cron 会每分钟执行一次 Laravel 的命令行调度器。当 schedule:run 命令被执行的时候，Laravel 会根据你的调度执行预定的程序。

--

2、定义调度

你可以在 App\Console\Kernel 类的 schedule 方法中定义所有的调度任务。 在开始之前，让我们来看一个例子。在此例中，我们计划每天午夜调用一个闭包。在闭包中，我们会执行一个数据库查询来清空一张表。

<?php

namespace App\Console;

use Illuminate\Support\Facades\DB;

use Illuminate\Console\Scheduling\Schedule;

use Illuminate\Foundation\Console\Kernel as ConsoleKernel;

class Kernel extends ConsoleKernel

{

/\*\*

\* 应用中自定义的 Artisan 命令

\*

\* @var array

\*/

protected $commands = [

//

];

/\*\*

\* 定义计划任务

\*

\* @param \Illuminate\Console\Scheduling\Schedule $schedule

\* @return void

\*/

protected function schedule(Schedule $schedule)

{

$schedule->call(function () {

DB::table('recent\_users')->delete();

})->daily();

}

}

?>

除了使用闭包来定义任务调度外，你也可以使用 invokable objects ( https://secure.php.net/manual/en/language.oop5.magic.php#object.invoke )。

Invokable objects 指的是 PHP 中含有 \_\_invoke 方法的类。

$schedule->call(new DeleteRecentUsers)->daily();

(1)、Artisan 命令调度

除了调用闭包这种方式来调度外，你还可以调用 Artisan commands 和操作系统命令。例如，你可以给 command 方法传递命令名称或者类来调度一个 Artisan 命令。

$schedule->command('emails:send Taylor --force')->daily();

$schedule->command(EmailsCommand::class, ['Taylor', '--force'])->daily();

(2)、队列任务调度

job 方法可以用来调度 队列任务 ( https://learnku.com/docs/laravel/6.x/queues )。 此方法提供了一种快捷方式来调度任务，而无需使用 call 方法创建闭包来调度任务。

$schedule->job(new Heartbeat)->everyFiveMinutes();

// 分发任务到「heartbeats」队列...

$schedule->job(new Heartbeat, 'heartbeats')->everyFiveMinutes();

(3)、Shell 命令调度 (Shell 调度命令)

exec 方法可用于向操作系统发送命令。

$schedule->exec('node /home/forge/script.js')->daily();

(4)、调度频率设置

你还可以给你的任务分配多种调度方式。

(方法/描述)

->cron('\* \* \* \* \*'); 自定义 Cron 计划执行任务

->everyMinute(); 每分钟执行一次任务

->everyFiveMinutes(); 每五分钟执行一次任务

->everyTenMinutes(); 每十分钟执行一次任务

->everyFifteenMinutes(); 每十五分钟执行一次任务

->everyThirtyMinutes(); 每三十分钟执行一次任务

->hourly(); 每小时执行一次任务

->hourlyAt(17); 每小时第 17 分钟执行一次任务

->daily(); 每天午夜执行一次任务（译者注：每天零点）

->dailyAt('13:00'); 每天 13 点执行一次任务

->twiceDaily(1, 13); 每天 1 点及 13 点各执行一次任务

->weekly(); 每周执行一次任务

->weeklyOn(1, '8:00'); 每周一的 8 点执行一次任务

->monthly(); 每月执行一次任务

->monthlyOn(4, '15:00'); 每月 4 号的 15 点 执行一次任务

->quarterly(); 每季度执行一次任务

->yearly(); 每年执行一次任务

->timezone('America/New\_York'); 设置时区

//

结合额外的限制条件，我们可以生成在一周中的特定时间运行的计划任务。例如，在每周一执行命令。

// 每周一 13:00 执行...

$schedule->call(function () {

//

})->weekly()->mondays()->at('13:00');

// 工作日（周一至周五） 8 点 至 17 点每小时执行一次...

$schedule->command('foo')

->weekdays()

->hourly()

->timezone('America/Chicago')

->between('8:00', '17:00');

额外的限制条件列表如下。

(方法/描述)

->weekdays(); 限制任务在工作日执行

->weekends(); 限制任务在周末执行

->sundays(); 限制任务在周日执行

->mondays(); 限制任务在周一执行

->tuesdays(); 限制任务在周二执行

->wednesdays(); 限制任务在周三执行

->thursdays(); 限制任务在周四执行

->fridays(); 限制任务在周五执行

->saturdays(); 限制任务在周六执行

->between($start, $end); 限制任务在 $start 和 $end 区间执行

->when(Closure); 限制任务在闭包返回为真时执行

->environments($env); 限制任务在特定环境中执行

//

///1/ 时间范围限制

between 方法可用于限制任务在一天中的某个时间段执行。

$schedule->command('reminders:send')

->hourly()

->between('7:00', '22:00');

相似地， unlessBetween 方法也可用于限制任务不在一天中的某个时间段执行。

$schedule->command('reminders:send')

->hourly()

->unlessBetween('23:00', '4:00');

//

///2/ 闭包检测限制

when 方法可根据检测结果来执行任务。换言之，若给定的闭包返回 true ，若无其他限制条件阻止，任务就会一直执行。

$schedule->command('emails:send')->daily()->when(function () {

return true;

});

skip 可看作是 when 的逆方法。若 skip 方法返回 true ，任务将不会执行。

$schedule->command('emails:send')->daily()->skip(function () {

return true;

});

当链式调用 when 方法时，仅当所有 when 都返回 true 时，任务才会执行。

//

///3/ 环境限制

environments 方法可限制任务在给定环境中执行。

$schedule->command('emails:send')

->daily()

->environments(['staging', 'production']);

(5)、时区

timezone 方法可指定任务在给定时区中的时间执行。

$schedule->command('report:generate')

->timezone('America/New\_York')

->at('02:00');

若想给所有计划任务分配相同的时区，那么需要在 app/Console/Kernel.php 文件中定义 scheduleTimezone 方法。这个方法会返回一个默认时区，最终分配给所有计划任务。

/\*\*

\* 获取计划事件默认使用的时区。

\*

\* @return \DateTimeZone|string|null

\*/

protected function scheduleTimezone()

{

return 'America/Chicago';

}

注意：请记住，有些时区会使用夏令时。当夏令时发生调整时，你的任务可能会执行两次，甚至根本不会执行。因此，我们建议尽可能避免使用时区来安排计划任务。

(6)、避免任务重复

默认情况下，即使之前的任务实例还在执行，调度内的任务也会执行。为避免这种情况的发生，你可以使用 withoutOverlapping 方法。

$schedule->command('emails:send')->withoutOverlapping();

在此例中，若 emails:send Artisan ( https://learnku.com/docs/laravel/6.x/artisan ) 命令 还未运行，那它将会每分钟执行一次。当你的任务执行时间非常不确定，导致你无法准确预测任务的执行时间，那 withoutOverlapping 方法会特别有用。

如有需要，你可以在「without overlapping」锁过期之前，指定它的过期时间。默认情况下，这个锁会在 24 小时后过期。

$schedule->command('emails:send')->withoutOverlapping(10);

(7)、任务只运行在一台服务器上

注意：你的应用默认缓存驱动必须是 memcached 或 redis 才能使用这个特性。除此之外，所有服务器必须使用同一台中央缓存服务器来通信。

若你的应用在多台服务器中运行，你可能需要限制某个计划任务仅在单台服务器上运行。假如你有一个计划任务：在每周五晚生成一份新报告。若此任务调度器在三台服务器上运行，那么该任务就会在三台服务器上运行并生成三份报告。这样不好。

当你定义计划任务时，可使用 onOneServer 方法表明该任务仅在单台服务器上运行。如此，第一台获取到该任务的服务器，会给它附加一个原子锁，这样便可防止其他服务器在同一时间内执行同一个任务。

$schedule->command('report:generate')

->fridays()

->at('17:00')

->onOneServer();

(8)、后台任务

默认情况下，计划同时运行的多个命令将会顺序执行。若你有长时间运行的命令，这可能导致后续命令的启动时间比预期的更晚。你可以使用 runInBackground 方法让命令在后台运行，如此，它们便可同时运行了。

$schedule->command('analytics:report')

->daily()

->runInBackground();

注意：仅能在通过 command 和 exec 方法调度任务时，方能使用 runInBackground 方法。

(9)、维护模式

Laravel 的队列任务在 维护模式 ( https://learnku.com/docs/laravel/5.8/configuration#maintenance-mode ) 下不会运行。因为我们不想你的调度任务干扰到你服务器上可能还未完成的项目。不过，如果你确实是想在维护模式下强制调度任务执行，你可以使用 evenInMaintenanceMode 方法。

$schedule->command('emails:send')->evenInMaintenanceMode();

--

3、任务输出

Laravel 调度器提供了一些方便的方法来处理调度任务输出。首先，你可以使用 sendOutputTo 方法来输出到文件以便于后续检查。

$schedule->command('emails:send')

->daily()

->sendOutputTo($filePath);

如果希望将输出 附加 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指定邮箱。在使用邮件发送之前，你需要配置 Laravel 的 邮件服务 ( https://learnku.com/docs/laravel/6.0/mail )。

$schedule->command('foo')

->daily()

->sendOutputTo($filePath)

->emailOutputTo('foo@example.com');

如果你只想在命令执行失败的时候输出到邮箱，你可以使用 emailOutputOnFailure 方法。

$schedule->command('foo')

->daily()

->emailOutputOnFailure('foo@example.com');

注意：emailOutputTo，emailOutputOnFailure，sendOutputTo 和 appendOutputTo 方法是 command 和 exec 独有的。

--

4、任务钩子

使用 before 和 after 方法，你可以在调度任务执行前或者执行后来执行特定代码。

$schedule->command('emails:send')

->daily()

->before(function () {

// 任务即将开始...

})

->after(function () {

// 任务完成...

});

onSuccess 和 onFailure 方法允许您指定计划任务成功或失败时要执行的代码。

$schedule->command('emails:send')

->daily()

->onSuccess(function () {

// 任务成功...

})

->onFailure(function () {

// 任务失败...

});

(1)、Pinging 网址

使用 pingBefore 和 thenPing 方法，你可以在任务执行前或者执行后来 ping 指定的 URL。这个方法在通知外部服务（如 Laravel Envoyer ( https://envoyer.io/ )）计划任务将要执行或者已完成时将会特别有用。

$schedule->command('emails:send')

->daily()

->pingBefore($url)

->thenPing($url);

只有在给定条件为 true 时，才能使用 pingBeforeIf 和 thenPingIf 方法 ping 指定的 URL。

$schedule->command('emails:send')

->daily()

->pingBeforeIf($condition, $url)

->thenPingIf($condition, $url);

在任务成功或失败时，使用 pingOnSuccess 和 pingOnFailure 方法 ping 给定网址。

$schedule->command('emails:send')

->daily()

->pingOnSuccess($successUrl)

->pingOnFailure($failureUrl);

所有 ping 方法都需要 Guzzle HTTP 库。你可以使用 Composer 来添加 Guzzle 到你的项目中。

composer require guzzlehttp/guzzle

--

--

第八卷 数据库

--

--

一、快速入门 ( 数据库：入门 )

--

https://learnku.com/docs/laravel/6.x/database/5170

--

1、简介

Laravel 能使用原生 SQL、流畅的查询构造器 ( https://learnku.com/docs/laravel/6.0/queries ) 和 Eloquent ORM ( https://learnku.com/docs/laravel/6.0/eloquent ) 在各种数据库后台与数据库进行非常简单的交互。当前 Laravel 支持四种数据库：MySQL、PostgreSQL、SQLite、SQL Server。

(1)、配置

数据库的配置文件放置在 config/database.php 文件中，你可以在此定义所有的数据库连接，并指定默认使用的连接。此文件内提供了大部分 Laravel 能支持的数据库配置示例。

默认情况下，Laravel 的示例 环境配置 ( https://learnku.com/docs/laravel/6.0/configuration#environment-configuration ) 使用了 Laravel Homestead ( https://learnku.com/docs/laravel/6.0/homestead )（这是一种小型虚拟机，能让你很方便地在本地进行 Laravel 的开发）。你可以根据本地数据库的需要修改这个配置。

//

///1/ SQLite 配置

使用类似 touch database/database.sqlite 之类命令创建一个新的 SQLite 数据库之后，可以使用数据库的绝对路径配置环境变量来指向这个新创建的数据库。

DB\_CONNECTION=sqlite

DB\_DATABASE=/absolute/path/to/database.sqlite

如果要开启 SQLite 连接的外键约束，您应该将 foreign\_key\_constraints 添加到 config / database.php 配置文件中。

'sqlite' => [

// ...

'foreign\_key\_constraints' => true,

],

//

///2/ URLs 式配置

通常，数据库连接使用多个配置值，例如 host, database, username, password 等。这些配置值中的每一个都有其相应的环境变量。这意味着在生产服务器上配置数据库连接信息时，需要管理多个环境变量。

一些托管数据库提供程序（如 heroku ）提供单个数据库「URL」，该 url 在单个字符串中包含数据库的所有连接信息。示例数据库 URL 可能如下所示：mysql://root:password@127.0.0.1/forge?charset=UTF-8。

这些 url 通常遵循标准模式约定：driver://username:password@host:port/database?options。

为了方便起见，Laravel 支持这些 URLs ，作为使用多个配置选项配置数据库的替代方法。如果存在 url （或相应的 DATABASE\_URL 环境变量）配置选项，则将使用该选项提取数据库连接和凭据信息。

(2)、读写分离

有时候你希望 SELECT 语句使用一个数据库连接，而 INSERT，UPDATE，和 DELETE 语句使用另一个数据库连接。在 Laravel 中，无论你是使用原生查询，查询构造器，或者是 Eloquent ORM，都能轻松的实现。

为了弄明白读写分离是如何配置的，我们先来看个例子。

'mysql' => [

'read' => [

'host' => [

'192.168.1.1',

'196.168.1.2',

],

],

'write' => [

'host' => [

'196.168.1.3',

],

],

'sticky' => true,

'driver' => 'mysql',

'database' => 'database',

'username' => 'root',

'password' => '',

'charset' => 'utf8mb4',

'collation' => 'utf8mb4\_unicode\_ci',

'prefix' => '',

],

注意在以上的例子中，配置数组中增加了三个键，分别是 read， write 和 sticky。 read 和 write 的键都包含一个键为 host 的数组。而 read 和 write 的其他数据库都在键为 mysql 的数组中。

如果你想重写主数组中的配置，只需要修改 read 和 write 数组即可。所以，这个例子中： 192.168.1.1 和 192.168.1.2 将作为 「读」 连接主机，而 192.168.1.3 将作为 「写」 连接主机。这两个连接会共享 mysql 数组的各项配置，如数据库的凭据（用户名 / 密码），前缀，字符编码等。

//

///1/ sticky 选项

sticky 是一个 可选值，它可用于立即读取在当前请求周期内已写入数据库的记录。若 sticky 选项被启用，并且当前请求周期内执行过 「写」 操作，那么任何 「读」 操作都将使用 「写」 连接。这样可确保同一个请求周期内写入的数据可以被立即读取到，从而避免主从延迟导致数据不一致的问题。不过是否启用它，取决于应用程序的需求。

(3)、使用多个数据库连接

当使用多个数据库连接时，你可以通过 DB Facade 的 connection 方法访问每一个连接。传递给 connection 方法的参数 name 应该是 config/database.php 配置文件中 connections 数组中的一个值。

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

你也可以使用一个连接实例上的 getPdo 方法访问底层的 PDO 实例。

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

--

2、运行原生的 SQL 查询

一旦配置好数据库连接后，便可以使用 DB facade 运行查询。 DB facade 为每种类型的查询提供了方法： select，update，insert，delete 和 statement。

(1)、运行 Select 查询

你可以使用 DB Facade 的 select 方法来运行基础的查询语句。

<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\DB;

use App\Http\Controllers\Controller;

class UserController extends Controller

{

/\*\*

\* 显示应用程序中所有用户的列表

\*

\* @return Response

\*/

public function index()

{

$users = DB::select('select \* from users where active = ?', [1]);

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

}

}

?>

传递给 select 方法的第一个参数就是一个原生的 SQL 查询，而第二个参数则是需要绑定到查询中的参数值。通常，这些值用于约束 where 语句。参数绑定用于防止 SQL 注入。

select 方法将始终返回一个数组，数组中的每个结果都是一个 StdClass 对象，可以像下面这样访问结果值。

foreach ($users as $user) {

echo $user->name;

}

//

///1/ 使用命名绑定

除了使用 ? 表示参数绑定外，你也可以使用命名绑定来执行一个查询。

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

(2)、运行插入语句

可以使用 DB Facade 的 insert 方法来执行 insert 语句。与 select 一样，该方法将原生 SQL 查询作为其第一个参数，并将绑定数据作为第二个参数。

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

(3)、运行更新语句

update 方法用于更新数据库中现有的记录。该方法返回受该语句影响的行数。

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

(4)、运行删除语句

delete 方法用于从数据库中删除记录。与 update 一样，返回受该语句影响的行数。

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

(5)、运行普通语句

有些数据库语句不会有任何返回值。对于这些语句，你可以使用 DB Facade 的 statement 方法来运行。

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

--

3、监听查询事件

如果你想监控程序执行的每一个 SQL 查询，你可以使用 listen 方法。这个方法对于记录查询或调试非常有用。你可以在 服务提供器( https://learnku.com/docs/laravel/6.x/providers ) 中注册你的查询监听器。

<?php

namespace App\Providers;

use Illuminate\Support\Facades\DB;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider

{

/\*\*

\* 注册服务提供器

\*

\* @return void

\*/

public function register()

{

//

}

/\*\*

\* 启动应用服务

\*

\* @return void

\*/

public function boot()

{

DB::listen(function ($query) {

// $query->sql

// $query->bindings

// $query->time

});

}

}

?>

--

4、数据库事务

你可以使用 DB facade 的 transaction 方法在数据库事务中运行一组操作。如果事务的闭包 Closure 中出现一个异常，事务将会回滚。如果事务闭包 Closure 执行成功，事务将自动提交。一旦你使用了 transaction ， 就不再需要担心手动回滚或提交的问题。

DB::transaction(function () {

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

DB::table('posts')->delete();

});

(1)、处理死锁

transaction 方法接受一个可选的第二个参数 ，该参数用来表示事务发生死锁时重复执行的次数。一旦定义的次数尝试完毕，就会抛出一个异常。

DB::transaction(function () {

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

DB::table('posts')->delete();

}, 5);

(2)、手动使用事务

如果你想要手动开始一个事务，并且对回滚和提交能够完全控制，那么你可以使用 DB Facade 的 beginTransaction 方法。

DB::beginTransaction();

你可以使用 rollBack 方法回滚事务。

DB::rollBack();

最后，你可以使用 commit 方法提交事务。

DB::commit();

Tip：DB facade 的事务方法同样适用于 查询构造器 ( https://learnku.com/docs/laravel/6.x/queries ) 和 Eloquent ORM ( https://learnku.com/docs/laravel/6.x/eloquent )。

--

--

二、查询构造器 ( 数据库：查询构造器 )

--

https://learnku.com/docs/laravel/6.x/queries/5171

--

1、简介

Laravel 的数据库查询构造器为创建和运行数据库查询提供了一个方便的接口。它可用于执行应用程序中大部分数据库操作，且可在所有支持的数据库系统上运行。

Laravel 的查询构造器使用 PDO 参数绑定来保护您的应用程序免受 SQL 注入攻击。因此没有必要清理作为绑定传递的字符串。

注意：PDO 不支持绑定列名。因此，不能让用户通过输入来指定查询语句所引用的列名，包括 order by 字段等等。如果必须要允许用户通过选择某些列来进行查询，请始终根据允许列的白名单来校验列名。

--

2、获取结果

(1)、从一个数据表中获取所有行

你可以使用 DB facade 里的 table 方法来开始查询。 table 方法为给定的表返回一个查询构造器实例，允许你在查询上链式调用更多的约束，最后使用 get 方法获取结果。

<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\DB;

use App\Http\Controllers\Controller;

class UserController extends Controller

{

/\*\*

\* 显示所有应用程序的用户列表。

\*

\* @return Response

\*/

public function index()

{

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

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

}

}

?>

该 get 方法返回一个包含 Illuminate\Support\Collection 的结果，其中每个结果都是 PHP StdClass 对象的一个实例。你可以访问字段作为对象的属性来访问每列的值。

foreach ($users as $user) {

echo $user->name;

}

(2)、从数据表中获取单行或单列

如果你只需要从数据表中获取一行数据，你可以使用 first 方法。该方法返回一个 StdClass 对象。

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

echo $user->name;

如果你甚至不需要整行数据，则可以使用 value 方法从记录中获取单个值。该方法将直接返回该字段的值。

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

(3)、获取一列的值

如果你想获取包含单列值的集合，则可以使用 pluck 方法。在下面的例子中，我们将获取角色表中标题的集合。

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

foreach ($titles as $title) {

echo $title;

}

你还可以在返回的集合中指定字段的自定义键名：(会将 roles 表中的 name 字段当做键名，title 字段当做键值返回)。

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

foreach ($roles as $name => $title) {

echo $title;

}

(4)、分块结果

如果你需要处理上千条数据库记录，你可以考虑使用 chunk 方法。该方法一次获取结果集的一小块，并将其传递给 闭包 函数进行处理。该方法在 Artisan 命令 ( https://learnku.com/docs/laravel/6.x/artisan ) 编写数千条处理数据的时候非常有用。例如，我们可以将全部 users 表数据切割成一次处理 100 条记录的一小块。

DB::table('users')->orderBy('id')->chunk(100, function ($users) {

foreach ($users as $user) {

//

}

});

你可以通过在 闭包 中返回 false 来终止继续获取分块结果。

DB::table('users')->orderBy('id')->chunk(100, function ($users) {

// Process the records...

return false;

});

如果要在分块结果时更新数据库记录，则块结果可能会和预计的返回结果不一致。 因此，在分块更新记录时，最好使用 chunkById 方法。 此方法将根据记录的主键自动对结果进行分页。

DB::table('users')->where('active', false)

->chunkById(100, function ($users) {

foreach ($users as $user) {

DB::table('users')

->where('id', $user->id)

->update(['active' => true]);

}

});

{提示} 在块的回调里面更新或删除记录时，对主键或外键的任何更改都可能影响块查询。 这可能会导致记录没有包含在分块结果中。

(5)、聚合

查询构造器还提供了各种聚合方法，比如 count, max，min， avg，还有 sum。你可以在构造查询后调用任何方法。

$users = DB::table('users')->count();

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

当然，你也可以将这些聚合方法与其他的查询语句相结合。

$price = DB::table('orders')

->where('finalized', 1)

->avg('price');

//

///1/ 判断记录是否存在

除了通过 count 方法可以确定查询条件的结果是否存在之外，还可以使用 exists 和 doesntExist 方法。

return DB::table('orders')->where('finalized', 1)->exists();

return DB::table('orders')->where('finalized', 1)->doesntExist();

--

3、查询 (Selects)

(1)、指定一个 Select 语句

当然你可能并不总是希望从数据库表中获取所有列。使用 select 方法，你可以自定义一个 select 查询语句来查询指定的字段。

$users = DB::table('users')->select('name', 'email as user\_email')->get();

distinct 方法会强制让查询返回的结果不重复。

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

如果你已经有了一个查询构造器实例，并且希望在现有的查询语句中加入一个字段，那么你可以使用 addSelect 方法。

$query = DB::table('users')->select('name');

$users = $query->addSelect('age')->get();

--

4、原生表达式

有时候你可能需要在查询中使用原生表达式。你可以使用 DB::raw 创建一个原生表达式。

$users = DB::table('users')

->select(DB::raw('count(\*) as user\_count, status'))

->where('status', '<>', 1)

->groupBy('status')

->get();

{提示} 原生表达式将会被当做字符串注入到查询中，因此你应该小心使用，避免创建 SQL 注入的漏洞。

(1)、原生方法

可以使用以下方法代替 DB::raw，将原生表达式插入查询的各个部分。

//

///1/ selectRaw

selectRaw 方法可以代替 select(DB::raw(...))。该方法的第二个参数是可选项，值是一个绑定参数的数组。

$orders = DB::table('orders')

->selectRaw('price \* ? as price\_with\_tax', [1.0825])

->get();

//

///2/ whereRaw / orWhereRaw

whereRaw 和 orWhereRaw 方法将原生的 where 注入到你的查询中。这两个方法的第二个参数还是可选项，值还是绑定参数的数组。

$orders = DB::table('orders')

->whereRaw('price > IF(state = "TX", ?, 100)', [200])

->get();

//

///3/ havingRaw / orHavingRaw

havingRaw 和 orHavingRaw 方法可以用于将原生字符串设置为 having 语句的值。

$orders = DB::table('orders')

->select('department', DB::raw('SUM(price) as total\_sales'))

->groupBy('department')

->havingRaw('SUM(price) > ?', [2500])

->get();

//

///4/ orderByRaw

orderByRaw 方法可用于将原生字符串设置为 order by 子句的值。

$orders = DB::table('orders')

->orderByRaw('updated\_at - created\_at DESC')

->get();

--

5、Joins

(1)、Inner Join 语句

查询构造器也可以编写 join 方法。若要执行基本的「内链接」，你可以在查询构造器实例上使用 join 方法。传递给 join 方法的第一个参数是你需要连接的表的名称，而其他参数则使用指定连接的字段约束。你还可以在单个查询中连接多个数据表。

$users = DB::table('users')

->join('contacts', 'users.id', '=', 'contacts.user\_id')

->join('orders', 'users.id', '=', 'orders.user\_id')

->select('users.\*', 'contacts.phone', 'orders.price')

->get();

(2)、Left Join / Right Join 语句

如果你想使用 「左连接」或者 「右连接」代替「内连接」 ，可以使用 leftJoin 或者 rightJoin 方法。这两个方法与 join 方法用法相同。

$users = DB::table('users')

->leftJoin('posts', 'users.id', '=', 'posts.user\_id')

->get();

$users = DB::table('users')

->rightJoin('posts', 'users.id', '=', 'posts.user\_id')

->get();

(3)、Cross Join 语句

使用 crossJoin 方法和你想要连接的表名做 「交叉连接」。交叉连接在第一个表和被连接的表之间会生成笛卡尔积。

$users = DB::table('sizes')

->crossJoin('colours')

->get();

(4)、高级 Join 语句

你可以指定更高级的 join 语句。比如传递一个 闭包 作为 join 方法的第二个参数。此 闭包 接收一个 JoinClause 对象，从而指定 join 语句中指定的约束。

DB::table('users')

->join('contacts', function ($join) {

$join->on('users.id', '=', 'contacts.user\_id')->orOn(...);

})

->get();

如果你想要在连接上使用「where」 风格的语句，你可以在连接上使用 where 和 orWhere 方法。这些方法会将列和值进行比较，而不是列和列进行比较。

DB::table('users')

->join('contacts', function ($join) {

$join->on('users.id', '=', 'contacts.user\_id')

->where('contacts.user\_id', '>', 5);

})

->get();

(5)、子连接查询

你可以使用 joinSub，leftJoinSub 和 rightJoinSub 方法关联一个查询作为子查询。他们每一种方法都会接收三个参数：子查询，表别名和定义关联字段的闭包。

$latestPosts = DB::table('posts')

->select('user\_id', DB::raw('MAX(created\_at) as last\_post\_created\_at'))

->where('is\_published', true)

->groupBy('user\_id');

$users = DB::table('users')

->joinSub($latestPosts, 'latest\_posts', function ($join) {

$join->on('users.id', '=', 'latest\_posts.user\_id');

})->get();

--

6、Unions

查询构造器还提供了将两个查询 「联合」 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();

{提示} 你也可以使用 unionAll 方法，用法和 union 方法一样。

--

7、Where 语句

(1)、简单的 Where 语句

在构造 where 查询实例中，你可以使用 where 方法。调用 where 最基本的方式是需要传递三个参数：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运算符，第三个是该列要比较的值。

例如，下面是一个要验证 「votes」 字段的值等于 100 的查询。

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

为了方便，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，你也可以使用其他的运算符来编写 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();

你还可以传递条件数组到 where 函数中。

$users = DB::table('users')->where([

['status', '=', '1'],

['subscribed', '<>', '1'],

])->get();

(2)、Or 语句

你可以一起链式调用 where 约束，也可以在查询中添加 or 字句。 orWhere 方法和 where 方法接收的参数一样。

$users = DB::table('users')

->where('votes', '>', 100)

->orWhere('name', 'John')

->get();

(3)、其他 Where 语句

//

///1/ whereBetween / orWhereBetween

whereBetween 方法验证字段值是否在给定的两个值之间。

$users = DB::table('users')

->whereBetween('votes', [1, 100])

->get();

//

///2/ whereNotBetween / orWhereNotBetween

whereNotBetween 方法验证字段值是否在给定的两个值之外。

$users = DB::table('users')

->whereNotBetween('votes', [1, 100])

->get();

//

///3/ whereIn / whereNotIn / orWhereIn / orWhereNotIn

// whereIn 方法验证字段的值必须存在指定的数组里。

$users = DB::table('users')

->whereIn('id', [1, 2, 3])

->get();

// whereNotIn 方法验证字段的值必须不存在于指定的数组里。

$users = DB::table('users')

->whereNotIn('id', [1, 2, 3])

->get();

//

///4/ whereNull / whereNotNull / orWhereNull / orWhereNotNull

// whereNull 方法验证指定的字段必须是 NULL。

$users = DB::table('users')

->whereNull('updated\_at')

->get();

// whereNotNull 方法验证指定的字段必须不是 NULL。

$users = DB::table('users')

->whereNotNull('updated\_at')

->get();

//

///5/ whereDate / whereMonth / whereDay / whereYear / whereTime

// whereDate 方法用于比较字段值与给定的日期。

$users = DB::table('users')

->whereDate('created\_at', '2016-12-31')

->get();

// whereMonth 方法用于比较字段值与一年中指定的月份。

$users = DB::table('users')

->whereMonth('created\_at', '12')

->get();

// whereDay 方法用于比较字段值与一月中指定的日期。

$users = DB::table('users')

->whereDay('created\_at', '31')

->get();

// whereYear 方法用于比较字段值与指定的年份。

$users = DB::table('users')

->whereYear('created\_at', '2016')

->get();

// whereTime 方法用于比较字段值与指定的时间（时分秒）。

$users = DB::table('users')

->whereTime('created\_at', '=', '11:20:45')

->get();

//

///6/ whereColumn / orWhereColumn

// whereColumn 方法用于比较两个字段的值 是否相等。

$users = DB::table('users')

->whereColumn('first\_name', 'last\_name')

->get();

你也可以传入一个比较运算符。

$users = DB::table('users')

->whereColumn('updated\_at', '>', 'created\_at')

->get();

whereColumn 方法也可以传递数组，用 and 运算符链接。

$users = DB::table('users')

->whereColumn([

['first\_name', '=', 'last\_name'],

['updated\_at', '>', 'created\_at'],

])->get();

(4)、参数分组

有时候你需要创建更高级的 where 子句，例如「where exists」或者嵌套的参数分组。 Laravel 的查询构造器也能够处理这些。下面，让我们看一个在括号中进行分组约束的例子。

$users = DB::table('users')

->where('name', '=', 'John')

->where(function ($query) {

$query->where('votes', '>', 100)

->orWhere('title', '=', 'Admin');

})

->get();

你可以看到，通过一个 Closure 写入 where 方法构建一个查询构造器 来约束一个分组。 这个 Closure 接收一个查询实例，你可以使用这个实例来设置应该包含的约束。上面的例子将生成以下 SQL。

select \* from users where name = 'John' and (votes > 100 or title = 'Admin')

{提示} 你应该用 orWhere 调用这个分组，以避免应用全局作用出现意外。

(5)、Where Exists 语法

whereExists 方法允许你使用 where exists SQL 语句。 whereExists 方法接收一个 Closure 参数，该 whereExists 方法接受一个 Closure 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 字句中查询。

$users = DB::table('users')

->whereExists(function ($query) {

$query->select(DB::raw(1))

->from('orders')

->whereRaw('orders.user\_id = users.id');

})

->get();

上述查询将产生如下的 SQL 语句。

select \* from users

where exists (

select 1 from orders where orders.user\_id = users.id

)

(6)、JSON Where 语法 (JSON Where 语句)

Laravel 也支持查询 JSON 类型的字段（仅在对 JSON 类型支持的数据库上）。目前，本特性仅支持 MySQL 5.7、PostgreSQL、SQL Server 2016 以及 SQLite 3.9.0 (with the JSON1 extension ( https://www.sqlite.org/json1.html ) )。使用 -> 操作符查询 JSON 数据。

$users = DB::table('users')

->where('options->language', 'en')

->get();

$users = DB::table('users')

->where('preferences->dining->meal', 'salad')

->get();

你也可以使用 whereJsonContains 来查询 JSON 数组（不支持 SQLite）。

$users = DB::table('users')

->whereJsonContains('options->languages', 'en')

->get();

MySQL 和 PostgreSQL 的 whereJsonContains 可以支持多个值。

$users = DB::table('users')

->whereJsonContains('options->languages', ['en', 'de'])

->get();

你可以使用 whereJsonLength 来查询 JSON 数组的长度。

$users = DB::table('users')

->whereJsonLength('options->languages', 0)

->get();

$users = DB::table('users')

->whereJsonLength('options->languages', '>', 1)

->get();

--

8、Ordering, Grouping, Limit, & Offset

(1)、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，第二个参数控制排序的方向，可以是 asc 或 desc。

$users = DB::table('users')

->orderBy('name', 'desc')

->get();

(2)、latest / oldest

latest 和 oldest 方法可以使你轻松地通过日期排序。它默认使用 created\_at 列作为排序依据。当然，你也可以传递自定义的列名。

$user = DB::table('users')

->latest()

->first();

(3)、inRandomOrder

inRandomOrder 方法被用来将结果随机排序。例如，你可以使用此方法随机找到一个用户。

$randomUser = DB::table('users')

->inRandomOrder()

->first();

(4)、groupBy / having

groupBy 和 having 方法可以将结果分组。 having 方法的使用与 where 方法十分相似。

$users = DB::table('users')

->groupBy('account\_id')

->having('account\_id', '>', 100)

->get();

你可以向 groupBy 方法传递多个参数。

$users = DB::table('users')

->groupBy('first\_name', 'status')

->having('account\_id', '>', 100)

->get();

对于更高级的 having 语法，参见 havingRaw 方法。

(5)、skip / take

要限制结果的返回数量，或跳过指定数量的结果，你可以使用 skip 和 take 方法。

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

或者你也可以使用 limit 和 offset 方法。

$users = DB::table('users')

->offset(10)

->limit(5)

->get();

--

9、条件语句

有时候你可能想要子句只适用于某个情况为真时才执行查询。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。 你可以通过使用 when 方法。

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

$users = DB::table('users')

->when($role, function ($query, $role) {

return $query->where('role\_id', $role);

})

->get();

when 方法只有在第一个参数为 true 的时候才执行给的闭包。如果第一个参数为 false ，那么这个闭包将不会被执行。你也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。为了说明如何使用这个特性，我们来配置一个查询的默认排序。

$sortBy = null;

$users = DB::table('users')

->when($sortBy, function ($query, $sortBy) {

return $query->orderBy($sortBy);

}, function ($query) {

return $query->orderBy('name');

})

->get();

--

10、插入

查询构造器还提供了 insert 方法用于插入记录到数据库中。 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]

);

你甚至可以将数组传递给 insert 方法，将多个记录插入到表中。

DB::table('users')->insert([

['email' => 'taylor@example.com', 'votes' => 0],

['email' => 'dayle@example.com', 'votes' => 0]

]);

insertOrIgnore 方法用于忽略重复插入记录到数据库的错误。

DB::table('users')->insertOrIgnore([

['id' => 1, 'email' => 'taylor@example.com'],

['id' => 2, 'email' => 'dayle@example.com']

]);

(1)、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]

);

注意：当使用 PostgreSQL 时，insertGetId 方法将默认把 id 作为自动递增字段的名称。如果你要从其他「序列」来获取 ID ，则可以将字段名称作为第二个参数传递给 insertGetId 方法。

--

11、更新

当然， 除了插入记录到数据库中，查询构造器也可以通过 update 方法更新已有的记录。 update 方法和 insert 方法一样，接受包含要更新的字段及值的数组。你可以通过 where 子句对 update 查询进行约束。

$affected = DB::table('users')

->where('id', 1)

->update(['votes' => 1]);

//

//// 更新或者新增

有时您可能希望更新数据库中的现有记录，或者如果不存在匹配记录则创建它。 在这种情况下，可以使用 updateOrInsert 方法。 updateOrInsert 方法接受两个参数：一个用于查找记录的条件数组，以及一个包含要更新该记录的键值对数组。

updateOrInsert 方法将首先尝试使用第一个参数的键和值对来查找匹配的数据库记录。 如果记录存在，则使用第二个参数中的值去更新记录。 如果找不到记录，将插入一个新记录，更新的数据是两个数组的集合。

DB::table('users')

->updateOrInsert(

['email' => 'john@example.com', 'name' => 'John'],

['votes' => '2']

);

(1)、更新 JSON ( 更新 JSON 字段 )

更新 JSON 字段时，你可以使用 -> 语法访问 JSON 对象中相应的值，此操作只能支持 MySQL 5.7+ 和 PostgreSQL 9.5+。

$affected = DB::table('users')

->where('id', 1)

->update(['options->enabled' => true]);

(2)、自增和自减 ( 自增 & 自减 )

查询构造器还为给定字段的递增或递减提供了方便的方法。此方法提供了一个比手动编写 update 语句更具表达力且更精练的接口。

这两种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，用于控制列递增或递减的量。

DB::table('users')->increment('votes');

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

DB::table('users')->decrement('votes');

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

你也可以在操作过程中指定要更新的字段。

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

--

12、删除

查询构造器也可以使用 delete 方法从表中删除记录。 在使用 delete 前，可以添加 where 子句来约束 delete 语法。

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

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

如果你需要清空表，你可以使用 truncate 方法，它将删除所有行，并重置自增 ID 为零。

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

--

13、悲观锁

查询构造器也包含一些可以帮助你在 select 语法上实现「悲观锁定」的函数。若想在查询中实现一个「共享锁」， 你可以使用 sharedLock 方法。 共享锁可防止选中的数据列被篡改，直到事务被提交为止。

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

或者，你可以使用 lockForUpdate 方法。使用 「update」锁可避免行被其它共享锁修改或选取。

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

--

14、调试

你可以使用 dd 或者 dump 方法输出查询结果或者 SQL 语句。 使用 dd 方法可以显示调试信息，然后停止执行请求。 dump 方法同样可以显示调试信息，但是不会停止执行请求。

DB::table('users')->where('votes', '>', 100)->dd();

DB::table('users')->where('votes', '>', 100)->dump();

--

--

三、分页 ( 数据库：分页 )

--

https://learnku.com/docs/laravel/6.x/pagination/5172

--

1、简介

在其他框架中，分页可能是非常痛苦的。Laravel 的分页器将 查询构造器 ( https://learnku.com/docs/laravel/6.x/queries ) 和 Eloquent ORM ( https://learnku.com/docs/laravel/6.x/eloquent ) 结合起来，提供了方便、易用的数据库结果集分页。通过分页器生成的 HTML 兼容 Bootstrap CSS 框架 ( https://getbootstrap.com/ )。

--

2、基本用法

(1)、查询构造器分页

数据分页有多种方法。最简单的是使用 查询构造器 ( https://learnku.com/docs/laravel/6.x/queries ) 或 Eloquent query ( https://learnku.com/docs/laravel/6.x/eloquent ) 的 paginate 方法。paginate 方法根据用户浏览的当前页码，自动设置恰当的偏移量 offset 和限制数 limit。默认情况下，HTTP 请求中的 page 查询参数值被当作当前页的页码。 Laravel 会自动检测该值，并自动将其插入到分页器生成的链接中。

在下面的例子中，传递给 paginate 方法的唯一参数是你想要在每页显示的记录数。在此例中，我们指定要在每页显示 15 条数据。

<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\DB;

use App\Http\Controllers\Controller;

class UserController extends Controller

{

/\*\*

\* 显示应用程序中的所有用户.

\*

\* @return Response

\*/

public function index()

{

$users = DB::table('users')->paginate(15);

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

}

}

?>

注意：目前，Laravel 无法高效地执行使用 groupBy 语句的分页操作。如果你需要对使用了 groupBy 的结果集分页，建议你手动查询数据库并创建分页。

//

///1/ 简单分页

如果你只需要在分页视图中简单地显示「下一页」和「上一页」的链接，你可以使用 simplePaginate 方法来执行更高效地查询。这在数据量很大且不需要在渲染视图时显示每页的页码时非常有用。

$users = DB::table('users')->simplePaginate(15);

(2)、Eloquent 分页

你还可以对 Eloquent ( https://learnku.com/docs/laravel/6.x/eloquent ) 查询结果进行分页。在下面的例子中，我们将 User 模型按每页 15 条分页。如你所见，其语法与查询构造器分页基本相同。

$users = App\User::paginate(15);

你还可以在调用 paginate 之前，在查询中设置诸如 where 从句的其它约束。

$users = User::where('votes', '>', 100)->paginate(15);

你还可以在 Eloquent 分页中使用 simplePaginate 方法。

$users = User::where('votes', '>', 100)->simplePaginate(15);

(3)、手动创建分页

有时你可能希望手动创建分页，并传递一个数组集给它。可以通过创建 Illuminate\Pagination\Paginator 或 Illuminate\Pagination\LengthAwarePaginator 实例来实现，这取决于你的需要。

Paginator 类不需要知道结果集的总数；然而，这样一来，这个类就没办法获知最后一页的索引。LengthAwarePaginator 接受和 Paginator 几乎相同的参数；不过，它会计算结果集的总数。

换句话说，Paginator 相当于查询构造器或 Eloquent 的 simplePaginate 方法，而 LengthAwarePaginator 相当于 paginate 方法。

{注意} 在手动创建分页实例时，需要人为「切割」传递给分页实例的结果数组。如果你对此没有把握，请参考 PHP 的 array\_slice ( https://secure.php.net/manual/en/function.array-slice.php ) 方法。

--

3、显示分页结果 ( 显示结果集 )

(1)、将结果转换为 JSON

调用 paginate 方法时，你会得到 Illuminate\Pagination\LengthAwarePaginator 实例，调用 simplePaginate 方法时，你会得到 Illuminate\Pagination\Paginator 实例。这些对象提供了几个描述结果集的方法。除了这些辅助方法，分页器的实例是迭代器，可以像数组一样循环。所以，当你得到结果后，你可以使用 Blade ( https://learnku.com/docs/laravel/6.x/blade ) 模版来显示数据、渲染分页链接。

<div class="container">

@foreach ($users as $user)

{{ $user->name }}

@endforeach

</div>

{{ $users->links() }}

links 方法会渲染结果集中剩余页面的链接。每个链接都已经包含了 page URL 变量。记住， links 方法生成的 HTML 兼容 Bootstrap CSS 框架 ( https://getbootstrap.com/ )。

//

///1/ 自定义分页器 URI

withPath 方法允许你在生成分页链接时自定义 URI。例如，如果你想生成像 http://example.com/custom/url?page=N 的分页链接，你应该传递 custom/url 参数给 withPath 方法。

Route::get('users', function () {

$users = App\User::paginate(15);

$users->withPath('custom/url');

});

//

///2/ 附加参数到分页链接

你可以使用 appends 方法向分页链接中添加查询参数。例如，要在每个分页链接中添加 sort=votes ，你应该这样调用 appends。

{{ $users->appends(['sort' => 'votes'])->links() }}

如果你希望向分页器的 URL 添加「哈希片段」，你可以使用 fragment 方法。要在每页链接中添加 #foo ，应该这样调用 fragment 方法。

{{ $users->fragment('foo')->links() }}

//

///3/ 调整分页链接窗口

你可以控制在分页器 URL「窗口」的每一侧显示多少个附加链接。默认情况下，主分页链接的每侧显示三个链接。但是，你可以使用 onEachSide 方法来控制这个值。

{{ $users->onEachSide(5)->links() }}

//

///4/ 将结果转换为 JSON

Laravel 分页器类实现了 Illuminate\Contracts\Support\Jsonable 接口契约，提供了 toJson 方法，所以可以方便的将分页结果转换为 JSON。你也可以通过来自路由或控制器的操作返回分页实例来将其转换为 JSON。

Route::get('users', function () {

return App\User::paginate();

});

来自分页器的 JSON 包括诸如 total，current\_page，last\_page 等元数据信息。实际结果对象将通过 JSON 数组的 data 键提供。以下是通过自路由中分页器实例的方式创建 JSON 的例子。

{

"total": 50,

"per\_page": 15,

"current\_page": 1,

"last\_page": 4,

"first\_page\_url": "http://laravel.app?page=1",

"last\_page\_url": "http://laravel.app?page=4",

"next\_page\_url": "http://laravel.app?page=2",

"prev\_page\_url": null,

"path": "http://laravel.app",

"from": 1,

"to": 15,

"data":[

{

// 结果对象集

},

{

// 结果对象集

}

]

}

--

4、自定义分页视图

默认情况下，渲染视图来显示分页链接是与 Bootstrap CSS 框架兼容的。然而，如果你不想使用 Bootstrap，你可以自由的选择你自己定制的视图来渲染这些链接。在调用分页器实例的 links 方法时，将视图名称作为传递给它的第一个参数。

{{ $paginator->links('view.name') }}

// 传递数据给视图...

{{ $paginator->links('view.name', ['foo' => 'bar']) }}

但是，自定义分页视图最简单的方法是使用 vendor:publish 命令将它们输出到 resources/views/vendor 目录。

// 发布分页视图

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

这个命令将在 resources/views/vendor/pagination 目录中放置这些视图。内置在该目录下的 bootstrap-4.blade.php 文件提供了默认的分页视图。你可以编辑这个文件来修改分页的 HTML。

如果你想要定义不同的文件作为默认的分页视图，你需要在 AppServiceProvider 中使用分页器的 defaultView 和 defaultSimpleView 方法。

use Illuminate\Pagination\Paginator;

public function boot()

{

Paginator::defaultView('view-name');

Paginator::defaultSimpleView('view-name');

}

--

5、分页器实例方法

每个分页器实例提供了如下方法来获取附加的分页信息。

(方法/描述)

$results->count() 获取当前页数据的数量。

$results->currentPage() 获取当前页页码。

$results->firstItem() 获取结果集中第一条数据的结果编号。

$results->getOptions() 获取分页器选项。

$results->getUrlRange($start, $end) 创建分页 URL 的范围。

$results->hasMorePages() 是否有多页。

$results->items() 获取当前页的所有项。

$results->lastItem() 获取结果集中最后一条数据的结果编号。

$results->lastPage() 获取最后一页的页码。(在 simplePaginate 无效)。

$results->nextPageUrl() 获取下一页的 URL。

$results->onFirstPage() 当前页是否为第一页。

$results->perPage() 每页的数据条数。

$results->previousPageUrl() 获取前一页的 URL。

$results->total() 数据总数（在 simplePaginate 无效）。

$results->url($page) 获取指定页的 URL。

--

--

四、数据库迁移 (数据库：迁移)

--

https://learnku.com/docs/laravel/6.x/migrations/5173

--

1、介绍

迁移就像是数据库的版本控制器，让你的团队更容易修改和共享程序的数据库结构。迁移通常配合 Laravel 的结构生成器，能更容易的生成应用程序的数据库结构。如果你曾经让一个团队成员在他本地的数据库结构中手动的添加了字段，那么你将面对解决数据库迁移的问题。

Laravel 的 Schema 门面 ( https://learnku.com/docs/laravel/6.x/facades ) 提供数据库无关的支持，用于在所有 Laravel 支持的数据库系统中创建和操作表。

2、创建迁移 (生成迁移)

使用 make:migration Artisan ( https://learnku.com/docs/laravel/6.x/artisan ) 命令来创建迁移。

php artisan make:migration create\_users\_table

新创建的迁移会放在你的 database/migrations 目录。每个迁移的文件名都包含一个时间戳来让 Laravel 确认迁移的顺序。

--table 和 --create 选项也可用于确定表的名称以及是否在迁移中创建新的数据表。这些选项用指定的迁移模板预先填充指定的数据表。

php artisan make:migration create\_users\_table --create=users

php artisan make:migration add\_votes\_to\_users\_table --table=users

如果你想自定义生成迁移文件的个人存放路径，你可以在你执行 make:migration 命令行的时候使用 --path 选项。给定的路径应该是相对于你的应用程序的基本路径。

--

3、迁移结构

一个迁移类包含两个方法： up 和 down。up 方法是用于新增数据库的数据表、字段或者索引的，而 down 方法应该与 up 方法的执行操作相反。

在这两种方法中，你可以使用 Laravel 的结构生成器以表达式方式创建和修改表。要了解 Schema 生成器上可用的所有方法，请查看其文档( https://learnku.com/docs/laravel/6.x/migrations/5173#creating-tables )。例如，这是一个创建 flights 数据表的例子。

<?php

use Illuminate\Support\Facades\Schema;

use Illuminate\Database\Schema\Blueprint;

use Illuminate\Database\Migrations\Migration;

class CreateFlightsTable extends Migration

{

/\*\*

\* Run the migrations.

\*

\* @return void

\*/

public function up()

{

Schema::create('flights', function (Blueprint $table) {

$table->bigIncrements('id');

$table->string('name');

$table->string('airline');

$table->timestamps();

});

}

/\*\*

\* Reverse the migrations.

\*

\* @return void

\*/

public function down()

{

Schema::drop('flights');

}

}

?>

--

4、运行迁移

请执行 Artisan 命令 migrate 来执行所有未执行的迁移。

php artisan migrate

{注意} 如果你正在使用 Homestead 虚拟机( https://learnku.com/docs/laravel/6.x/homestead ), 你应该在虚拟机中执行这个命令。

//

///1/ 在生产环境中强制执行迁移

大多数迁移操作都是破坏性的，这意味着也许会丢失数据。为了防止有人在生产环境数据中运行这些命令，在执行这些命令之前将提示你进行确认。如果要强制迁移命令在没有提示的情况下运行，请使用 --force 参数。

// 强制执行迁移

php artisan migrate --force

(1)、回滚迁移

要回滚到最后一次操作，你可以用下 rollback 命令。此命令会回滚到最后一批的迁移，这可能会包含多个迁移文件。

// 回滚迁移

php artisan migrate:rollback

//

通过向 rollback 命令加上 step 参数，可以回滚指定数量的迁移。例如，以下命令将回滚最后五个迁移：

// 回归步数

php artisan migrate:rollback --step=5

//

migrate:reset 命令将会滚回你应用程序所有的迁移。

// 重置迁移

php artisan migrate:reset

(2)、使用单个命令来执行迁移和回滚

migrate:refresh 命令将会在回滚你所有的迁移后执行 migrate 命令。这个命令可以高效的重新创建你的整个数据库。

php artisan migrate:refresh

// 刷新数据库并执行数据库填充

php artisan migrate:refresh --seed

通过向 refresh 命令加上 step 参数，可以回滚并重新迁移指定数量的迁移。例如，以下命令将回滚并重新迁移最后五个迁移。

php artisan migrate:refresh --step=5

(3)、删除所有数据表和迁移

migrate:fresh 命名将会删除数据中所有的数据表并在之后执行 migrate 命令。

php artisan migrate:fresh

php artisan migrate:fresh --seed

--

5、数据表

(1)、创建数据表

请使用 Schema 门面中的 create 方法来在数据库中创建一个新表。create 可以接受两个参数。一个是数据表的名字，另一个是 Closure 可以在定义一个新数据表的时候接受 Blueprint 对象。

Schema::create('users', function (Blueprint $table) {

$table->bigIncrements('id');

});

在创建数据表时，你可以使用结构生成器的任何 字段方法 ( https://learnku.com/docs/laravel/6.x/migrations/5173#creating-columns ) 去定义数据表的字段。

//

///1/ 检查表 / 列是否存在

你可以使用 hasTable 和 hasColumn 方法轻松的检查数据表和字段是否存在。

if (Schema::hasTable('users')) {

//

}

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

//

}

//

///2/ 数据库连接 & 数据表选项

如果你想要对非默认连接的数据库连接执行结构操作，请使用 connection 方法。

Schema::connection('foo')->create('users', function (Blueprint $table) {

$table->bigIncrements('id');

});

你可以在结构生成器上使用以下命令定义表的选项(命令/描述)。

$table->engine = 'InnoDB'; 指定表存储引擎 (MySQL).

$table->charset = 'utf8'; 指定表的默认字符编码 (MySQL).

$table->collation = 'utf8\_unicode\_ci'; 指定表的默认排序格式 (MySQL).

$table->temporary(); 创建临时数据表 (except SQL Server).

(2)、重命名 / 删除 数据表

重命名一个存在的数据库表，请使用 rename 方法。

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

//

删除一个存在的数据表，你可以使用 drop 或者 dropIfExists 方法。

Schema::drop('users');

Schema::dropIfExists('users');

//

///1/ 重命名有外键约束的数据表

在你重命名数据表之前，你应该验证表上的任何外键约束在迁移文件中是否具有确切的名称，而不是让 Laravel 按照约定来分配一个名称。否则，外键约束名将引用旧表名。

--

6、字段

(1)、创建字段

在 Schema 门面中可以使用 table 方法来更新已存在的数据表。和 create 方法一样，table 方法接受两个参数：一个是数据表的名字，另一个是 Closure 可以在新增一个数据表中字段的时候接受 Blueprint 对象。

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

$table->string('email');

});

//

///1/ 可用的字段类型

数据库结构生成器包含构建表时可以指定的各种字段类型(命令/描述)。

$table->bigIncrements('id'); 递增 ID（主键），相当于「UNSIGNED BIG INTEGER」

$table->bigInteger('votes'); 相当于 BIGINT

$table->binary('data'); 相当于 BLOB

$table->boolean('confirmed'); 相当于 BOOLEAN

$table->char('name', 100); 相当于带有长度的 CHAR

$table->date('created\_at'); 相当于 DATE

$table->dateTime('created\_at'); 相当于 DATETIME

$table->dateTimeTz('created\_at'); 相当于带时区 DATETIME

$table->decimal('amount', 8, 2); 相当于带有精度与基数 DECIMAL

$table->double('amount', 8, 2); 相当于带有精度与基数 DOUBLE

$table->enum('level', ['easy', 'hard']); 相当于 ENUM

$table->float('amount', 8, 2); 相当于带有精度与基数 FLOAT

$table->geometry('positions'); 相当于 GEOMETRY

$table->geometryCollection('positions'); 相当于 GEOMETRYCOLLECTION

$table->increments('id'); 递增的 ID (主键)，相当于「UNSIGNED INTEGER」

$table->integer('votes'); 相当于 INTEGER

$table->ipAddress('visitor'); 相当于 IP 地址

$table->json('options'); 相当于 JSON

$table->jsonb('options'); 相当于 JSONB

$table->lineString('positions'); 相当于 LINESTRING

$table->longText('description'); 相当于 LONGTEXT

$table->macAddress('device'); 相当于 MAC 地址

$table->mediumIncrements('id'); 递增 ID (主键) ，相当于「UNSIGNED MEDIUM INTEGER」

$table->mediumInteger('votes'); 相当于 MEDIUMINT

$table->mediumText('description'); 相当于 MEDIUMTEXT

$table->morphs('taggable'); 相当于加入递增的 taggable\_id 与字符串 taggable\_type

$table->uuidMorphs('taggable'); 相当于加入 taggable\_id 与字符串 taggable\_typeUUID 列。

$table->multiLineString('positions'); 相当于 MULTILINESTRING

$table->multiPoint('positions'); 相当于 MULTIPOINT

$table->multiPolygon('positions'); 相当于 MULTIPOLYGON

$table->nullableMorphs('taggable'); 相当于可空版本的 morphs () 字段

$table->nullableUuidMorphs('taggable'); 相当于可空版本的 uuidMorphs() 字段

$table->nullableTimestamps(); 相当于可空版本的 timestamps() 字段

$table->point('position'); 相当于 POINT

$table->polygon('positions'); 相当于 POLYGON

$table->rememberToken(); 相当于可空版本的 VARCHAR (100) 的 remember\_token 字段

$table->set('flavors', ['strawberry', 'vanilla']); 相当于 SET

$table->smallIncrements('id'); 递增 ID（主键），相当于「UNSIGNED SMALLINT」

$table->smallInteger('votes'); 相当于 SMALLINT

$table->softDeletes(); 相当于为软删除添加一个可空的 deleted\_at 字段

$table->softDeletesTz(); 相当于为软删除添加一个可空的 带时区的 deleted\_at 字段

$table->string('name', 100); 相当于带长度的 VARCHAR

$table->text('description'); 相当于 TEXT

$table->time('sunrise'); 相当于 TIME

$table->timeTz('sunrise'); 相当于带时区的 TIME

$table->timestamp('added\_on'); 相当于 TIMESTAMP

$table->timestampTz('added\_on'); 相当于带时区的 TIMESTAMP

$table->timestamps(); 相当于可空的 created\_at 和 updated\_at TIMESTAMP

$table->timestampsTz(); 相当于可空且带时区的 created\_at 和 updated\_at TIMESTAMP

$table->tinyIncrements('id'); 相当于自动递增 UNSIGNED TINYINT

$table->tinyInteger('votes'); 相当于 TINYINT

$table->unsignedBigInteger('votes'); 相当于 Unsigned BIGINT

$table->unsignedDecimal('amount', 8, 2); 相当于带有精度和基数的 UNSIGNED DECIMAL

$table->unsignedInteger('votes'); 相当于 Unsigned INT

$table->unsignedMediumInteger('votes'); 相当于 Unsigned MEDIUMINT

$table->unsignedSmallInteger('votes'); 相当于 Unsigned SMALLINT

$table->unsignedTinyInteger('votes'); 相当于 Unsigned TINYINT

$table->uuid('id'); 相当于 UUID

$table->year('birth\_year'); 相当于 YEAR

(2)、字段修饰

除了上述列出的字段类型之外，还有几个可以在添加字段到数据库表时使用的 “修饰符”。例如，如果要把字段设置为 “可空 "， 你可以使用 nullable 方法。

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

$table->string('email')->nullable();

});

以下是所有可用的字段修饰符的列表。此列表不包括 索引修饰符 ( https://learnku.com/docs/laravel/6.x/migrations/5173#creating-indexes ) (命令/描述)。

->after('column') 将此字段放置在其它字段 "之后" (MySQL)

->autoIncrement() 将 INTEGER 类型的字段设置为自动递增的主键

->charset('utf8') 指定一个字符集 (MySQL)

->collation('utf8\_unicode\_ci') 指定列的排序规则 (MySQL/SQL Server)

->comment('my comment') 为字段增加注释 (MySQL)

->default($value) 为字段指定 "默认" 值

->first() 将此字段放置在数据表的 "首位" (MySQL)

->nullable($value = true) 此字段允许写入 NULL 值（默认情况下）

->storedAs($expression) 创建一个存储生成的字段 (MySQL)

->unsigned() 设置 INTEGER 类型的字段为 UNSIGNED (MySQL)

->useCurrent() 将 TIMESTAMP 类型的字段设置为使用 CURRENT\_TIMESTAMP 作为默认值

->virtualAs($expression) 创建一个虚拟生成的字段 (MySQL)

->generatedAs($expression) 使用指定的序列生成标识列（PostgreSQL）

->always() 定义序列值优先于标识列的输入 (PostgreSQL)

(3)、修改字段

//

///1/ 先决条件

在修改字段之前，请确保将 doctrine/dbal 依赖添加到 composer.json 文件中。Doctrine DBAL 库用于确定字段的当前状态， 并创建对该字段进行指定调整所需的 SQL 查询。

// 修改字段依赖

composer require doctrine/dbal

//

///2/ 更新字段属性

change 方法可以将现有的字段类型修改为新的类型或修改属性。

比如，你可能想增加。字符串字段的长度，可以使用 change 方法把 name 字段的长度从 25 增加到 50。

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

$table->string('name', 50)->change();

});

我们同样可以将字段修改为可空。

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

$table->string('name', 50)->nullable()->change();

});

{注} 只有下面的字段类型能被 "修改"： bigInteger、 binary、 boolean、date、dateTime、dateTimeTz、decimal、integer、json、 longText、mediumText、smallInteger、string、text、time、 unsignedBigInteger、unsignedInteger and unsignedSmallInteger。

//

///3/ 重命名字段

可以使用结构生成器上的 renameColumn 方法来重命名字段。在重命名字段前 ， 请确保你的 composer.json 文件内已经加入 doctrine/dbal 依赖。

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

$table->renameColumn('from', 'to');

});

{注} 当前不支持 enum 类型的字段重命名。

(4)、删除字段

可以使用结构生成器上的 dropColumn 方法来删除字段。 在从 SQLite 数据库删除字段前，你需要在 composer.json 文件中加入 doctrine/dbal 依赖并在终端执行 composer update 来安装该依赖。

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

$table->dropColumn('votes');

});

你可以传递一个字段数组给 dropColumn 方法来删除多个字段。

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

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

});

{注} 不支持在使用 SQLite 数据库时在单个迁移中删除或修改多个字段。

//

///1/ 可用的命令别名

命令 描述

$table->dropMorphs('morphable'); 删除 morphable\_id 和 morphable\_type 字段。

$table->dropRememberToken(); 删除 remember\_token 字段。

$table->dropSoftDeletes(); 删除 deleted\_at 字段.

$table->dropSoftDeletesTz(); dropSoftDeletes() 方法的别名。

$table->dropTimestamps(); 删除 created\_at and updated\_at 字段。

$table->dropTimestampsTz(); dropTimestamps() 方法的别名。

--

7、索引

(1)、创建索引

结构生成器支持多种类型的索引。首先，先指定字段值唯一，即简单地在字段定义之后链式调用 unique 方法来创建索引，例如。

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

或者，你也可以在定义完字段之后创建索引。例如。

$table->unique('email');

你甚至可以将数组传递给索引方法来创建一个复合（或合成）索引。

$table->index(['account\_id', 'created\_at']);

Laravel 会自动生成一个合理的索引名称，但你也可以传递第二个参数来自定义索引名称。

$table->unique('email', 'unique\_email');

//

///1/ 可用的索引类型

每个索引方法都接受一个可选的第二个参数来指定索引的名称。如果省略，名称将根据表和列的名称生成。

(命令/描述)

$table->primary('id'); 添加主键

$table->primary(['id', 'parent\_id']); 添加复合键

$table->unique('email'); 添加唯一索引

$table->index('state'); 添加普通索引

$table->spatialIndex('location'); 添加空间索引（不支持 SQLite）

//

///2/ 索引长度 & Mysql / MariaDB

Laravel 默认使用 utf8mb4 编码，它支持在数据库中储存 emojis 。如果你是在版本低于 5.7.7 的 MySQL 或者版本低于 10.2.2 的 MariaDB 上创建索引，那你就需要手动配置数据库迁移的默认字符串长度。即在 AppServiceProvider 中调用 Schema::defaultStringLength 方法来配置它。

use Illuminate\Support\Facades\Schema;

/\*\*

\* Bootstrap any application services.

\*

\* @return void

\*/

public function boot()

{

Schema::defaultStringLength(191);

}

当然，你也可以选择开启数据库的 innodb\_large\_prefix 选项。至于如何正确开启，请自行查阅数据库文档。

(2)、重命名索引

若要重命名索引，你需要调用 renameIndex 方法。此方法接受当前索引名称作为其第一个参数，并将所需名称作为其第二个参数。

$table->renameIndex('from', 'to');

(3)、删除索引

若要删除索引，则必须指定索引的名称。Laravel 默认会自动将数据表名称、索引的字段名及索引类型简单地连接在一起作为名称。举例如下。

(命令/描述)

$table->dropPrimary('users\_id\_primary'); 从 users 表中删除主键

$table->dropUnique('users\_email\_unique'); 从 users 表中删除唯一索引

$table->dropIndex('geo\_state\_index'); 从 geo 表中删除基本索引

$table->dropSpatialIndex('geo\_location\_spatialindex'); 从 geo 表中删除空间索引（不支持 SQLite）

如果将字段数组传给 dropIndex 方法，会删除根据表名、字段和键类型生成的索引名称。

Schema::table('geo', function (Blueprint $table) {

$table->dropIndex(['state']); // Drops index 'geo\_state\_index'

});

(4)、外键约束

Laravel 还支持创建用于在数据库层中的强制引用完整性的外键约束。例如，让我们在 posts 表上定义一个引用 users 表的 id 字段的 user\_id 字段。

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

$table->unsignedBigInteger('user\_id');

$table->foreign('user\_id')->references('id')->on('users');

});

//

还可以为 on delete 和 on update 属性指定所需的操作。

$table->foreign('user\_id')

->references('id')->on('users')

->onDelete('cascade');

你可以使用 dropForeign 方法来删除外键。外键约束采用的命名方式与索引相同。即，将数据表名称和约束的字段连接起来，再加上 \_foreign 后缀。

$table->dropForeign('posts\_user\_id\_foreign');

或者，你也可以传递一个字段数组，在删除的时候会按照约定字段转换为对应的外键名称。

$table->dropForeign(['user\_id']);

//

//// 开启/关闭外键约束

你可以在迁移文件中使用以下方法来开启或关闭外键约束。

Schema::enableForeignKeyConstraints();

Schema::disableForeignKeyConstraints();

{注} SQLite 默认禁用外键约束。 使用 SQLite 时，请确保在数据库配置中启用 启用外键支持( https://learnku.com/docs/laravel/6.x/database#configuration ) 然后再尝试在迁移中创建它们。

--

--

五、数据填充 (数据库：填充)

--

https://learnku.com/docs/laravel/6.x/seeding/5174

--

1、简介

Laravel 包含一个填充类可以为你的数据库填充测试数据，所有的填充类都放在 database/seeds 目录下。你可以随意为填充类命名，但是更建议您遵守类似 UsersTableSeeder 的命名规范。通常， Laravel 默认定义了一个 DatabaseSeeder 类。通过这个类，你可以用 call 方法来运行其它的 seed 类从而控制数据填充的顺序。

--

2、编写 Seeders

运行 Artisan 命令 make:seeder 生成 Seeder，框架生成的 seeders 都放在 database/seeds 目录下。

// 生成Seeders类

php artisan make:seeder UsersTableSeeder

seeder 类只包含一个默认方法：run 。这个方法会在执行 db:seed 这个 Artisan 命令 ( https://learnku.com/docs/laravel/6.x/artisan ) 时被调用。在 run 方法里你可以根据需要在数据库中插入数据。你也可以用 查询构造器 ( https://learnku.com/docs/laravel/6.x/queries ) 或 Eloquent 模型工厂 ( https://learnku.com/docs/laravel/6.x/database-testing#writing-factories ) 来手动插入数据。

Tip：使用数据填充时会自动禁用 批量赋值保护 ( https://learnku.com/docs/laravel/6.x/eloquent#mass-assignment )。

//

如下所示，在默认的 DatabaseSeeder 类中的 run 方法中添加一条数据插入语句。

<?php

use Illuminate\Support\Str;

use Illuminate\Database\Seeder;

use Illuminate\Support\Facades\DB;

class DatabaseSeeder extends Seeder

{

/\*\*

\* Run the database seeds.

\*

\* @return void

\*/

public function run()

{

DB::table('users')->insert([

'name' => Str::random(10),

'email' => Str::random(10).'@gmail.com',

'password' => bcrypt('password'),

]);

}

}

?>

Tip：你在 run 的方法签名中可以用类型来约束你需要的依赖。它们会被 Laravel 服务容器 ( https://learnku.com/docs/laravel/6.x/container ) 自动解析。

(1)、使用模型工厂

手动为每个模型填充指定属性很麻烦，你可以使用 model 工厂 ( https://learnku.com/docs/laravel/6.x/database-testing#writing-factories ) 轻松地生成大量数据库数据。首先，阅读 model 工厂文档 ( https://learnku.com/docs/laravel/6.x/database-testing#writing-factories ) 来学习如何定义工厂文件，一旦定义好了你的工厂文件，然后就可以使用 factory 这个辅助函数来向数据库中插入数据。

//

例如，创建 50 个用户并为每个用户创建关联。

/\*\*

\* Run the database seeds.

\*

\* @return void

\*/

public function run()

{

factory(App\User::class, 50)->create()->each(function ($user) {

$user->posts()->save(factory(App\Post::class)->make());

});

}

(2)、调用其他 Seeders

在 DatabaseSeeder 类中，你可以使用 call 方法来运行其它的 seed 类。使用 call 方法可以将数据填充拆分成多个文件，这样就不会使单个 seeder 变得非常大。只需简单传递要运行的 seeder 类名称即可。

/\*\*

\* Run the database seeds.

\*

\* @return void

\*/

public function run()

{

$this->call([

UsersTableSeeder::class,

PostsTableSeeder::class,

CommentsTableSeeder::class,

]);

}

--

3、运行 Seeders

完成 seeder 类的编写之后，你可能需要使用 dump-autoload 命令重新生成 Composer 的自动加载器。

// 重新生成Composer自动加载器

composer dump-autoload

//

现在你可以使用 Artisan 命令 db:seed 来填充数据库了。默认情况下， db:seed 命令将运行 DatabaseSeeder 类，这个类可以用来调用其它 Seed 类。不过，你也可以使用 --class 选项来指定一个特定的 seeder 类。

php artisan db:seed

php artisan db:seed --class=UsersTableSeeder

//

你也可以用 migrate:refresh 这个命令来填充数据库，该命令会回滚并重新运行所有迁移。这个命令可以用来重建数据库。

php artisan migrate:refresh --seed

(2)、在生产环境中强制使用

一些填充操作可能会导致原有数据的更新或丢失。为了保护生产环境数据库的数据，在运行填充命令前会进行确认。可以添加 --force 选项来强制运行填充命令。

php artisan db:seed --force

--

--

六、Redis

--

https://learnku.com/docs/laravel/6.x/redis/5175

--

1、简介

Redis ( https://redis.io/ ) 是一个开源的，高级键值对存储数据库。由于它包含 字符串( https://redis.io/topics/data-types#strings ), 哈希 ( https://redis.io/topics/data-types#hashes ), 列表 ( https://redis.io/topics/data-types#lists ), 集合 ( https://redis.io/topics/data-types#sets ), 和 有序集合 ( https://redis.io/topics/data-types#sorted-sets )这些数据类型，所以它通常被称为数据结构服务器。

在将 Redis 与 Laravel 一起使用之前，我们鼓励你通过 PECL 安装并使用 PhpRedis ( https://github.com/phpredis/phpredis ) PHP 扩展。 尽管扩展安装起来更复杂，但对于大量使用 Redis 的应用程序可能会产生更好的性能。

或者，您可以通过 Composer 安装 predis / predis 包。

composer require predis/predis

注意：Predis 已被该软件包的原始作者遗弃，并可能在将来的版本中从 Laravel 中删除。

(1)、配置

Laravel 应用的 Redis 配置都在配置文件 config/database.php 中。在这个文件里，你可以看到 redis 数组里包含了应用程序使用的 Redis 服务器信息。

'redis' => [

'client' => env('REDIS\_CLIENT', 'phpredis'),

'default' => [

'host' => env('REDIS\_HOST', '127.0.0.1'),

'password' => env('REDIS\_PASSWORD', null),

'port' => env('REDIS\_PORT', 6379),

'database' => env('REDIS\_DB', 0),

],

'cache' => [

'host' => env('REDIS\_HOST', '127.0.0.1'),

'password' => env('REDIS\_PASSWORD', null),

'port' => env('REDIS\_PORT', 6379),

'database' => env('REDIS\_CACHE\_DB', 1),

],

],

默认的服务器配置应该足以进行开发。当然，你也可以根据使用的环境来随意更改这个数组。只需在配置文件中给每个 Redis 服务器指定名称、host（主机）和 port（端口）即可。

//

///1/ 集群配置

如果你的应用使用 Redis 服务器集群，你应该在 Redis 配置文件中使用 clusters 键来定义这些集群。

'redis' => [

'client' => env('REDIS\_CLIENT', 'phpredis'),

'clusters' => [

'default' => [

[

'host' => env('REDIS\_HOST', 'localhost'),

'password' => env('REDIS\_PASSWORD', null),

'port' => env('REDIS\_PORT', 6379),

'database' => 0,

],

],

],

],

默认情况下，集群可以在节点上实现客户端分片，允许你实现节点池以及创建大量可用内存。这里要注意，客户端共享不会处理失败的情况；因此，这个功能主要适用于从另一个主数据库获取的缓存数据。如果要使用 Redis 原生集群，需要在配置文件下的 options 键中做出如下指定。

'redis' => [

'client' => env('REDIS\_CLIENT', 'phpredis'),

'options' => [

'cluster' => env('REDIS\_CLUSTER', 'redis'),

],

'clusters' => [

// ...

],

],

(2)、Predis

如果您要使用 Predis 扩展，您应该将 REDIS\_CLIENT 环境变量从 phpredis 更改为 predis。

'redis' => [

'client' => env('REDIS\_CLIENT', 'predis'),

// Rest of Redis configuration...

],

除了默认的 host, port, database, 和 password 这些服务配置选项外，Predis 还支持为每个 Redis 服务器定义其它的链接参数 connection parameters ( https://github.com/nrk/predis/wiki/Connection-Parameters )。如果要使用这些额外的配置选项，可以将以下配置添加到 Redis 服务器的配置文件 config/database.php 中。

'default' => [

'host' => env('REDIS\_HOST', 'localhost'),

'password' => env('REDIS\_PASSWORD', null),

'port' => env('REDIS\_PORT', 6379),

'database' => 0,

'read\_write\_timeout' => 60,

],

(3)、PhpRedis

要使用 PhpRedis 扩展，您应该在 env 配置 REDIS\_CLIENT 和 config/database.php 配置中为默认值。

'redis' => [

'client' => env('REDIS\_CLIENT', 'phpredis'),

// 其余的 Redis 配置...

],

如果您计划将 PhpRedis 扩展名与 Redis Facade 别名一起使用，则应将其重命名为其他内容，例如 RedisManager，以避免与 Redis 类发生冲突。 您可以在 app.php 配置文件的别名部分执行此操作。

'RedisManager' => Illuminate\Support\Facades\Redis::class,

除了默认的 host， port， database 和 password 这些服务配置选项外，PhpRedis 还支持以下几个额外的连接参数： persistent， prefix， read\_timeout 和 timeout 。你可以将这些配置选项添加到配置文件 config/database.php 中 Redis 服务配置项如下。

'default' => [

'host' => env('REDIS\_HOST', 'localhost'),

'password' => env('REDIS\_PASSWORD', null),

'port' => env('REDIS\_PORT', 6379),

'database' => 0,

'read\_timeout' => 60,

],

//

///1/ Redis Facade

为了避免与 Redis PHP 扩展本身发生类命名冲突，您需要从 app 配置文件的 aliases 数组中删除或重命名 Illuminate \ Support \ Facades \ Redis 外观别名。 通常，您应该完全删除此别名，并且仅在使用 Redis PHP 扩展时通过其完全限定的类名引用 Facade。

--

2、Redis 交互

你可以通过调用 Redis facade 上的各种方法来与 Redis 交互。Redis facade ( https://learnku.com/docs/laravel/6.x/facades ) 支持动态方法，意味着你可以在 facade 上调用各种 Redis 命令 ( https://redis.io/commands )，命令将直接传递给 Redis。在示例中，将通过 Redis facade 上的 get 方法来调用 Redis 上的 GET 命令。

<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

use Illuminate\Support\Facades\Redis;

class UserController extends Controller

{

/\*\*

\* 显示给定用户的配置文件。

\*

\* @param int $id

\* @return Response

\*/

public function showProfile($id)

{

$user = Redis::get('user:profile:'.$id);

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

}

}

?>

如上所述，你可以在 Redis facade 上调用任意 Redis 命令。Laravel 使用魔术方法将命令传递给 Redis 服务器，因此只需传递 Redis 命令所需的参数即可。

Redis::set('name', 'Taylor');

$values = Redis::lrange('names', 5, 10);

或者，你也可以使用 command 方法将命令传递给服务器，它接受命令的名称作为其第一个参数，并将值的数组作为其第二个参数。

$values = Redis::command('lrange', ['name', 5, 10]);

(1)、使用多个 Redis 连接

你可以通过 Redis::connection 方法获得 Redis 实例。

$redis = Redis::connection();

这会返回一个默认的 Redis 实例。你可以传递连接或者集群名称给 connection 方法来获取在 Redis 配置中特定服务或集群。

$redis = Redis::connection('my-connection');

(1)、管道命令

当你需要在一个操作中给服务器发送很多命令时，推荐你使用管道命令。 pipeline 方法接受一个 Redis 实例的 闭包 。你可以将所有的命令发送给 Redis 实例，它们都会在一个操作中执行完成。

Redis::pipeline(function ($pipe) {

for ($i = 0; $i < 1000; $i++) {

$pipe->set("key:$i", $i);

}

});

--

3、发布 / 订阅

Laravel 为 Redis 的 publish 及 subscribe 提供了方便的接口。这些 Redis 命令让你可以监听指定「频道」上的消息。你可以从另一个应用程序发布消息给另一个应用程序，甚至使用其它编程语言，让应用程序和进程之间能够轻松进行通信。

首先，我们使用 subscribe 方法设置频道监听器。我们将这个方法调用放在 Artisan 命令 ( https://learnku.com/docs/laravel/6.x/artisan ) 中，因为调用 subscribe 方法会启动一个长时间运行的进程。

<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;

use Illuminate\Support\Facades\Redis;

class RedisSubscribe extends Command

{

/\*\*

\* The name and signature of the console command.

\*

\* @var string

\*/

protected $signature = 'redis:subscribe';

/\*\*

\* The console command description.

\*

\* @var string

\*/

protected $description = 'Subscribe to a Redis channel';

/\*\*

\* Execute the console command.

\*

\* @return mixed

\*/

public function handle()

{

Redis::subscribe(['test-channel'], function ($message) {

echo $message;

});

}

}

?>

现在我们可以使用 publish 方法将消息发布到频道。

// Redis发布/订阅模式\_发布信息

Route::get('publish', function () {

// Route logic...

Redis::publish('test-channel', json\_encode(['foo' => 'bar']));

});

(1)、通配符订阅

使用 psubscribe 方法可以订阅通配符频道，可以用来在所有频道上获取所有消息。$channel 名称将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;

});

--

--

第九卷 Eloquent ORM

--

--

一、快速入门 ( Eloquent：入门 )

--

https://learnku.com/docs/laravel/6.x/eloquent/5176

--

1、简介

Laravel 的 Eloquent ORM 提供了一个漂亮、简洁的 ActiveRecord 实现来和数据库交互。每个数据库表都有一个对应的「模型」用来与该表交互。你可以通过模型查询数据表中的数据，以及在数据表中插入新记录。

在开始之前，请确保在 config/database.php 中配置数据库连接。更多关于数据库配置的信息，请查看 文档 ( https://learnku.com/docs/laravel/6.x/database#configuration )。

--

2、模型定义

首先，创建一个 Eloquent 模型。 模型通常在 app 目录中，但你可以根据 composer.json 文件将他们放置在可以被自动加载的任意位置。所有的 Eloquent 模型都继承至 Illuminate\Database\Eloquent\Model 类。

创建模型最简单的方法就是使用 make:model Artisan 命令。

// 创建模型

php artisan make:model Flight

如果要在生成模型的时候生成 数据库迁移 ( https://learnku.com/docs/laravel/6.x/migrations )，可以使用 --migration 或 -m 选项。

// 指定迁移

php artisan make:model Flight --migration

php artisan make:model Flight -m

(1)、模型约定 (Eloquent 模型约定)

现在，我们来看一个 Flight 模型的示例，我们将用它从 flights 数据库表中检索和存储数据信息。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model

{

//

}

?>

//

///1/ 数据表名称

请注意，我们并没有告诉 Eloquent 我们的 Flight 模型使用哪个数据表。 除非明确地指定了其它名称，否则将使用类的复数形式「蛇形命名」来作为表名。因此，在这种情况下，Eloquent 将假设 Flight 模型存储的是 flights 数据表中的数据。你可以通过在模型上定义 table 属性来指定自定义数据表。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model

{

/\*\*

\* 与模型关联的表名

\*

\* @var string

\*/

protected $table = 'my\_flights';

}

?>

//

///2/ 主键

Eloquent 也会假设每个数据表都有一个名为 id 的主键列。你可以定义一个受保护的 $primaryKey 属性来重写约定。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model

{

/\*\*

\* 重定义主键

\*

\* @var string

\*/

protected $primaryKey = 'flight\_id';

}

?>

此外，Eloquent 假设主键是一个自增的整数值，这意味着默认情况下主键会自动转换为 int 类型。如果您希望使用非递增或非数字的主键则需要设置公共的 $incrementing 属性设置为 false。

<?php

class Flight extends Model

{

/\*\*

\* 指示模型主键是否递增

\*

\* @var bool

\*/

public $incrementing = false;

}

?>

如果你的主键不是一个整数，你需要将模型上受保护的 $keyType 属性设置为 string。

<?php

class Flight extends Model

{

/\*\*

\* 自动递增ID的“类型”。

\*

\* @var string

\*/

protected $keyType = 'string';

}

//

///3/ 时间戳

默认情况下，Eloquent 预期你的数据表中存在 created\_at 和 updated\_at 。如果你不想让 Eloquent 自动管理这两个列， 请将模型中的 $timestamps 属性设置为 false。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model

{

/\*\*

\* 指示是否自动维护时间戳

\*

\* @var bool

\*/

public $timestamps = false;

}

?>

如果需要自定义时间戳的格式，在你的模型中设置 $dateFormat 属性。这个属性决定日期属性在数据库的存储方式，以及模型序列化为数组或者 JSON 的格式。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model

{

/\*\*

\* 模型日期列的存储格式。

\*

\* @var string

\*/

protected $dateFormat = 'U';

}

?>

如果你需要自定义存储时间戳的字段名，可以在模型中设置 CREATED\_AT 和 UPDATED\_AT 常量的值来实现。

<?php

class Flight extends Model

{

const CREATED\_AT = 'creation\_date';

const UPDATED\_AT = 'last\_update';

}

?>

//

///4/ 数据库连接

默认情况下，Eloquent 模型将使用你的应用程序配置的默认数据库连接。如果你想为模型指定一个不同的连接，设置 $connection 属性。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model

{

/\*\*

\* 模型的连接名称

\*

\* @var string

\*/

protected $connection = 'connection-name';

}

?>

(2)、默认属性值

如果要为模型的某些属性定义默认值，可以在模型上定义 $attributes 属性。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model

{

/\*\*

\* 模型的默认属性值。

\*

\* @var array

\*/

protected $attributes = [

'delayed' => false,

];

}

?>

--

3、模型检索

创建模型和 它关联的数据库表 ( https://learnku.com/docs/laravel/6.x/migrations#writing-migrations ) 后，你就可以从数据库中查询数据了。将每个 Eloquent 模型想象成一个强大的查询构造器 query builder ( https://learnku.com/docs/laravel/6.x/queries )，你可以用它更快速的查询与其相关联的数据表。例如。

<?php

$flights = App\Flight::all();

foreach ($flights as $flight) {

echo $flight->name;

}

?>

//

(1)、附加约束

Eloquent 的 all 方法会返回模型中所有的结果。由于每个 Eloquent 模型都充当一个查询构造器( https://learnku.com/docs/laravel/6.x/queries )，所以你也可以添加查询条件，然后使用 get 方法获取查询结果。

$flights = App\Flight::where('active', 1)

->orderBy('name', 'desc')

->take(10)

->get();

Tip：因为 Eloquent 模型也是查询构造器，所以你也应当阅读 查询构造器( https://learnku.com/docs/laravel/6.x/queries )可用的所有方法。你可以在 Eloquent 查询中使用这些方法。

//

(2)、重新加载模型

你可以使用 fresh 和 refresh 方法重新加载模型。 fresh 方法会重新从数据库中检索模型。现有的模型实例不受影响。

$flight = App\Flight::where('number', 'FR 900')->first();

$freshFlight = $flight->fresh();

refresh 方法使用数据库中的新数据重新赋值现有模型。此外，已经加载的关系会被重新加载。

$flight = App\Flight::where('number', 'FR 900')->first();

$flight->number = 'FR 456';

$flight->refresh();

$flight->number; // "FR 900"

(3)、集合

Eloquent 的 all 和 get 方法可以查询到多个结果，返回一个 Illuminate\Database\Eloquent\Collection 实例。 Collection 类提供了 大量的辅助函数( https://learnku.com/docs/laravel/6.0/eloquent-collections#available-methods ) 来处理 Eloquent 结果。

$flights = $flights->reject(function ($flight) {

return $flight->cancelled;

});

你可以像数组一样遍历集合。

foreach ($flights as $flight) {

echo $flight->name;

}

(4)、结果分块 (分块结果)

如果你需要处理数以千计的 Eloquent 结果，使用 chunk 命令。 chunk 方法会检索 Eloquent 模型中的『分块』将他们提供给指定的 Closure 处理。在处理大型结果集时，使用 chunk 方法可以节省内存。

Flight::chunk(200, function ($flights) {

foreach ($flights as $flight) {

//

}

});

传递到方法的第一个参数是希望每个『分块』接收的数据量。闭包作为第二个参数传递，它在每次从数据库中检索分块的时候调用。它将执行数据库查询把检索分块的结果传递给闭包方法。

(5)、使用游标

cursor 方法允许你使用游标遍历数据库，它只执行一次查询。处理大量的数据时， cursor 方法可以大大减少内存的使用量。

foreach (Flight::where('foo', 'bar')->cursor() as $flight) {

//

}

cursor 返回 Illuminate\Support\LazyCollection 实例。 Lazy collections ( https://learnku.com/docs/laravel/6.0/collections#lazy-collections ) 允许你使用 Laravel 集合中大多数集合方法，而且每次只会加载单个模型到内存中。

$users = App\User::cursor()->filter(function ($user) {

return $user->id > 500;

});

foreach ($users as $user) {

echo $user->id;

}

(6)、高级子查询

//

///1/ Selects 子查询

Eloquent 提供了高级子查询支持，你可以用单条查询语句从相关表中提取信息。

举个例子，假设我们有一个目的地表 destinations 和一个到目的地的航班表 flights。flights 表包含一个 arrival\_at 字段，表示航班何时到达目的地。

使用子查询功能提供的 select 和 addSelect 方法，我们可以用单条语句查询全部目的地 destinations，以及抵达各目的地最后一班飞机的名称。

use App\Flight;

use App\Destination;

return Destination::addSelect(['last\_flight' => Flight::select('name')

->whereColumn('destination\_id', 'destinations.id')

->orderBy('arrived\_at', 'desc')

->latest()

->limit(1)

])->get();

//

///2/ Ordering 子查询

此外，查询构建器的 orderBy 函数也支持子查询。我们可以使用此功能根据最后一班航班到达目的地的时间对所有目的地排序。 同样，这可以只对数据库执行单个查询。

return Destination::orderByDesc(

Flight::select('arrived\_at')

->whereColumn('destination\_id', 'destinations.id')

->orderBy('arrived\_at', 'desc')

->latest()

->limit(1)

)->get();

--

4、检索单个模型 / 集合

除了从指定的数据表检索所有记录外，你可以使用 find 或 first 方法来检索单条记录。这些方法返回单个模型实例，而不是返回模型集合。

// 通过主键检索一个模型...

$flight = App\Flight::find(1);

// 检索符合查询限制的第一个模型...

$flight = App\Flight::where('active', 1)->first();

你也可以使用主键数组作为参数调用 find 方法，它将返回匹配记录的集合。

$flights = App\Flight::find([1, 2, 3]);

(1)、『未找到』异常

有时你希望在未找到模型时抛出异常。这在控制器和路由中非常有用。 findOrFail 和 firstOrFail 方法会检索查询的第一个结果，如果未找到，将抛出 Illuminate\Database\Eloquent\ModelNotFoundException 异常。

$model = App\Flight::findOrFail(1);

$model = App\Flight::where('legs', '>', 100)->firstOrFail();

如果没有捕获异常，则会自动返回 404 响应给用户。也就是说，使用这些方法时，没有必要再写个检查来返回 404 响应。

Route::get('/api/flights/{id}', function ($id) {

return App\Flight::findOrFail($id);

});

(2)、检索集合

你还可以使用 查询构造器 ( https://learnku.com/docs/laravel/6.0/queries ) 提供的 count ， sum， max, 和其他的聚合函数( https://learnku.com/docs/laravel/6.0/queries#aggregates )。这些方法只会返回适当的标量值而不是一个模型实例。

$count = App\Flight::where('active', 1)->count();

$max = App\Flight::where('active', 1)->max('price');

--

5、插入 & 更新模型

(1)、插入

要往数据库新增一条记录，先创建新模型实例，给实例设置属性，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)

{

// 校验请求...

$flight = new Flight;

$flight->name = $request->name;

$flight->save();

}

}

?>

在这个示例中，我们将 HTTP 请求参数 name 赋值给了 App\Flight 模型实例的 name 属性。当调用 save 方法时，将会插入一条新记录。 created\_at 和 updated\_at 时间戳将会自动设置，不需要手动赋值。

(2)、更新

save 方法也可以用来更新数据库已经存在的模型。更新模型，你需要先检索出来，设置要更新的属性，然后调用 save 方法。同样， updated\_at 时间戳会自动更新，所以也不需要手动赋值。

$flight = App\Flight::find(1);

$flight->name = 'New Flight Name';

$flight->save();

//

///1/ 批量更新

也可以更新匹配查询条件的多个模型。在这个示例中，所有的 active 和 destination 为 San Diego 的航班会标记为延误。

App\Flight::where('active', 1)

->where('destination', 'San Diego')

->update(['delayed' => 1]);

update 方法接受一个键为字段名称数据为值的数组。

注意：通过 Eloquent 批量更新时， 更新的模型不会触发 saved 和 updated 事件。因为在批量更新时，从不会去检索模型。

(3)、批量赋值

你也可以使用 create 方法来保存新模型，此方法会返回模型实例。不过，在使用之前，你需要在模型上指定 fillable 或 guarded 属性，因为所有的 Eloquent 模型都默认不可进行批量赋值。

当用户通过 HTTP 请求传入一个意外的参数，并且该参数更改了数据库中你不需要更改的字段时。比如：恶意用户可能会通过 HTTP 请求传入 is\_admin 参数，然后将其传给 create 方法，此操作能让用户将自己升级成管理员。

所以，在开始之前，你应该定义好模型上的哪些属性是可以被批量赋值的。你可以通过模型上的 $fillable 属性来实现。 例如：让 Flight 模型的 name 属性可以被批量赋值。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model

{

/\*\*

\* 可以被批量赋值的属性。

\*

\* @var array

\*/

protected $fillable = ['name'];

}

?>

一旦我们设置好了可以批量赋值的属性，就可以通过 create 方法插入新数据到数据库中了。 create 方法将返回保存的模型实例。

$flight = App\Flight::create(['name' => 'Flight 10']);

如果你已经有一个模型实例，你可以传递一个数组给 fill 方法来赋值。

$flight->fill(['name' => 'Flight 22']);

//

///1/ 保护属性

$fillable 可以看作批量赋值的「白名单」, 你也可以使用 $guarded 属性来实现。 $guarded 属性包含的是不允许批量赋值的数组。也就是说， $guarded 从功能上将更像是一个「黑名单」。注意：你只能使用 $fillable 或 $guarded 二者中的一个，不可同时使用。下面这个例子中，除了 price 属性，其他的属性都可以批量赋值。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model

{

/\*\*

\* 不可批量赋值的属性。

\*

\* @var array

\*/

protected $guarded = ['price'];

}

?>

如果你想让所有属性都可以批量赋值， 你可以将 $guarded 定义成一个空数组。

/\*\*

\* 不可批量赋值的属性。

\*

\* @var array

\*/

protected $guarded = [];

(4)、其他创建方法

//

///1/ firstOrCreate/ firstOrNew

这里有两个你可能用来批量赋值的方法： firstOrCreate 和 firstOrNew。firstOrCreate 方法会通过给定的 列 / 值 来匹配数据库中的数据。如果在数据库中找不到对应的模型， 则会从第一个参数的属性乃至第二个参数的属性中创建一条记录插入到数据库。

firstOrNew 方法像 firstOrCreate 方法一样尝试通过给定的属性查找数据库中的记录。不同的是，如果 firstOrNew 方法找不到对应的模型，会返回一个新的模型实例。注意 firstOrNew 返回的模型实例尚未保存到数据库中，你需要手动调用 save 方法来保存。

// 通过 name 来查找航班，不存在则创建...

$flight = App\Flight::firstOrCreate(['name' => 'Flight 10']);

//

// 通过 name 查找航班，不存在则使用 name、 delayed 和 arrival\_time 属性创建...

$flight = App\Flight::firstOrCreate(

['name' => 'Flight 10'],

['delayed' => 1, 'arrival\_time' => '11:30']

);

//

// 通过 name 查找航班，不存在则创建一个实例...

$flight = App\Flight::firstOrNew(['name' => 'Flight 10']);

//

// 通过 name 查找航班，不存在则使用 name 和 delayed 属性创建一个实例...

$flight = App\Flight::firstOrNew(

['name' => 'Flight 10'],

['delayed' => 1, 'arrival\_time' => '11:30']

);

//

///2/ updateOrCreate

你还可能遇到希望更新现有模型或在不存在的情况下则创建新的模型的情景。 Laravel 提供了 updateOrCreate 方法仅一个步骤就可以实现。跟 firstOrCreate 方法一样，updateOrCreate 匹配到对应模型，所以不需要调用 save() 方法。

// 如果有从奥克兰到圣地亚哥的航班，则价格定为99美元。

// 如果没匹配到存在的模型，则创建一个。

$flight = App\Flight::updateOrCreate(

['departure' => 'Oakland', 'destination' => 'San Diego'],

['price' => 99, 'discounted' => 1]

);

--

6、删除模型

可以在模型实例上调用 delete 方法来删除实例。

$flight = App\Flight::find(1);

$flight->delete();

(1)、通过主键删除模型

在上面的例子中，在调用 delete 之前需要先去数据库中查找对应的模型。事实上，如果你知道了模型的主键，你可以直接使用 destroy 方法来删除模型，而不用先去数据库中查找。 destroy 方法除了接受单个主键作为参数之外，还接受多个主键，或者使用数组，集合来保存多个主键。

App\Flight::destroy(1);

App\Flight::destroy(1, 2, 3);

App\Flight::destroy([1, 2, 3]);

App\Flight::destroy(collect([1, 2, 3]));

(2)、通过查询删除模型

你也可以在模型上运行删除语句。在这个例子中，我们将删除所有标记为非活跃的航班。与批量更新一样，批量删除不会为删除的模型启动任何模型事件。

$deletedRows = App\Flight::where('active', 0)->delete();

注意：通过 Eloquent 执行批量删除语句时，不会触发 deleting 和 deleted 模型事件。因此，在执行删除语句时，从不检索模型示例。

(3)、软删除

除了真实删除数据库记录， Eloquent 也可以「软删除」模型。软删除的模型并不是真的从数据库中删除了。事实上，是在模型上设置了 deleted\_at 属性并将其值写入数据库。如果 deleted\_at 值非空，代表这个模型已被软删除。如果要开启模型软删除功能，你需要在模型上使用 Illuminate\Database\Eloquent\SoftDeletes trait。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

use Illuminate\Database\Eloquent\SoftDeletes;

class Flight extends Model

{

use SoftDeletes;

}

?>

Tip： SoftDeletes trait 会自动将 deleted\_at 属性转换成 DateTime / Carbon 实例。

当然，你需要把 deleted\_at 字段添加到数据表中。 Laravel 的 数据库迁移 ( https://learnku.com/docs/laravel/6.0/migrations ) 有创建这个字段的方法。

Schema::table('flights', function (Blueprint $table) {

$table->softDeletes();

});

那现在，当你在模型实例上使用 delete 方法， 当前日期时间会写入 deleted\_at 字段。同时，查询出来的结果也会自动排除已被软删除的记录。

你可以使用 trashed 方法来验证给定的模型实例是否已被软删除。

if ($flight->trashed()) {

//

}

(4)、查询软删除的模型 ( 查询软删除模型 )

//

///1/ 包括已软删除的模型

前面提到，查询结果会自动剔除已被软删除的结果。当然，你可以使用 withTrashed 方法来获取包括软删除模型在内的模型。

$flights = App\Flight::withTrashed()

->where('account\_id', 1)

->get();

withTrashed 方法也可以用在 关联 ( https://learnku.com/docs/laravel/6.x/eloquent-relationships ) 查询。

$flight->history()->withTrashed()->get();

//

///2/ 只检索软删除模型

onlyTrashed 方法 只 获取已软删除的模型。

$flights = App\Flight::onlyTrashed()

->where('airline\_id', 1)

->get();

//

///3/ 恢复软删除模型

有时会对软删除模型进行「撤销」，在已软删除的数据上使用 restore 方法即可恢复到有效状态。

$flight->restore();

你也可以在查询中使用 restore 方法，从而快速恢复多个模型。和其他「批量」操作一样，这个操作不会触发模型的任何事件。

App\Flight::withTrashed()

->where('airline\_id', 1)

->restore();

类似 withTrashed 方法，restore 方法也用在 关联 ( https://learnku.com/docs/laravel/6.x/eloquent-relationships ) 上。

$flight->history()->restore();

//

///4/ 永久删除

有时你可能需要从数据库中真正删除模型。要从数据库中永久删除软删除的模型，请使用 forceDelete 方法。

// 强制删除单个模型实例...

$flight->forceDelete();

// 强制删除所有相关模型...

$flight->history()->forceDelete();

--

7、查询作用域

(1)、全局作用域

全局作用域可以给模型的查询都添加上约束。Laravel 的 软删除 功能就是利用此特性从数据库中获取「未删除」的模型。你可以编写你自己的全局作用域，很简单、方便的为每个模型查询都加上约束条件。

//

///1/ 编写全局作用域

编写全局作用域很简单。定义一个实现 Illuminate\Database\Eloquent\Scope 接口的类，并实现 apply 这个方法。根据你的需求，在 apply 方法中加入查询的 where 条件。

<?php

namespace App\Scopes;

use Illuminate\Database\Eloquent\Scope;

use Illuminate\Database\Eloquent\Model;

use Illuminate\Database\Eloquent\Builder;

class AgeScope implements Scope

{

/\*\*

\* 把约束加到 Eloquent 查询构造中。

\*

\* @param \Illuminate\Database\Eloquent\Builder $builder

\* @param \Illuminate\Database\Eloquent\Model $model

\* @return void

\*/

public function apply(Builder $builder, Model $model)

{

$builder->where('age', '>', 200);

}

}

?>

提示：如果你需要在 select 语句里添加字段，应使用 addSelect 方法，而不是 select 方法。这将有效防止无意中替换现有 select 语句的情况。

//

///2/ 应用全局作用域

要将全局作用域分配给模型，需要重写模型的 boot 方法并使用 addGlobalScope 方法。

<?php

namespace App;

use App\Scopes\AgeScope;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

/\*\*

\* 模型的「启动」方法

\*

\* @return void

\*/

protected static function boot()

{

parent::boot();

static::addGlobalScope(new AgeScope);

}

}

?>

添加作用域后，对 User::all() 的查询会生成以下 SQL 查询语句。

select \* from `users` where `age` > 200

//

///3/ 匿名全局作用域

Eloquent 同样允许使用闭包定义全局作用域，这样就不需要为一个简单的作用域而编写一个单独的类。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

use Illuminate\Database\Eloquent\Builder;

class User extends Model

{

/\*\*

\* 模型的「启动」方法

\*

\* @return void

\*/

protected static function boot()

{

parent::boot();

static::addGlobalScope('age', function (Builder $builder) {

$builder->where('age', '>', 200);

});

}

}

?>

//

///4/ 取消全局作用域

如果需要对当前查询取消全局作用域，需要使用 withoutGlobalScope 方法。 该方法仅接受全局作用域类名作为它唯一的参数。

User::withoutGlobalScope(AgeScope::class)->get();

或者，如果使用闭包定义全局作用域的话。

User::withoutGlobalScope('age')->get();

如果你需要取消部分或者全部的全局作用域的话，需要使用 withoutGlobalScopes 方法。

//

// 取消所有的全局作用域...

User::withoutGlobalScopes()->get();

//

// 取消部分全局作用域...

User::withoutGlobalScopes([

FirstScope::class, SecondScope::class

])->get();

(2)、本地作用域

本地作用域允许定义通用的约束集合以便在应用程序中重复使用。例如，你可能经常需要获取所有「流行」的用户。要定义这样一个范围，只需要在对应的 Eloquent 模型方法前添加 scope 前缀。作用域总是返回一个查询构造器实例。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

/\*\*

\* 只查询受欢迎的用户的作用域

\*

\* @param \Illuminate\Database\Eloquent\Builder $query

\* @return \Illuminate\Database\Eloquent\Builder

\*/

public function scopePopular($query)

{

return $query->where('votes', '>', 100);

}

/\*\*

\* 只查询 active 用户的作用域

\*

\* @param \Illuminate\Database\Eloquent\Builder $query

\* @return \Illuminate\Database\Eloquent\Builder

\*/

public function scopeActive($query)

{

return $query->where('active', 1);

}

}

?>

//

///1/ 使用本地作用域

一旦定义了作用域，就可以在查询该模型时调用作用域方法。不过，在调用这些方法时不必包含 scope 前缀。甚至可以链式调用多个作用域，例如。

$users = App\User::popular()->active()->orderBy('created\_at')->get();

借助 or 查询运行符整合多个 Eloquent 模型，可能需要使用闭包回调。

$users = App\User::popular()->orWhere(function (Builder $query) {

$query->active();

})->get();

因为这样可能会有点麻烦，Laravel 提供了「更高阶的」orWhere 方法，它允许你在链式调用作用域时不使用闭包。

$users = App\User::popular()->orWhere->active()->get();

//

///2/ 动态作用域

有时可能地希望定义一个可以接受参数的作用域。把额外参数传递给作用域就可以达到此目的。作用域参数要放在 $query 参数之后。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

/\*\*

\* 将查询作用域限制为仅包含给定类型的用户

\*

\* @param \Illuminate\Database\Eloquent\Builder $query

\* @param mixed $type

\* @return \Illuminate\Database\Eloquent\Builder

\*/

public function scopeOfType($query, $type)

{

return $query->where('type', $type);

}

}

?>

这样就可以在调用作用域时传递参数了。

$users = App\User::ofType('admin')->get();

--

8、模型对比 (模型比较)

有时可能需要判断两个模型是否「相同」。is 方法可以用来快速校验两个模型是否拥有相同的主键、表和数据库连接。

if ($post->is($anotherPost)) {

//

}

--

9、事件

Eloquent 模型触发几个事件，允许你挂接到模型生命周期的如下节点： retrieved、creating、created、updating、updated、saving、saved、deleting、deleted、restoring 和 restored。事件允许你每当特定模型保存或更新数据库时执行代码。每个事件通过其构造器接受模型实例。

retrieved 事件在现有模型从数据库中查找数据时触发。当新模型每一次保存时，creating 和 created 事件被触发。如果数据库中已经存在模型并且调用了 save 方法，updating / updated 事件被触发。这些情况下，saving / saved 事件也被触发。

注意：通过 Eloquent 进行批量更新时，被更新模型的 saved 和 updated 事件不会被触发。这是因为批量更新时，并没有真的获取模型。

首先，在 Eloquent 模型上定义一个 $dispatchesEvents 属性，将 Eloquent 模型生命周期的几个节点映射到你自己的 event 类 ( https://learnku.com/docs/laravel/6.x/events )。

<?php

namespace App;

use App\Events\UserSaved;

use App\Events\UserDeleted;

use Illuminate\Notifications\Notifiable;

use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable

{

use Notifiable;

/\*\*

\* 模型的事件映射

\*

\* @var array

\*/

protected $dispatchesEvents = [

'saved' => UserSaved::class,

'deleted' => UserDeleted::class,

];

}

?>

定义并且映射了 Eloquent 事件，就可以使用 事件监听 ( https://learnku.com/docs/laravel/6.x/events#defining-listeners ) 处理这些事件了。

(1)、观察器 (观察者)

//

///1/ 定义观察者

如果在一个模型上监听了多个事件，可以使用观察者来将这些监听器组织到一个单独的类中。观察者类的方法名映射到你希望监听的 Eloquent 事件。 这些方法都以模型作为其唯一参数。make:observer Artisan 命令可以快速建立新的观察者类。

// 模型事件观察者创建

php artisan make:observer UserObserver --model=User

此命令将在 App/Observers 文件夹放置新的观察者类。如果这个目录不存在，Artisan 将替你创建。使用如下方式开启观察者。

<?php

namespace App\Observers;

use App\User;

class UserObserver

{

/\*\*

\* 处理 User「新建」事件

\*

\* @param \App\User $user

\* @return void

\*/

public function created(User $user)

{

//

}

/\*\*

\* 处理 User「更新」事件

\*

\* @param \App\User $user

\* @return void

\*/

public function updated(User $user)

{

//

}

/\*\*

\* 处理 User「删除」事件

\*

\* @param \App\User $user

\* @return void

\*/

public function deleted(User $user)

{

//

}

}

?>

在你希望观察的模型上使用 observe 方法注册观察者。也可以在服务提供者的 boot 方法注册观察者。下面是在 AppServiceProvider 中注册观察者的示例。

<?php

namespace App\Providers;

use App\User;

use App\Observers\UserObserver;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider

{

/\*\*

\* 注册服务提供者

\*

\* @return void

\*/

public function register()

{

//

}

/\*\*

\* 引导应用程序服务.

\*

\* @return void

\*/

public function boot()

{

User::observe(UserObserver::class);

}

}

?>

--

--

二、模型关联 (Eloquent：关联)

--

https://learnku.com/docs/laravel/6.x/eloquent-relationships/5177

--

1、简介

数据库表通常相互关联。例如，一篇博客文章可能有很多评论，或者一个订单对应一个下单用户。 Eloquent 让这些关联的管理和使用变得简单，并支持多种类型的关联。一对一、一对多、多对多、远程一对一、远程一对多、一对一 (多态关联)、一对多 (多态关联)、多对多 (多态关联)。

--

2、定义关联

Eloquent 关联在 Eloquent 模型类中以方法的形式呈现。如同 Eloquent 模型本身，关联也可以作为强大的 查询语句构造器( https://learnku.com/docs/laravel/6.x/queries ) 使用，提供了强大的链式调用和查询功能。例如，我们可以在 posts 关联的链式调用中附加一个约束条件。

$user->posts()->where('active', 1)->get();

不过在深入使用关联之前，让我们先学习如何定义每种关联类型。

(1)、一对一

一对一是最基本的关联关系。例如，一个 User 模型可能关联一个 Phone 模型。为了定义这个关联，我们要在 User 模型中写一个 phone 方法。在 phone 方法内部调用 hasOne 方法并返回其结果。

//

///1/ 一对一

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

/\*\*

\* 获取与用户关联的电话记录。

\*/

public function phone()

{

return $this->hasOne('App\Phone');

}

}

?>

hasOne 方法的第一个参数是关联模型的类名。一旦定义了模型关联，我们就可以使用 Eloquent 动态属性获得相关的记录。动态属性允许你访问关系方法就像访问模型中定义的属性一样。

$phone = User::find(1)->phone;

Eloquent 会基于模型名决定外键名称。在这种情况下，会自动假设 Phone 模型有一个 user\_id 外键。如果你想覆盖这个约定，可以传递第二个参数给 hasOne 方法。

return $this->hasOne('App\Phone', 'foreign\_key');

另外，Eloquent 假设外键的值是与父级 id (或自定义 $primaryKey) 列的值相匹配的。换句话说，Eloquent 将会在 Phone 记录的 user\_id 列中查找与用户表的 id 列相匹配的值。如果您希望该关联使用 id 以外的自定义键名，则可以给 hasOne 方法传递第三个参数。

return $this->hasOne('App\Phone', 'foreign\_key', 'local\_key');

//

///2/ 定义反向关联

我们已经能从 User 模型访问到 Phone 模型了。现在，让我们再在 Phone 模型上定义一个关联，这个关联能让我们访问到拥有该电话的 User 模型。我们可以使用与 hasOne 方法对应的 belongsTo 方法来定义反向关联。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Phone extends Model

{

/\*\*

\* 获得拥有此电话的用户。

\*/

public function user()

{

return $this->belongsTo('App\User');

}

}

?>

在上面的例子中， Eloquent 会尝试匹配 Phone 模型上的 user\_id 至 User 模型上的 id 。它是通过检查关系方法的名称并使用 \_id 作为后缀名来确定默认外键名称的。但是，如果 Phone 模型的外键不是 user\_id，那么可以将自定义键名作为第二个参数传递给 belongsTo 方法。

/\*\*

\* 获得拥有此电话的用户。

\*/

public function user()

{

return $this->belongsTo('App\User', 'foreign\_key');

}

如果父级模型没有使用 id 作为主键，或者是希望用不同的字段来连接子级模型，则可以通过给 belongsTo 方法传递第三个参数的形式指定父级数据表的自定义键。

/\*\*

\* 获得拥有此电话的用户

\*/

public function user()

{

return $this->belongsTo('App\User', 'foreign\_key', 'other\_key');

}

(2)、一对多

『一对多』关联用于定义单个模型拥有任意数量的其它关联模型。例如，一篇博客文章可能会有无限多条评论。正如其它所有的 Eloquent 关联一样，一对多关联的定义也是在 Eloquent 模型中写一个方法。

<?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');

}

}

?>

记住一点，Eloquent 将会自动确定 Comment 模型的外键属性。按照约定，Eloquent 将会使用所属模型名称的『snake case』形式，再加上 \_id 后缀作为外键字段。因此，在上面这个例子中，Eloquent 将假定 Comment 对应到 Post 模型上的外键就是 post\_id。

一旦关系被定义好以后，就可以通过访问 Post 模型的 comments 属性来获取评论的集合。记住，由于 Eloquent 提供了『动态属性』，所以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，由于所有的关联还可以作为查询语句构造器使用，因此你可以使用链式调用的方式，在 comments 方法上添加额外的约束条件。

$comment = App\Post::find(1)->comments()->where('title', 'foo')->first();

正如 hasOne 方法一样，你也可以在使用 hasMany 方法的时候，通过传递额外参数来覆盖默认使用的外键与本地键。

return $this->hasMany('App\Comment', 'foreign\_key');

return $this->hasMany('App\Comment', 'foreign\_key', 'local\_key');

(3)、一对多 (反向)

现在，我们已经能获得一篇文章的所有评论，接着再定义一个通过评论获得所属文章的关联关系。这个关联是 hasMany 关联的反向关联，需要在子级模型中使用 belongsTo 方法定义它。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model

{

/\*\*

\* 获取此评论所属文章

\*/

public function post()

{

return $this->belongsTo('App\Post');

}

}

?>

这个关系定义好以后，我们就可以通过访问 Comment 模型的 post 这个『动态属性』来获取关联的 Post 模型了。

$comment = App\Comment::find(1);

echo $comment->post->title;

在上面的例子中，Eloquent 会尝试用 Comment 模型的 post\_id 与 Post 模型的 id 进行匹配。默认外键名是 Eloquent 依据关联名，并在关联名后加上 \_再加上主键字段名作为后缀确定的。当然，如果 Comment 模型的外键不是 post\_id，那么可以将自定义键名作为第二个参数传递给 belongsTo 方法。

/\*\*

\* 获取此评论所属文章

\*/

public function post()

{

return $this->belongsTo('App\Post', 'foreign\_key');

}

如果父级模型没有使用 id 作为主键，或者是希望用不同的字段来连接子级模型，则可以通过给 belongsTo 方法传递第三个参数的形式指定父级数据表的自定义键。

/\*\*

\* 获取此评论所属文章

\*/

public function post()

{

return $this->belongsTo('App\Post', 'foreign\_key', 'other\_key');

}

(4)、多对多

多对多关联比 hasOne 和 hasMany 关联稍微复杂些。举个例子，一个用户可以拥有很多种角色，同时这些角色也被其他用户共享。例如，许多用户可能都有 「管理员」 这个角色。要定义这种关联，需要三个数据库表： users，roles 和 role\_user。role\_user 表的命名是由关联的两个模型按照字母顺序来的，并且包含了 user\_id 和 role\_id 字段。

多对多关联通过调用 belongsToMany 这个内部方法返回的结果来定义，例如，我们在 User 模型中定义 roles 方法。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

/\*\*

\* 用户拥有的角色

\*/

public function roles()

{

return $this->belongsToMany('App\Role');

}

}

?>

一旦关联关系被定义后，你可以通过 roles 动态属性获取用户角色。

$user = App\User::find(1);

foreach ($user->roles as $role) {

//

}

当然，像其它所有关联模型一样，你可以使用 roles 方法，利用链式调用对查询语句添加约束条件。

$roles = App\User::find(1)->roles()->orderBy('name')->get();

正如前面所提到的，为了确定关联连接表的表名，Eloquent 会按照字母顺序连接两个关联模型的名字。当然，你也可以不使用这种约定，传递第二个参数到 belongsToMany 方法即可。

return $this->belongsToMany('App\Role', 'role\_user');

除了自定义连接表的表名，你还可以通过传递额外的参数到 belongsToMany 方法来定义该表中字段的键名。第三个参数是定义此关联的模型在连接表里的外键名，第四个参数是另一个模型在连接表里的外键名。

return $this->belongsToMany('App\Role', 'role\_user', 'user\_id', 'role\_id');

//

///1/ 定义反向关联

要定义多对多的反向关联， 你只需要在关联模型中调用 belongsToMany 方法。我们在 Role 模型中定义 users 方法。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Role extends Model

{

/\*\*

\* 拥有此角色的用户

\*/

public function users()

{

return $this->belongsToMany('App\User');

}

}

?>

如你所见，除了引入模型为 App\User 外，其它与在 User 模型中定义的完全一样。由于我们重用了 belongsToMany 方法，自定义连接表表名和自定义连接表里的键的字段名称在这里同样适用。

//

///2/ 获取中间表字段

就如你刚才所了解的一样，多对多的关联关系需要一个中间表来提供支持， Eloquent 提供了一些有用的方法来和这张表进行交互。例如，假设我们的 User 对象关联了多个 Role 对象。在获得这些关联对象后，可以使用模型的 pivot 属性访问中间表的数据。

$user = App\User::find(1);

foreach ($user->roles as $role) {

echo $role->pivot->created\_at;

}

需要注意的是，我们获取的每个 Role 模型对象，都会被自动赋予 pivot 属性，它代表中间表的一个模型对象，并且可以像其他的 Eloquent 模型一样使用。

默认情况下，pivot 对象只包含两个关联模型的主键，如果你的中间表里还有其他额外字段，你必须在定义关联时明确指出。

return $this->belongsToMany('App\Role')->withPivot('column1', 'column2');

如果你想让中间表自动维护 created\_at 和 updated\_at 时间戳，那么在定义关联时附加上 withTimestamps 方法即可。

return $this->belongsToMany('App\Role')->withTimestamps();

//

///3/ 自定义 pivot 属性名称

如前所述，来自中间表的属性可以使用 pivot 属性访问。但是，你可以自由定制此属性的名称，以便更好的反应其在应用中的用途。

例如，如果你的应用中包含可能订阅的用户，则用户与博客之间可能存在多对多的关系。如果是这种情况，你可能希望将中间表访问器命名为 subscription 取代 pivot 。这可以在定义关系时使用 as 方法完成。

return $this->belongsToMany('App\Podcast')

->as('subscription')

->withTimestamps();

一旦定义完成，你可以使用自定义名称访问中间表数据。

$users = User::with('podcasts')->get();

foreach ($users->flatMap->podcasts as $podcast) {

echo $podcast->subscription->created\_at;

}

//

///4/ 通过中间表过滤关系

在定义关系时，你还可以使用 wherePivot 和 wherePivotIn 方法来过滤 belongsToMany 返回的结果。

return $this->belongsToMany('App\Role')->wherePivot('approved', 1);

return $this->belongsToMany('App\Role')->wherePivotIn('priority', [1, 2]);

(5)、自定义中间表模型 (定义中间表模型)

如果你想定义一个自定义模型来表示关联关系中的中间表，可以在定义关联时调用 using 方法。自定义多对多中间表模型都必须扩展自 Illuminate\Database\Eloquent\Relations\Pivot 类，自定义多对多（多态）中间表模型必须继承 Illuminate\Database\Eloquent\Relations\MorphPivot 类。例如，我们在写 Role 模型的关联时，使用自定义中间表模型 RoleUser。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Role extends Model

{

/\*\*

\* 拥有此角色的所有用户

\*/

public function users()

{

return $this->belongsToMany('App\User')->using('App\RoleUser');

}

}

?>

当定义 RoleUser 模型时，我们要扩展 Pivot 类。

<?php

namespace App;

use Illuminate\Database\Eloquent\Relations\Pivot;

class RoleUser extends Pivot

{

//

}

?>

你可以组合使用 using 和 withPivot 从中间表来检索列。例如，通过将列名传递给 withPivot 方法，就可以从 UserRole 中间表中检索出 created\_by 和 updated\_by 两列数据。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Role extends Model

{

/\*\*

\* 拥有此角色的用户。

\*/

public function users()

{

return $this->belongsToMany('App\User')

->using('App\RoleUser')

->withPivot([

'created\_by',

'updated\_by',

]);

}

}

?>

注意: Pivot 模型可能不使用 SoftDeletes 特性。 如果您需要软删除数据关联记录，请考虑将您的数据关联模型转换为实际的 Eloquent 模型。

//

///1/ 带有递增 ID 的自定义中继模型

如果你用一个自定义的中继模型定义了多对多的关系，而且这个中继模型拥有一个自增的主键，你应当确保这个自定义中继模型类中定义了一个 incrementing 属性其值为 true。

/\*\*

\* 标识 ID 是否自增。

\*

\* @var bool

\*/

public $incrementing = true;

(6)、远程一对一 (远程一对一关系)

远程一对一关联通过一个中间关联模型实现。

例如，如果每个供应商都有一个用户，并且每个用户与一个用户历史记录相关联，那么供应商可以通过用户访问用户的历史记录，让我们看看定义这种关系所需的数据库表。

~~~ s ~~~

users

id - integer

supplier\_id - integer

suppliers

id - integer

history

id - integer

user\_id - integer

~~~ e ~~~

虽然 history 表不包含 supplier\_id ，但 hasOneThrough 关系可以提供对用户历史记录的访问，以访问供应商模型。现在我们已经检查了关系的表结构，让我们在 Supplier 模型上定义相应的方法。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Supplier extends Model

{

/\*\*

\* 用户的历史记录。

\*/

public function userHistory()

{

return $this->hasOneThrough('App\History', 'App\User');

}

}

?>

传递给 hasOneThrough 方法的第一个参数是希望访问的模型名称，第二个参数是中间模型的名称。

当执行关联查询时，通常会使用 Eloquent 约定的外键名。如果你想要自定义关联的键，可以通过给 hasOneThrough 方法传递第三个和第四个参数实现，第三个参数表示中间模型的外键名，第四个参数表示最终模型的外键名。第五个参数表示本地键名，而第六个参数表示中间模型的本地键名。

class Supplier extends Model

{

/\*\*

\* 用户的历史记录。

\*/

public function userHistory()

{

return $this->hasOneThrough(

'App\History',

'App\User',

'supplier\_id', // 用户表外键

'user\_id', // 历史记录表外键

'id', // 供应商本地键

'id' // 用户本地键

);

}

}

(7)、远程一对多 (远程一对多关联)

远程「一对多」关联提供了方便、简短的方式通过中间的关联来获得远层的关联。例如，一个 Country 模型可以通过中间的 User 模型获得多个 Post 模型。在这个例子中，你可以轻易地收集给定国家的所有博客文章。让我们来看看定义这种关联所需的数据表。

~~~ s ~~~

countries

id - integer

name - string

users

id - integer

country\_id - integer

name - string

posts

id - integer

user\_id - integer

title - string

~~~ e ~~~

虽然 posts 表中不包含 country\_id 字段，但 hasManyThrough 关联能让我们通过 $country->posts 访问到一个国家下所有的用户文章。为了完成这个查询，Eloquent 会先检查中间表 users 的 country\_id 字段，找到所有匹配的用户 ID 后，使用这些 ID，在 posts 表中完成查找。

现在，我们已经知道了定义这种关联所需的数据表结构，接下来，让我们在 Country 模型中定义它。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Country extends Model

{

/\*\*

\* 当前国家所有文章。

\*/

public function posts()

{

return $this->hasManyThrough('App\Post', 'App\User');

}

}

?>

hasManyThrough 方法的第一个参数是我们最终希望访问的模型名称，而第二个参数是中间模型的名称。

当执行关联查询时，通常会使用 Eloquent 约定的外键名。如果你想要自定义关联的键，可以通过给 hasManyThrough 方法传递第三个和第四个参数实现，第三个参数表示中间模型的外键名，第四个参数表示最终模型的外键名。第五个参数表示本地键名，而第六个参数表示中间模型的本地键名。

class Country extends Model

{

public function posts()

{

return $this->hasManyThrough(

'App\Post',

'App\User',

'country\_id', // 国家表外键

'user\_id', // 用户表外键

'id', // 国家表本地键

'id' // 用户表本地键

);

}

}

?>

--

3、多态关联

多态关联允许目标模型借助单个关联从属于多个模型。

(1)、一对一 (多态)

///1/ 表结构

一对一多态关联与简单的一对一关联类似；不过，目标模型能够在一个关联上从属于多个模型。例如，博客 Post 和 User 可能共享一个关联到 Image 模型的关系。使用一对一多态关联允许使用一个唯一图片列表同时用于博客文章和用户账户。让我们先看看表结构。

~~~ s ~~~

posts

id - integer

name - string

users

id - integer

name - string

images

id - integer

url - string

imageable\_id - integer //users\_id/posts\_id;

imageable\_type - string //users/posts类型;

~~~ e ~~~

要特别留意 images 表的 imageable\_id 和 imageable\_type 列。 imageable\_id 列包含文章或用户的 ID 值，而 imageable\_type 列包含的则是父模型的类名。Eloquent 在访问 imageable 时使用 imageable\_type 列来判断父模型的 「类型」。

//

///2/ 模型结构

接下来，再看看建立关联的模型定义。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Image extends Model

{

/\*\*

\* 获取拥有此图片的模型。

\*/

public function imageable()

{

return $this->morphTo();

}

}

class Post extends Model

{

/\*\*

\* 获取文章图片。

\*/

public function image()

{

return $this->morphOne('App\Image', 'imageable');

}

}

class User extends Model

{

/\*\*

\* 获取用户图片。

\*/

public function image()

{

return $this->morphOne('App\Image', 'imageable');

}

}

?>

//

///3/ 获取关联

一旦定义了表和模型，就可以通过模型访问此关联。比如，要获取文章图片，可以使用 image 动态属性。

$post = App\Post::find(1);

$image = $post->image;

还可以通过访问执行 morphTo 调用的方法名来从多态模型中获知父模型。在这个例子中，就是 Image 模型的 imageable 方法。所以，我们可以像动态属性那样访问这个方法。

$image = App\Image::find(1);

$imageable = $image->imageable;

Image 模型的 imageable 关联将返回 Post 或 User 实例，其结果取决于图片属性哪个模型。

(2)、一对多 (多态)

///1/ 表结构

一对多多态关联与简单的一对多关联类似；不过，目标模型可以在一个关联中从属于多个模型。假设应用中的用户可以同时 「评论」 文章和视频。使用多态关联，可以用单个 comments 表同时满足这些情况。我们还是先来看看用来构建这种关联的表结构。

~~~ s ~~~

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

~~~ e ~~~

//

///2/ 模型结构

接下来，看看构建这种关联的模型定义。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model

{

/\*\*

\* 获取拥有此评论的模型。

\*/

public function commentable()

{

return $this->morphTo();

}

}

class Post extends Model

{

/\*\*

\* 获取此文章的所有评论。

\*/

public function comments()

{

return $this->morphMany('App\Comment', 'commentable');

}

}

class Video extends Model

{

/\*\*

\* 获取此视频的所有评论。

\*/

public function comments()

{

return $this->morphMany('App\Comment', 'commentable');

}

}

?>

//

///3/ 获取关联

一旦定义了数据库表和模型，就可以通过模型访问关联。例如，可以使用 comments 动态属性访问文章的全部评论。

$post = App\Post::find(1);

foreach ($post->comments as $comment) {

//

}

还可以通过访问执行 morphTo 调用的方法名来从多态模型获取其所属模型。在本例中，就是 Comment 模型的 commentable 方法。

$comment = App\Comment::find(1);

$commentable = $comment->commentable;

Comment 模型的 commentable 关联将返回 Post 或 Video 实例，其结果取决于评论所属的模型。

(3)、多对多 (多态)

//

///1/ 表结构

多对多多态关联比 morphOne 和 morphMany 关联略微复杂一些。例如，博客 Post 和 Video 模型能够共享关联到 Tag 模型的多态关系。使用多对多多态关联允许使用一个唯一标签在博客文章和视频间共享。以下是多对多多态关联的表结构。

~~~ s ~~~

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

~~~ e ~~~

//

///2/ 模型结构

接下来，在模型上定义关联。Post 和 Video 模型都有调用 Eloquent 基类上 morphToMany 方法的 tags 方法。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model

{

/\*\*

\* 获取文章的所有标签。

\*/

public function tags()

{

return $this->morphToMany('App\Tag', 'taggable');

}

}

?>

//

///3/ 定义反向关联关系

下面，需要在 Tag 模型上为每个关联模型定义一个方法。在这个示例中，我们将会定义 posts 方法和 videos 方法。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Tag extends Model

{

/\*\*

\* 获取被打上此标签的所有文章。

\*/

public function posts()

{

return $this->morphedByMany('App\Post', 'taggable');

}

/\*\*

\* 获取被打上此标签的所有视频。

\*/

public function videos()

{

return $this->morphedByMany('App\Video', 'taggable');

}

}

?>

//

///4/ 获取关联

一旦定义了数据库表和模型，就可以通过模型访问关联。例如，可以使用 tags 动态属性访问文章的所有标签。

$post = App\Post::find(1);

foreach ($post->tags as $tag) {

//

}

还可以访问执行 morphedByMany 方法调用的方法名来从多态模型获取其所属模型。在这个示例中，就是 Tag 模型的 posts 或 videos 方法。可以像动态属性一样访问这些方法。

$tag = App\Tag::find(1);

foreach ($tag->videos as $video) {

//

}

(4)、自定义多态模型 (自定义多态类型)

默认情况下， Laravel 使用完全限定类名存储关联模型类型。在上面的一对多示例中， 因为 Comment 可能从属于一个 Post 或一个 Video，默认的 commentable\_type 就将分别是 App\Post 或 App\Video。不过，你可能希望数据库与应用的内部结构解耦。在这种情况下，可以定义一个 「morph 映射」 来通知 Eloquent 使用自定义名称代替对应的类名。

use Illuminate\Database\Eloquent\Relations\Relation;

Relation::morphMap([

'posts' => 'App\Post',

'videos' => 'App\Video',

]);

可以在 AppServiceProvider 的 boot 函数中注册 morphMap，或者创建一个单独的服务提供者。

注意：在现有应用程序中添加「morph 映射」时，数据库中仍包含完全限定类的每个可变形 \*\_type 列值都需要转换为其「映射」名称。

--

4、查询关联

由于 Eloquent 关联的所有类型都通过方法定义，你可以调用这些方法，而无需真实执行关联查询。另外，所有 Eloquent 关联类型用作 查询构造器( https://learnku.com/docs/laravel/6.x/queries )，允许你在数据库上执行 SQL 之前，持续通过链式调用添加约束。

例如，假设一个博客系统的 User 模型有许多关联的 Post 模型。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

/\*\*

\* 获取该用户的所有文章。

\*/

public function posts()

{

return $this->hasMany('App\Post');

}

}

?>

你可以查询 posts 关联，并为其添加额外的约束。

$user = App\User::find(1);

$user->posts()->where('active', 1)->get();

你可以在关联上使用任意 查询构造器( https://learnku.com/docs/laravel/6.x/queries ) 方法，请查阅查询构造器文档，学习那些对你有用的方法。

//

///1/ 在关联之后链式添加 orWhere 条件

如上所示，你可以在查询关联时自由添加其他约束。 但是，在将 orWhere 子句链接到关联时要小心，因为 orWhere 子句将在逻辑上与关联约束处于同一级别。

$user->posts()

->where('active', 1)

->orWhere('votes', '>=', 100)

->get();

// select \* from posts

// where user\_id = ? and active = 1 or votes >= 100

在大多数情况下，你可以使用约束组( https://learnku.com/docs/laravel/6.x/queries#parameter-grouping ) 在括号中对条件检查进行逻辑分组。

use Illuminate\Database\Eloquent\Builder;

$user->posts()

->where(function (Builder $query) {

return $query->where('active', 1)

->orWhere('votes', '>=', 100);

})

->get();

// select \* from posts

// where user\_id = ? and (active = 1 or votes >= 100)

(1)、关联方法 Vs. 动态属性

如果不需要向 Eloquent 关联查询添加额外的约束，可以像属性一样访问关联。例如，继续使用 User 和 Post 示例模型，可以这样访问用户的全部文章。

$user = App\User::find(1);

foreach ($user->posts as $post) {

//

}

动态属性是「懒加载」的，这意味着它们仅在你真实访问关联数据时才被载入。因此，开发者经常使用 预加载 ( https://learnku.com/docs/laravel/6.x/eloquent-relationships/5177#eager-loading ) 预先加载那些他们确知在载入模型后将访问的关联。对载入模型关联中必定被执行的 SQL 查询而言，预加载显著减少了查询的执行次数。

(2)、基于存在的关联查询 (查询已存在的关联)

在访问模型记录时，可能希望基于关联的存在限制查询结果。比如想要获取至少存在一条评论的所有文章，可以通过给 has 和 orHas 方法传递关联名称来实现。

// 获取至少存在一条评论的所有文章...

$posts = App\Post::has('comments')->get();

还可以指定运算符和数量进一步自定义查询。

// 获取评论超过三条的文章...

$posts = App\Post::has('comments', '>=', 3)->get();

还可以用 「点」语法构造嵌套的 has 语句。比如，可以获取拥有至少一条评论和投票的文章。

// 获取拥有至少一条带有投票评论的文章...

$posts = App\Post::has('comments.votes')->get();

如果需要更多功能，可以使用 whereHas 和 orWhereHas 方法将「where」 条件放到 has 查询上。这些方法允许你向关联加入自定义约束，比如检查评论内容。

use Illuminate\Database\Eloquent\Builder;

//

// 获取至少带有一条评论内容包含 foo% 关键词的文章...

$posts = App\Post::whereHas('comments', function (Builder $query) {

$query->where('content', 'like', 'foo%');

})->get();

//

// 获取至少带有十条评论内容包含 foo% 关键词的文章...

$posts = App\Post::whereHas('comments', function (Builder $query) {

$query->where('content', 'like', 'foo%');

}, '>=', 10)->get();

(3)、基于不存在的关联查询 (查询不存在的关联)

在访问模型记录时，可能希望基于关联不存在来限制查询结果。假设想要获取不存在任何评论的文章，可以通过向 doesntHave 和 orDoesntHave 方法传递关联名称来实现。

$posts = App\Post::doesntHave('comments')->get();

如果需要更多功能，可以使用 whereDoesntHave 和 orWhereDoesntHave 方法将「where」 条件加到 doesntHave 查询上。这些方法允许你向关联加入自定义限制，比如检测评论内容。

use Illuminate\Database\Eloquent\Builder;

$posts = App\Post::whereDoesntHave('comments', function (Builder $query) {

$query->where('content', 'like', 'foo%');

})->get();

还可以使用 「点」 语法执行嵌套关联查询。例如，下面的查询用于获取带有没被禁用的作者发表评论的文章。

use Illuminate\Database\Eloquent\Builder;

$posts = App\Post::whereDoesntHave('comments.author', function (Builder $query) {

$query->where('banned', 1);

})->get();

(4)、多态的关联查询 (多态关联查询)

要查询 MorphTo 关联的存在，可以使用 whereHasMorph 方法及其相应的方法。

use Illuminate\Database\Eloquent\Builder;

//

// 查询与帖子或视频相关并且标题包含 foo 的评论...

$comments = App\Comment::whereHasMorph(

'commentable',

['App\Post', 'App\Video'],

function (Builder $query) {

$query->where('title', 'like', 'foo%');

}

)->get();

//

// 查询与帖子相关的评论，标题不像 foo％...

$comments = App\Comment::whereDoesntHaveMorph(

'commentable',

'App\Post',

function (Builder $query) {

$query->where('title', 'like', 'foo%');

}

)->get();

你可以使用 $type 参数根据相关模型添加不同的约束。

use Illuminate\Database\Eloquent\Builder;

$comments = App\Comment::whereHasMorph(

'commentable',

['App\Post', 'App\Video'],

function (Builder $query, $type) {

$query->where('title', 'like', 'foo%');

if ($type === 'App\Post') {

$query->orWhere('content', 'like', 'foo%');

}

}

)->get();

您可以提供 \* 作为通配符，让 Laravel 从数据库中查询所有可能的多态类型，而不是传递可能的多态模型数组。 Laravel 将执行其他查询以执行此操作。

use Illuminate\Database\Eloquent\Builder;

$comments = App\Comment::whereHasMorph('commentable', '\*', function (Builder $query) {

$query->where('title', 'like', 'foo%');

})->get();

(5)、关联数据计数 (关联模型计数)

如果想要只计算关联结果的统计数量而不需要真实加载它们，可以使用 withCount 方法，它将放在结果模型的 {relation}\_count 列。示例如下。

$posts = App\Post::withCount('comments')->get();

foreach ($posts as $post) {

echo $post->comments\_count;

}

可以像给查询添加限制一样为多个关系添加「计数」。

use Illuminate\Database\Eloquent\Builder;

$posts = App\Post::withCount(['votes', 'comments' => function (Builder $query) {

$query->where('content', 'like', 'foo%');

}])->get();

echo $posts[0]->votes\_count;

echo $posts[0]->comments\_count;

还可以给关联计数结果起别名，这允许你在同一关联上添加多个计数。

use Illuminate\Database\Eloquent\Builder;

$posts = App\Post::withCount([

'comments',

'comments as pending\_comments\_count' => function (Builder $query) {

$query->where('approved', false);

},

])->get();

echo $posts[0]->comments\_count;

echo $posts[0]->pending\_comments\_count;

如果将 withCount 和 select 查询组装在一起，请确保在 select 方法之后调用 withCount。

$posts = App\Post::select(['title', 'body'])->withCount('comments')->get();

echo $posts[0]->title;

echo $posts[0]->body;

echo $posts[0]->comments\_count;

--

5、预加载

当以属性方式访问 Eloquent 关联时，关联数据「懒加载」。这意味着直到第一次访问属性时关联数据才会被真实加载。不过 Eloquent 能在查询父模型时「预先载入」子关联。预加载可以缓解 N + 1 查询问题。为了说明 N + 1 查询问题，考虑 Book 模型关联到 Author 的情形。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Book extends Model

{

/\*\*

\* 获取书籍作者。

\*/

public function author()

{

return $this->belongsTo('App\Author');

}

}

?>

现在，我们来获取所有的书籍及其作者。

$books = App\Book::all();

foreach ($books as $book) {

echo $book->author->name;

}

此循环将执行一个查询，用于获取全部书籍，然后为每本书执行获取作者的查询。如果我们有 25 本书，此循环将运行 26 个查询：1 个用于查询书籍，25 个附加查询用于查询每本书的作者。

谢天谢地，我们能够使用预加载将操作压缩到只有 2 个查询。在查询时，可以使用 with 方法指定想要预加载的关联。

$books = App\Book::with('author')->get();

foreach ($books as $book) {

echo $book->author->name;

}

在这个例子中，仅执行了两个查询。

select \* from books

select \* from authors where id in (1, 2, 3, 4, 5, ...)

(1)、预加载多个关联

有时，你可能需要在单一操作中预加载几个不同的关联。要达成此目的，只要向 with 方法传递多个关联名称构成的数组参数。

$books = App\Book::with(['author', 'publisher'])->get();

(2)、嵌套预加载

可以使用 「点」 语法预加载嵌套关联。比如在一个 Eloquent 语句中预加载所有书籍作者及其联系方式。

$books = App\Book::with('author.contacts')->get();

(3)、嵌套预加载 morphTo 关联

如果你希望加载一个 morphTo 关系，以及该关系可能返回的各种实体的嵌套关系，可以将 with 方法与 morphTo 关系的 morphWith 方法结合使用。 为了帮助说明这种方法，让我们考虑以下模型。

<?php

use Illuminate\Database\Eloquent\Model;

class ActivityFeed extends Model

{

/\*\*

\* 获取活动提要记录的父级。

\*/

public function parentable()

{

return $this->morphTo();

}

}

?>

在这个例子中，我们假设 Event、Photo 和 Post 模型可以创建 ActivityFeed 模型。 另外，我们假设 Event 模型属于 Calendar 模型，Photo 模型与 Tag 模型相关联，Post 模型属于 Author 模型。

使用这些模型定义和关联，我们可以查询 ActivityFeed 模型实例并预加载所有 parentable 模型及其各自的嵌套关系。

use Illuminate\Database\Eloquent\Relations\MorphTo;

$activities = ActivityFeed::query()

->with(['parentable' => function (MorphTo $morphTo) {

$morphTo->morphWith([

Event::class => ['calendar'],

Photo::class => ['tags'],

Post::class => ['author'],

]);

}])->get();

(4)、预加载指定列

并不是总需要获取关系的每一列。在这种情况下，Eloquent 允许你为关联指定想要获取的列。

$books = App\Book::with('author:id,name')->get();

注意：在使用这个特性时，一定要在要获取的列的列表中包含 id 列。

(4)、默认预加载

有时可能希望在查询模型时始终加载某些关联。 为此，你可以在模型上定义 $with 属性。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Book extends Model

{

/\*\*

\* 默认加载的关联。

\*

\* @var array

\*/

protected $with = ['author'];

/\*\*

\* 获取书籍作者

\*/

public function author()

{

return $this->belongsTo('App\Author');

}

}

?>

如果你想从单个查询的 $with 属性中删除一个预加载，你可以使用 without 方法。

$books = App\Book::without('author')->get();

(5)、为预加载添加约束

有时，可能希望预加载一个关联，同时为预加载查询添加额外查询条件，就像下面的例子。

$users = App\User::with(['posts' => function ($query) {

$query->where('title', 'like', '%first%');

}])->get();

在这个例子中， Eloquent 将仅预加载那些 title 列包含 first 关键词的文章。也可以调用其它的 查询构造器 ( https://learnku.com/docs/laravel/6.x/queries ) 方法进一步自定义预加载操作。

$users = App\User::with(['posts' => function ($query) {

$query->orderBy('created\_at', 'desc');

}])->get();

注意：在约束预加载时，不能使用 limit 和 take 查询构造器方法。

(6)、延迟预加载

有可能你还希望在模型加载完成后在进行渴求式加载。举例来说，如果你想要根据某个条件动态决定是否加载关联数据，那么 load 方法对你来说会非常有用。

$books = App\Book::all();

if ($someCondition) {

$books->load('author', 'publisher');

}

如果你想要在渴求式加载的查询语句中进行条件约束，你可以通过数组的形式去加载，键为对应的关联关系，值为 Closure 闭包函数，该闭包的参数为一个 query 实例。

$books->load(['author' => function ($query) {

$query->orderBy('published\_date', 'asc');

}]);

如果希望加载还没有加载的关联关系时，你可以使用 loadMissing 方法。

public function format(Book $book)

{

$book->loadMissing('author');

return [

'name' => $book->name,

'author' => $book->author->name,

];

}

(7)、嵌套延迟预加载 & morphTo

如果希望快速加载 morphTo 关系，以及该关系可能返回的各种实体上的嵌套关系，可以使用 loadMorph 方法。

这个方法接受 morphTo 关系的名称作为它的第一个参数，第二个参数接收模型数组、关系数组。为了帮助说明这个方法，可以看一下以下模型例子。

<?php

use Illuminate\Database\Eloquent\Model;

class ActivityFeed extends Model

{

/\*\*

\* 获取 activity feed 记录的父级。

\*/

public function parentable()

{

return $this->morphTo();

}

}

?>

在这个例子中，让我们假设 Event 、Photo 和 Post 模型可以创建 ActivityFeed 模型。此外，让我们假设 Event 模型属于 Calendar 模型，Photo 模型与 Tag 模型相关联，Post 模型属于 Author 模型。

使用这些模型定义和关系，我们可以检索 ActivityFeed 模型实例，并立即加载所有 parentable 模型及其各自的嵌套关系。

$activities = ActivityFeed::with('parentable')

->get()

->loadMorph('parentable', [

Event::class => ['calendar'],

Photo::class => ['tags'],

Post::class => ['author'],

]);

--

6、插入 & 更新关联模型

(1)、save 方法 (保存方法)

Eloquent 为新模型添加关联提供了便捷的方法。例如，也许你需要添加一个新的 Comment 到一个 Post 模型中。你不用在 Comment 中手动设置 post\_id 属性，就可以直接使用关联模型的 save 方法将 Comment 直接插入。

$comment = new App\Comment(['message' => 'A new comment.']);

$post = App\Post::find(1);

$post->comments()->save($comment);

需要注意的是，我们并没有使用动态属性的方式访问 comments 关联。相反，我们调用 comments 方法来获得关联实例。save 方法将自动添加适当的 post\_id 值到 Comment 模型中。

如果你需要保存多个关联模型，你可以使用 saveMany 方法。

$post = App\Post::find(1);

$post->comments()->saveMany([

new App\Comment(['message' => 'A new comment.']),

new App\Comment(['message' => 'Another comment.']),

]);

//

///1/ 递归保存模型和关联数据

如果你想 save 你的模型及其所有关联数据，你可以使用 push 方法。

$post = App\Post::find(1);

$post->comments[0]->message = 'Message';

$post->comments[0]->author->name = 'Author Name';

$post->push();

(2)、create 方法 (新增方法)

除了 save 和 saveMany 方法外，你还可以使用 create 方法。它接受一个属性数组，同时会创建模型并插入到数据库中。 还有， save 方法和 create 方法的不同之处在于， save 方法接受一个完整的 Eloquent 模型实例，而 create 则接受普通的 PHP 数组。

$post = App\Post::find(1);

$comment = $post->comments()->create([

'message' => 'A new comment.',

]);

Tip：在使用 create 方法前，请务必确保查看过本文档的 批量赋值 ( https://learnku.com/docs/laravel/6.x/eloquent#mass-assignment ) 章节。

你还可以使用 createMany 方法去创建多个关联模型。

$post = App\Post::find(1);

$post->comments()->createMany([

[

'message' => 'A new comment.',

],

[

'message' => 'Another new comment.',

],

]);

你还可以使用 findOrNew、firstOrNew、firstOrCreate 和 updateOrCreate 方法来 创建和更新关系模型( https://learnku.com/docs/laravel/6.x/eloquent#other-creation-methods )。

(3)、更新 Belongs To 关联

当更新 belongsTo 关联时，可以使用 associate 方法。此方法将会在子模型中设置外键。

$account = App\Account::find(10);

$user->account()->associate($account);

$user->save();

当移除 belongsTo 关联时，可以使用 dissociate 方法。此方法会将关联外键设置为 null。

$user->account()->dissociate();

$user->save();

//

///1/ 默认模型

belongsTo，hasOne，hasOneThrough 和 morphOne 关系允许你指定默认模型，当给定关系为 null 时，将会返回默认模型。 这种模式被称作 Null 对象模式 ( https://en.wikipedia.org/wiki/Null\_Object\_pattern ) ，可以减少你代码中不必要的检查。在下面的例子中，如果发布的帖子没有找到作者， user 关系会返回一个空的 App\User 模型。

/\*\*

\* 获取帖子的作者。

\*/

public function user()

{

return $this->belongsTo('App\User')->withDefault();

}

如果需要在默认模型里添加属性， 你可以传递数组或者回调方法到 withDefault 中。

/\*\*

\* 获取帖子的作者。

\*/

public function user()

{

return $this->belongsTo('App\User')->withDefault([

'name' => 'Guest Author',

]);

}

/\*\*

\* 获取帖子的作者。

\*/

public function user()

{

return $this->belongsTo('App\User')->withDefault(function ($user, $post) {

$user->name = 'Guest Author';

});

}

(4)、多对多关联

//

///1/ 附加 / 分离

Eloquent 也提供了一些额外的辅助方法，使相关模型的使用更加方便。例如，我们假设一个用户可以拥有多个角色，并且每个角色都可以被多个用户共享。给某个用户附加一个角色是通过向中间表插入一条记录实现的，可以使用 attach 方法完成该操作。

$user = App\User::find(1);

$user->roles()->attach($roleId);

在将关系附加到模型时，还可以传递一组要插入到中间表中的附加数据。

$user->roles()->attach($roleId, ['expires' => $expires]);

当然，有时也需要移除用户的角色。可以使用 detach 移除多对多关联记录。detach 方法将会移除中间表对应的记录；但是这 2 个模型都将会保留在数据库中。

// 移除用户的一个角色...

$user->roles()->detach($roleId);

//

// 移除用户的所有角色...

$user->roles()->detach();

//

为了方便，attach 和 detach 也允许传递一个 ID 数组。

$user = App\User::find(1);

$user->roles()->detach([1, 2, 3]);

$user->roles()->attach([

1 => ['expires' => $expires],

2 => ['expires' => $expires],

]);

//

///2/ 同步关联

你也可以使用 sync 方法构建多对多关联。sync 方法接收一个 ID 数组以替换中间表的记录。中间表记录中，所有未在 ID 数组中的记录都将会被移除。所以该操作结束后，只有给出数组的 ID 会被保留在中间表中。

$user->roles()->sync([1, 2, 3]);

你也可以通过 ID 传递额外的附加数据到中间表。

$user->roles()->sync([1 => ['expires' => true], 2, 3]);

如果你不想移除现有的 ID，可以使用 syncWithoutDetaching 方法。

$user->roles()->syncWithoutDetaching([1, 2, 3]);

//

///3/ 切换关联

多对多关联也提供了 toggle 方法用于「切换」给定 ID 数组的附加状态。 如果给定的 ID 已被附加在中间表中，那么它将会被移除，同样，如果如果给定的 ID 已被移除，它将会被附加

$user->roles()->toggle([1, 2, 3]);

//

///4/ 在中间表上保存额外的数据

当处理多对多关联时，save 方法接收一个额外的数据数组作为第二个参数。

App\User::find(1)->roles()->save($role, ['expires' => $expires]);

//

///5/ 更新中间表记录

如果你需要在中间表中更新一条已存在的记录，可以使用 updateExistingPivot 。此方法接收中间表的外键与要更新的数据数组进行更新。

$user = App\User::find(1);

$user->roles()->updateExistingPivot($roleId, $attributes);

--

7、更新父集时间戳

当一个模型属 belongsTo 或者 belongsToMany 另一个模型时， 例如 Comment 属于 Post，有时更新子模型导致更新父模型时间戳非常有用。例如，当 Comment 模型被更新时，您要自动「触发」父级 Post 模型的 updated\_at 时间戳的更新。Eloquent 让它变得简单。只要在子模型加一个包含关联名称的 touches 属性即可。

<?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'];

/\*\*

\* 评论所属的文章

\*/

public function post()

{

return $this->belongsTo('App\Post');

}

}

?>

现在，当你更新一个 Comment 时，对应父级 Post 模型的 updated\_at 字段同时也会被更新，使其更方便得知何时让一个 Post 模型的缓存失效。

$comment = App\Comment::find(1);

$comment->text = 'Edit to this comment!';

$comment->save();

--

--

三、Eloquent 集合 (Eloquent：集合)

--

https://learnku.com/docs/laravel/6.x/eloquent-collections/5178

--

1、简介

Eloquent 返回的所有结果集都是 Illuminate\Database\Eloquent\Collection 对象的实例，包括通过 get 方法检索或通过访问关联关系获取到的结果。 Eloquent 的集合对象继承了 Laravel 的 集合基类( https://learnku.com/docs/laravel/6.x/collections )，因此它自然也继承了数十种能优雅地处理 Eloquent 模型底层数组的方法。

而且，所有的集合都可以作为迭代器，你可以像遍历简单的 PHP 数组一样来遍历它们。

$users = App\User::where('active', 1)->get();

foreach ($users as $user) {

echo $user->name;

}

不过，集合比数组更加强大，它通过更加直观的接口暴露出可链式调用的 map /reduce 等操作。例如，让我们移除所有未激活的用户并收集剩余用户的名字。

$users = App\User::all();

$names = $users->reject(function ($user) {

return $user->active === false;

})

->map(function ($user) {

return $user->name;

});

注意：大多数 Eloquent 集合方法会返回新的 Eloquent 集合实例，但是 pluck, keys, zip, collapse, flatten 和 flip 方法除外，它们会返回一个 集合基类 ( https://learnku.com/docs/laravel/6.x/collections ) 实例。同样，如果 map 操作返回的集合不包括任何 Eloquent 模型，那么它会被自动转换成集合基类。

--

2、可用的方法 (可用方法)

所有 Eloquent 的集合都继承了 Laravel Collection ( https://learnku.com/docs/laravel/6.x/collections#available-methods ) 对象；因此，他们也继承了所有集合基类提供的强大的方法。

另外，Illuminate\Database\Eloquent\Collection 类提供了一套上层的方法来帮你管理你的模型集合。大多数方法返回 Illuminate\Database\Eloquent\Collection 实例； 然而，也会有一些方法返回基于 Illuminate\Support\Collection 类的实例。

(1)、contains($key, $operator = null, $value = null)

contains 方法可用于判断集合中是否包含指定的模型实例。这个方法接收一个主键或者模型实例。

$users->contains(1);

$users->contains(User::find(1));

(2)、diff($items)

diff 方法返回不在给定集合中的所有模型。

use App\User;

$users = $users->diff(User::whereIn('id', [1, 2, 3])->get());

(3)、except($keys)

except 方法返回给定主键外的所有模型。

$users = $users->except([1, 2, 3]);

(4)、find($key)

find 方法查找给定主键的模型。如果 $key 是一个模型实例， find 将会尝试返回与主键匹配的模型。 如果 $key 是一个关联数组，find 将会使用 whereIn() 返回所有匹配 $keys 的模型。

$users = User::all();

$user = $users->find(1);

(5)、fresh($with = [])

fresh 方法从数据库检索集合中的每个模型的新实例。 此外，任何指定的关系都会被预先加载。

$users = $users->fresh();

$users = $users->fresh('comments');

(6)、intersect($items)

intersect 方法返回给定集合中也存在的所有模型。

use App\User;

$users = $users->intersect(User::whereIn('id', [1, 2, 3])->get());

(7)、load($relations)

load 方法会提前加载集合中所有模型的指定关系。

$users->load('comments', 'posts');

$users->load('comments.author');

(8)、loadMissing($relations)

如果尚未加载关系，loadMissing 方法将会为集合中的所有模型加载给定的关系。

$users->loadMissing('comments', 'posts');

$users->loadMissing('comments.author');

(9)、modelKeys()

modelKeys 方法返回集合中所有模型的主键。

$users->modelKeys(); // [1, 2, 3, 4, 5]

(10)、makeVisible($attributes)

makeVisible 方法显示集合中的每个模型的指定隐藏属性。

$users = $users->makeVisible(['address', 'phone\_number']);

(11)、makeHidden($attributes)

makeHidden 隐藏集合中的每个模型指定属性。

$users = $users->makeHidden(['address', 'phone\_number']);

(12)、only($keys)

only 方法返回给定主键的所有模型。

$users = $users->only([1, 2, 3]);

(13)、unique($key = null, $strict = false)

unique 方法返回集合中的所有唯一模型。并且删除与集合中另一个模型有相同主键的同一类型的所有模型。

$users = $users->unique();

--

3、自定义集合

如果你需要使用自定义 Collection 对象与自己的扩展方法一起使用， 可以在你的模型上重写 newCollection 方法。

<?php

namespace App;

use App\CustomCollection;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

/\*\*

\* 创建一个 Eloquent 集合实例.

\*

\* @param array $models

\* @return \Illuminate\Database\Eloquent\Collection

\*/

public function newCollection(array $models = [])

{

return new CustomCollection($models);

}

}

?>

一旦你定义了 newCollection 方法， 你将随时可以从你的自定义集合中接收一个 Collection 返回的一个模型实例。如果要为应用程序的每个模型使用自定义集合，就应该重写所有模型扩展的基类模型中 newCollection 方法。

--

--

四、修改器 (Eloquent: 修改器)

--

https://learnku.com/docs/laravel/6.x/eloquent-mutators/5179

--

1、简介

当你在 Eloquent 模型实例中获取或设置某些属性值的时候，访问器和修改器允许你对 Eloquent 属性值进行格式化。例如，你可能需要使用 Laravel 加密器 ( https://learnku.com/docs/laravel/6.x/encryption ) 来加密保存在数据库中的值，而在使用 Eloquent 模型访问该属性的时候自动进行解密其值。

除了自定义访问器和修改器外，Eloquent 也会自动将日期字段类型转换为 Carbon ( https://github.com/briannesbitt/Carbon ) 实例，或将 文本类型转换为 JSON( https://learnku.com/docs/laravel/6.x/eloquent-mutators/5179#attribute-casting )。

--

2、访问器 & 修改器

(1)、定义一个访问器

若要定义一个访问器， 则需在模型上创建一个 getFooAttribute 方法，要访问的 Foo 字段需使用「驼峰式」命名。 在这个示例中，我们将为 first\_name 属性定义一个访问器。当 Eloquent 尝试获取 first\_name 属性时，将自动调用此访问器。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

/\*\*

\* 获取用户的姓名.

\*

\* @param string $value

\* @return string

\*/

public function getFirstNameAttribute($value)

{

return ucfirst($value);

}

}

?>

如你所见，字段的原始值被传递到访问器中，允许你对它进行处理并返回结果。如果想获取被修改后的值，你可以在模型实例上访问 first\_name 属性。

$user = App\User::find(1);

$firstName = $user->first\_name;

当然，你也可以通过已有的属性值，使用访问器返回新的计算值。

/\*\*

\* 获取用户的姓名.

\*

\* @return string

\*/

public function getFullNameAttribute()

{

return "{$this->first\_name} {$this->last\_name}";

}

Tip：如果你需要将这些计算值添加到模型的数组 / JSON 中， 你需要追加它们 ( https://learnku.com/docs/laravel/6.x/eloquent-serialization#appending-values-to-json )。

(2)、定义一个修改器

若要定义一个修改器，则需在模型上面定义 setFooAttribute 方法。要访问的 Foo 字段使用「驼峰式」命名。让我们再来定义一个 first\_name 属性的修改器。当我们尝试在模型上设置 first\_name 属性值时，该修改器将被自动调用。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

/\*\*

\* 设置用户的姓名.

\*

\* @param string $value

\* @return void

\*/

public function setFirstNameAttribute($value)

{

$this->attributes['first\_name'] = strtolower($value);

}

}

?>

修改器会获取属性已经被设置的值，允许你修改并且将其值设置到 Eloquent 模型内部的 $attributes 属性上。举个例子，如果我们尝试将 first\_name 属性的值设置为 Sally。

$user = App\User::find(1);

$user->first\_name = 'Sally';

在这个例子中， setFirstNameAttribute 方法在调用的时候接受 Sally 这个值作为参数。接着修改器会应用 strtolower 函数并将处理的结果设置到内部的 $attributes 数组。

--

3、日期转换器

默认情况下，Eloquent 会将 created\_at 和 updated\_at 字段转换为 Carbon ( https://github.com/briannesbitt/Carbon ) 实例，它继承了 PHP 原生的 DateTime 类并提供了各种有用的方法。你可以通过设置模型的 $dates 属性来添加其他日期属性。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

/\*\*

\* 应该转换为日期格式的属性.

\*

\* @var array

\*/

protected $dates = [

'seen\_at',

];

}

?>

Tip：你可以通过将模型的公有属性 $timestamps 值设置为 false 来禁用默认的 created\_at 和 updated\_at 时间戳。

当某个字段是日期格式时，你可以将值设置为一个 UNIX 时间戳，日期时间 (Y-m-d) 字符串，或者 DateTime / Carbon 实例。日期值会被正确格式化并保存到你的数据库中。

$user = App\User::find(1);

$user->deleted\_at = now();

$user->save();

就如上面所说，当获取到的属性包含在 $dates 属性中时，都会自动转换为 Carbon 实例，允许你在属性上使用任意的 Carbon ( https://github.com/briannesbitt/Carbon ) 方法。

$user = App\User::find(1);

return $user->deleted\_at->getTimestamp();

(1)、日期格式

默认情况下，时间戳都将以 'Y-m-d H:i:s' 形式格式化。如果你需要自定义时间戳格式，可在模型中设置 $dateFormat 属性。这个属性决定了日期属性将以何种形式保存在数据库中，以及当模型序列化成数组或 JSON 时的格式。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model

{

/\*\*

\* 模型日期字段的保存格式.

\*

\* @var string

\*/

protected $dateFormat = 'U';

}

?>

--

4、属性类型转换

模型中的 $casts 属性提供了一个便利的方法来将属性转换为常见的数据类型。$casts 属性应是一个数组，且数组的键是那些需要被转换的属性名称，值则是你希望转换的数据类型。支持转换的数据类型有：integer，real，float，double，decimal:<digits>，string，boolean，object，array，collection，date，datetime，和 timestamp。当需要转换为 decimal 类型时，你需要定义小数位的个数，如：decimal:2。

示例， 让我们把以整数（0 或 1 ）形式存储在数据库中的 is\_admin 属性转成布尔值。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

/\*\*

\* 这个属性应该被转换为原生类型.

\*

\* @var array

\*/

protected $casts = [

// 把以整数（0 或 1 ）形式存储在数据库中的 is\_admin 属性转成布尔值。

'is\_admin' => 'boolean',

];

}

?>

现在当你访问 is\_admin 属性时，虽然保存在数据库里的值是一个整数类型，但是返回值总是会被转换成布尔值类型。

$user = App\User::find(1);

if ($user->is\_admin) {

//

}

(1)、数组 & JSON 转换

当你在数据库存储序列化的 JSON 的数据时，array 类型的转换非常有用。比如：如果你的数据库具有被序列化为 JSON 的 JSON 或 TEXT 字段类型，并且在 Eloquent 模型中加入了 array 类型转换，那么当你访问的时候就会自动被转换为 PHP 数组。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

/\*\*

\* 这个属性应该被转换为原生类型.

\*

\* @var array

\*/

protected $casts = [

'options' => 'array',

];

}

?>

一旦定义了转换，你访问 options 属性时他会自动从 JSON 类型反序列化为 PHP 数组。当你设置了 options 属性的值时，给定的数组也会自动序列化为 JSON 类型存储。

$user = App\User::find(1);

$options = $user->options;

$options['key'] = 'value';

$user->options = $options;

$user->save();

(2)、Date 转换

当使用 date 或 datetime 属性时，可以指定日期的格式。 这种格式会被用在 模型序列化为数组或者 JSON ( https://learnku.com/docs/laravel/6.x/eloquent-serialization )。

/\*\*

\* 这个属性应该被转化为原生类型.

\*

\* @var array

\*/

protected $casts = [

'created\_at' => 'datetime:Y-m-d',

];

--

--

五、API 资源 (Eloquent：API 资源)

--

https://learnku.com/docs/laravel/6.x/eloquent-resources/5180

--

1、简介

当构建 API 时，你往往需要一个转换层来联结你的 Eloquent 模型和实际返回给用户的 JSON 响应。Laravel 的资源类能够让你以更直观简便的方式将模型和模型集合转化成 JSON。

--

2、生成资源

你可以使用 make:resource Artisan 命令来生成一个资源类。默认情况下生成的资源都会被放置在应用程序的 app/Http/Resources 文件夹下。资源继承自 Illuminate\Http\Resources\Json\JsonResource 类。

// API资源

php artisan make:resource User

(1)、资源集合

除了生成资源转换单个模型外，你还可以生成资源集合用来转换模型的集合。这允许你在响应中包含与给定资源相关的链接与其他元信息。

你需要在生成资源时添加 --collection 标志以生成一个资源集合。或者，你也可以直接在资源的名称中包含 Collection 向 Laravel 表示应该生成一个资源集合。资源集合继承自 Illuminate\Http\Resources\Json\ResourceCollection 类。

// API资源集合

php artisan make:resource Users --collection

php artisan make:resource UserCollection

--

3、概念综述

Tip：这是对资源和资源集合的高度概述。强烈建议你阅读本文档的其他部分，以深入了解如何更好地自定义和使用资源。

在深入了解如何定制化编写你的资源之前，让我们先来看看在 Laravel 中如何使用资源。一个资源类表示一个单一模型需要被转换成 JSON 格式。例如，现在我们有一个简单的 User 资源类。

<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class User extends JsonResource

{

/\*\*

\* 将资源转换成数组。

\*

\* @param \Illuminate\Http\Request $request

\* @return array

\*/

public function toArray($request)

{

return [

'id' => $this->id,

'name' => $this->name,

'email' => $this->email,

'created\_at' => $this->created\_at,

'updated\_at' => $this->updated\_at,

];

}

}

?>

每一个资源类都定义了一个 toArray 方法，在发送响应时它会返回应该被转化成 JSON 的属性数组。注意在这里我们可以直接使用 $this 变量来访问模型属性。这是因为资源类将自动代理属性和方法到底层模型以方便访问。你可以在路由或控制器中返回已定义的资源。

use App\User;

use App\Http\Resources\User as UserResource;

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

return new UserResource(User::find(1));

});

(1)、资源集合

你可以在路由或者控制器中使用 collection 方法来创建资源实例，以返回多个资源的集合或分页响应。

use App\User;

use App\Http\Resources\User as UserResource;

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

return UserResource::collection(User::all());

});

当然了，使用如上方法你将不能添加任何附加的元数据和集合一起返回。如果你需要自定义资源集合响应，你需要创建一个专用的资源来表示集合。

// 资源集合

php artisan make:resource UserCollection

你可以轻松的在已生成的资源集合类中定义任何你想在响应中返回的元数据。

<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection

{

/\*\*

\* 将资源集合转换成数组。

\*

\* @param \Illuminate\Http\Request $request

\* @return array

\*/

public function toArray($request)

{

return [

'data' => $this->collection,

'links' => [

'self' => 'link-value',

],

];

}

}

?>

你可以在路由或者控制器中返回已定义的资源集合。

use App\User;

use App\Http\Resources\UserCollection;

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

return new UserCollection(User::all());

});

//

///1/ 保护集合的键

当从路由返回资源集合时，Laravel 将重置集合的键，使它们以简单的数字顺序。但是，可以将 preserveKeys 属性添加到资源类中，指示是否应保留集合键。

<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class User extends JsonResource

{

/\*\*

\* 指示是否应保留资源的集合键。

\*

\* @var bool

\*/

public $preserveKeys = true;

}

?>

当 preserveKeys 属性被设置为 true，集合的键将会被保护。

use App\User;

use App\Http\Resources\User as UserResource;

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

return UserResource::collection(User::all()->keyBy->id);

});

//

///2/ 自定义基础资源类

通常，资源集合的 $this->collection 属性会自动填充，结果是将集合的每个项映射到其单个资源类。假定单一资源类是集合的类名，但结尾没有 Collection 字符串。

例如，UserCollection 将给定的用户实例映射到 User 资源中。若要自定义此行为，你可以重写资源集合的 $collects 属性。

<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection

{

/\*\*

\* collects 属性定义了资源类。

\*

\* @var string

\*/

public $collects = 'App\Http\Resources\Member';

}

?>

--

4、编写资源

Tip：如果你还没有阅读 概念综述( https://learnku.com/docs/laravel/6.x/eloquent-resources/5180#concept-overview )，那么在继续阅读本文档前，强烈建议你去阅读一下。

从本质上来说，资源的作用很简单。它们只需要将一个给定的模型转换成一个数组。所以每一个资源都包含一个 toArray 方法用来将你的模型属性转换成一个可以返回给用户的 API 友好数组。

<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class User extends JsonResource

{

/\*\*

\* 将资源转换成数组。

\*

\* @param \Illuminate\Http\Request $request

\* @return array

\*/

public function toArray($request)

{

return [

'id' => $this->id,

'name' => $this->name,

'email' => $this->email,

'created\_at' => $this->created\_at,

'updated\_at' => $this->updated\_at,

];

}

}

?>

你可以在路由或者控制器中返回已经定义的资源。

use App\User;

use App\Http\Resources\User as UserResource;

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

return new UserResource(User::find(1));

});

(1)、关联

如果你希望在响应中包含关联资源，你只需要将它们添加到 toArray 方法返回的数组中。在下面这个例子里，我们将使用 Post 资源的 collection 方法将用户的文章添加到资源响应中。

/\*\*

\* 将资源转换成数组。

\*

\* @param \Illuminate\Http\Request $request

\* @return array

\*/

public function toArray($request)

{

return [

'id' => $this->id,

'name' => $this->name,

'email' => $this->email,

'posts' => PostResource::collection($this->posts),

'created\_at' => $this->created\_at,

'updated\_at' => $this->updated\_at,

];

}

Tip： 如果你只想在关联已经加载时才添加关联资源，请查看文档 条件关联 ( https://learnku.com/docs/laravel/5.7/eloquent-resources/1407#conditional-relationships )。

(2)、资源集合

资源是将单个模型转换成数组，而资源集合是将多个模型的集合转换成数组。所有的资源都提供了一个 collection 方法来生成一个 「临时」资源集合，所以你没有必要为每一个模型类型都编写一个资源集合类。

use App\User;

use App\Http\Resources\User as UserResource;

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

return UserResource::collection(User::all());

});

不过，如果你需要自定义返回集合的元数据，则仍需要定义一个资源集合。

<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection

{

/\*\*

\* 将资源集合转换成数组

\*

\* @param \Illuminate\Http\Request $request

\* @return array

\*/

public function toArray($request)

{

return [

'data' => $this->collection,

'links' => [

'self' => 'link-value',

],

];

}

}

?>

和单个资源一样，你可以在路由或控制器中直接返回资源集合。

use App\User;

use App\Http\Resources\UserCollection;

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

return new UserCollection(User::all());

});

(3)、数据包裹

默认情况下，当资源响应被转换成 JSON 时，顶层资源将会被包裹在 data 键中。因此一个典型的资源集合响应如下所示。

{

"data": [

{

"id": 1,

"name": "Eladio Schroeder Sr.",

"email": "therese28@example.com",

},

{

"id": 2,

"name": "Liliana Mayert",

"email": "evandervort@example.com",

}

]

}

你可以使用资源基类的 withoutWrapping 方法来禁用顶层资源的包裹。通常，你应该在 AppServiceProvider 或其他在程序每一个请求中都会被加载的 服务提供者 ( https://learnku.com/docs/laravel/6.x/providers ) 中调用此方法。

<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

use Illuminate\Http\Resources\Json\Resource;

class AppServiceProvider extends ServiceProvider

{

/\*\*

\* 在注册后进行服务的启动

\*

\* @return void

\*/

public function boot()

{

Resource::withoutWrapping();

}

/\*\*

\* 在容器中注册绑定

\*

\* @return void

\*/

public function register()

{

//

}

}

?>

注意：withoutWrapping 方法只会禁用顶层资源的包裹，不会删除你手动添加到资源集合中的 data 键。

(4)、包裹嵌套资源

你可以完全自由地决定资源关联如何被包裹。如果你希望无论怎样嵌套，都将所有资源集合包裹在 data 键中，那么你需要为每个资源都定义一个资源集合类，并将返回的集合包裹在 data 键中。

当然，你可能会担心这样顶层资源将会被包裹在两个 data 键中。请放心， Laravel 将永远不会让你的资源被双层包裹，因此你不必担心被转换的资源集合会被多重嵌套。

<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class CommentsCollection extends ResourceCollection

{

/\*\*

\* 将资源集合转换成数组

\*

\* @param \Illuminate\Http\Request $request

\* @return array

\*/

public function toArray($request)

{

return ['data' => $this->collection];

}

}

?>

(5)、数据包裹和分页

当在资源响应中返回分页集合时，即使你调用了 withoutWrapping 方法， Laravel 也会将你的资源数据包裹在 data 键中。这是因为分页响应中总会有 meta 和 links 键包含着分页状态信息。

{

"data": [

{

"id": 1,

"name": "Eladio Schroeder Sr.",

"email": "therese28@example.com",

},

{

"id": 2,

"name": "Liliana Mayert",

"email": "evandervort@example.com",

}

],

"links":{

"first": "http://example.com/pagination?page=1",

"last": "http://example.com/pagination?page=1",

"prev": null,

"next": null

},

"meta":{

"current\_page": 1,

"from": 1,

"last\_page": 1,

"path": "http://example.com/pagination",

"per\_page": 15,

"to": 10,

"total": 10

}

}

(6)、分页

你可以将分页实例传递给资源的 collection 方法或者自定义的资源集合。

use App\User;

use App\Http\Resources\UserCollection;

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

return new UserCollection(User::paginate());

});

分页响应中总有 meta 和 links 键包含着分页状态信息。

{

"data": [

{

"id": 1,

"name": "Eladio Schroeder Sr.",

"email": "therese28@example.com",

},

{

"id": 2,

"name": "Liliana Mayert",

"email": "evandervort@example.com",

}

],

"links":{

"first": "http://example.com/pagination?page=1",

"last": "http://example.com/pagination?page=1",

"prev": null,

"next": null

},

"meta":{

"current\_page": 1,

"from": 1,

"last\_page": 1,

"path": "http://example.com/pagination",

"per\_page": 15,

"to": 10,

"total": 10

}

}

(7)、条件属性

有些时候，你可能希望在给定条件满足时添加属性到资源响应里。例如，你可能希望如果当前用户是 「管理员」 时添加某个值到资源响应中。在这种情况下 Laravel 提供了一些辅助方法来帮助你解决问题。 when 方法可以被用来有条件地向资源响应添加属性。

/\*\*

\* 将资源转换成数组

\*

\* @param \Illuminate\Http\Request $request

\* @return array

\*/

public function toArray($request)

{

return [

'id' => $this->id,

'name' => $this->name,

'email' => $this->email,

'secret' => $this->when(Auth::user()->isAdmin(), 'secret-value'),

'created\_at' => $this->created\_at,

'updated\_at' => $this->updated\_at,

];

}

在上面这个例子中，只有当 isAdmin 方法返回 true 时， secret 键才会最终在资源响应中被返回。如果该方法返回 false ，则 secret 键将会在资源响应被发送给客户端之前被删除。 when 方法可以使你避免使用条件语句拼接数组，转而用更优雅的方式来编写你的资源。

when 方法也接受闭包作为其第二个参数，只有在给定条件为 true 时，才从闭包中计算返回的值。

'secret' => $this->when(Auth::user()->isAdmin(), function () {

return 'secret-value';

}),

//

///1/ 有条件的合并数据

有些时候，你可能希望在给定条件满足时添加多个属性到资源响应里。在这种情况下，你可以使用 mergeWhen 方法在给定的条件为 true 时将多个属性添加到响应中。

/\*\*

\* 将资源转换成数组

\*

\* @param \Illuminate\Http\Request $request

\* @return array

\*/

public function toArray($request)

{

return [

'id' => $this->id,

'name' => $this->name,

'email' => $this->email,

$this->mergeWhen(Auth::user()->isAdmin(), [

'first-secret' => 'value',

'second-secret' => 'value',

]),

'created\_at' => $this->created\_at,

'updated\_at' => $this->updated\_at,

];

}

同理，如果给定的条件为 false 时，则这些属性将会在资源响应被发送给客户端之前被移除。

注意：mergeWhen 方法不应该被使用在混合字符串和数字键的数组中。此外，它也不应该被使用在不按顺序排列的数字键的数组中。

(8)、条件关联

除了有条件地添加属性之外，你还可以根据模型关联是否已加载来有条件地在你的资源响应中包含关联。这允许你在控制器中决定加载哪些模型关联，这样你的资源可以在模型关联被加载后才添加它们。

这样做可以避免在你的资源中出现 「N+1」 查询问题。你应该使用 whenLoaded 方法来有条件的加载关联。为了避免加载不必要的关联，此方法接受关联的名称而不是关联本身作为其参数。

/\*\*

\* 将资源转换成数组

\*

\* @param \Illuminate\Http\Request $request

\* @return array

\*/

public function toArray($request)

{

return [

'id' => $this->id,

'name' => $this->name,

'email' => $this->email,

'posts' => PostResource::collection($this->whenLoaded('posts')),

'created\_at' => $this->created\_at,

'updated\_at' => $this->updated\_at,

];

}

在上面这个例子中，如果关联没有被加载，则 posts 键将会在资源响应被发送给客户端之前被删除。

//

///1/ 条件中间表信息

除了在你的资源响应中有条件地包含关联外，你还可以使用 whenPivotLoaded 方法有条件地从多对多关联的中间表中添加数据。 whenPivotLoaded 方法接受的第一个参数为中间表的名称。第二个参数是一个闭包，它定义了在模型上如果中间表信息可用时要返回的值。

/\*\*

\* 将资源转换成数组

\*

\* @param \Illuminate\Http\Request $request

\* @return array

\*/

public function toArray($request)

{

return [

'id' => $this->id,

'name' => $this->name,

'expires\_at' => $this->whenPivotLoaded('role\_user', function () {

return $this->pivot->expires\_at;

}),

];

}

如果你的中间表使用的是 pivot 以外的访问器，你可以使用 whenPivotLoadedAs 方法。

/\*\*

\* 将资源转换成数组

\*

\* @param \Illuminate\Http\Request $request

\* @return array

\*/

public function toArray($request)

{

return [

'id' => $this->id,

'name' => $this->name,

'expires\_at' => $this->whenPivotLoadedAs('subscription', 'role\_user', function () {

return $this->subscription->expires\_at;

}),

];

}

(9)、添加元数据

一些 JSON API 标准需要你在资源和资源集合响应中添加元数据。这通常包括资源或相关资源的 links ，或一些关于资源本身的元数据。如果你需要返回有关资源的其他元数据，只需要将它们包含在 toArray 方法中即可。例如在转换资源集合时你可能需要添加 links 信息。

/\*\*

\* 将资源转换成数组

\*

\* @param \Illuminate\Http\Request $request

\* @return array

\*/

public function toArray($request)

{

return [

'data' => $this->collection,

'links' => [

'self' => 'link-value',

],

];

}

当添加额外的元数据到你的资源中时，你不必担心会覆盖 Laravel 在返回分页响应时自动添加的 links 或 meta 键。你添加的任何其他 links 会与分页响应添加的 links 相合并。

//

///1/ 顶层元数据

有时候你可能希望当资源被作为顶层资源返回时添加某些元数据到资源响应中。这通常包括整个响应的元信息。你可以在资源类中添加 with 方法来定义元数据。此方法应返回一个元数据数组，当资源被作为顶层资源渲染时，这个数组将会被包含在资源响应中。

<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection

{

/\*\*

\* 将资源集合转换成数组

\*

\* @param \Illuminate\Http\Request $request

\* @return array

\*/

public function toArray($request)

{

return parent::toArray($request);

}

/\*\*

\* 返回应该和资源一起返回的其他数据数组

\*

\* @param \Illuminate\Http\Request $request

\* @return array

\*/

public function with($request)

{

return [

'meta' => [

'key' => 'value',

],

];

}

}

?>

//

///2/ 构造资源时添加元数据

你还可以在路由或者控制器中构造资源实例时添加顶层数据。所有资源都可以使用 additional 方法来接受应该被添加到资源响应中的数据数组。

return (new UserCollection(User::all()->load('roles')))

->additional(['meta' => [

'key' => 'value',

]]);

--

5、资源响应 (响应资源)

就像你知道的那样， 资源可以直接在路由和控制器中被返回。

use App\User;

use App\Http\Resources\User as UserResource;

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

return new UserResource(User::find(1));

});

但有些时候，在发送给客户端前你可能需要自定义 HTTP 响应。 你有两种办法。第一，你可以在链式上调用 response 方法。此方法将会返回 Illuminate\Http\Response 实例，允许你自定义响应头信息。

use App\User;

use App\Http\Resources\User as UserResource;

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

return (new UserResource(User::find(1)))

->response()

->header('X-Value', 'True');

});

另外，你还可以在资源中定义一个 withResponse 方法。此方法将会在资源被作为顶层资源在响应时被调用。

<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class User extends JsonResource

{

/\*\*

\* 资源转换成数组

\*

\* @param \Illuminate\Http\Request $request

\* @return array

\*/

public function toArray($request)

{

return [

'id' => $this->id,

];

}

/\*\*

\* 自定义响应

\*

\* @param \Illuminate\Http\Request $request

\* @param \Illuminate\Http\Response $response

\* @return void

\*/

public function withResponse($request, $response)

{

$response->header('X-Value', 'True');

}

}

?>

--

--

六、序列化 (Eloquent：序列化)

--

https://learnku.com/docs/laravel/6.x/eloquent-serialization/5181

--

1、简介

构建 JSON API 时，经常需要把模型和关联转化为数组或 JSON。针对这些操作，Eloquent 提供了一些便捷方法，以及对序列化中的属性控制。

--

2、序列化模型 & 集合

(1)、序列化为数组

要转化模型及其加载的 关联 ( https://learnku.com/docs/laravel/6.x/eloquent-relationships ) 为数组，可以使用 toArray 方法。这是一个递归的方法，因此所有的属性和关联（包括关联的关联）都将转化成数组。

$user = App\User::with('roles')->first();

return $user->toArray();

可以转化模型的单个实例为数组。

$user = App\User::first();

return $user->attributesToArray();

也可以转化整个模型 集合 为数组。

$users = App\User::all();

return $users->toArray();

(2)、序列化为 JSON

方法 toJson 可以把模型转化成 JSON。和方法 toArray 一样， toJson 方法也是递归的，因此所有属性和关联都会转化成 JSON, 你还可以指定由 PHP 支持的 JSON 编码选项( http://php.net/manual/en/function.json-encode.php )。

$user = App\User::find(1);

return $user->toJson();

return $user->toJson(JSON\_PRETTY\_PRINT);

也可以把模型或集合转成字符串，方法 toJson 将自动调用。

$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转成 JSON， 因此可以在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();

});

//

///1/ 关联关系

当一个模型被转化为 JSON 的时候，它加载的关联关系也将自动转化为 JSON 对象被包含进来。同时，通过小驼峰定义的关联方法，关联的 JSON 属性将会是蛇形命名。

--

3、隐藏 JSON 属性

有时要将模型数组或 JSON 中的某些属性进行隐藏，比如密码，则可以在模型中添加 $hidden 属性。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

/\*\*

\* 数组中的属性会被隐藏。

\*

\* @var array

\*/

protected $hidden = ['password'];

}

?>

注意：隐藏关联时，需使用关联的方法名。

此外，也可以使用属性 $visible 定义一个模型数组和 JSON 可见的白名单。转化后的数组或 JSON 不会出现其他的属性。

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

/\*\*

\* 数组中的属性会被展示。

\*

\* @var array

\*/

protected $visible = ['first\_name', 'last\_name'];

}

?>

(1)、临时修改可见属性

//// 可见属性

如果你想要在一个模型实例中显示隐藏的属性，你可以使用 makeVisible 方法。makeVisible 方法返回模型实例。

return $user->makeVisible('attribute')->toArray();

//

//// 隐藏属性

相应地，如果你想要在一个模型实例中隐藏可见的属性，你可以使用 makeHidden 方法。

return $user->makeHidden('attribute')->toArray();

--

4、追加 JSON 值

有时，需要在数组或 JSON 中添加一些数据库中不存在字段的对应属性。要实现这个功能，首先要定义一个 修改器( https://learnku.com/docs/laravel/6.x/eloquent-mutators )。

<?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';

}

}

?>

然后，在模型属性 appends 中添加该属性名。注意，尽管访问器使用「驼峰命名法」方式定义，但是属性名通常以「蛇形命名法」的方式来引用。

<?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'];

}

?>

使用 append 方法追加属性后，它将包含在模型的数组和 JSON 中。appends 数组中的属性也将遵循模型上配置的 visible 和 hidden 设置。

(1)、运行时追加

你可以在单个模型实例上使用 append 方法来追加属性。或者，使用 setAppends 方法来重写整个追加属性的数组。

return $user->append('is\_admin')->toArray();

return $user->setAppends(['is\_admin'])->toArray();

--

5、序列化日期

(1)、自定义任意属性的日期格式

你可以在 Eloquent 的 属性类型转换 ( https://learnku.com/docs/laravel/6.x/eloquent-mutators#attribute-casting ) 中单独为日期属性自定义日期格式。

protected $casts = [

'birthday' => 'date:Y-m-d',

'joined\_at' => 'datetime:Y-m-d H:00',

];

(2)、Carbon 全局自定义

Laravel 扩展了 Carbon 日期库来为 Carbon ( https://github.com/briannesbitt/Carbon ) 的 JSON 序列化提供便利。可以使用 Carbon::serializeUsing 方法来自定义所有 Carbon 日期在整个应用中的序列化。serializeUsing 方法接受一个闭包，该闭包返回一个字符串形式的日期。

<?php

namespace App\Providers;

use Illuminate\Support\Carbon;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider

{

/\*\*

\* 在服务容器中注册绑定。

\*

\* @return void

\*/

public function register()

{

//

}

/\*\*

\* 执行注册后，启动服务。

\*

\* @return void

\*/

public function boot()

{

Carbon::serializeUsing(function ($carbon) {

return $carbon->format('U');

});

}

}

?>

--

--

第十卷 测试相关

--

--

一、快速入门 (测试：入门指南)

--

https://learnku.com/docs/laravel/6.x/testing/5182

--

1、介绍

Laravel 天生就具有测试的基因。事实上，Laravel 默认就支持用 PHPUnit 来做测试，并为你的应用程序配置好了 phpunit.xml 文件。框架还提供了一些便利的辅助函数，让你可以更直观的测试你的应用程序。

默认情况，你的应用 tests 目录中包含两个子目录：Feature 和 Unit。单元测试是针对你的代码中非常少，而且相对独立的一部分代码来进行的测试。实际上，大部分单元测试都是针对单个方法进行的。功能测试是针对大面积代码进行的测试，包括多个对象之间的交互，甚至是对 JSON 端点的完整 HTTP 请求。

Feature 和 Unit 目录中都提供一个 ExampleTest.php 测试示例文件。安装一个新的 Laravel 应用程序之后，在命令行下运行 phpunit 命令，即可运行测试。

--

2、环境

在使用 phpunit 进行测试时，Laravel 将根据 phpunit.xml 文件设定的环境变量自动将环境设置为 testing，并将 Session 及缓存以 array 的形式存储，也就是说在测试时不会持久化任何 Session 或缓存数据。

你可以随意创建其它必要的测试环境配置。testing 环境变量可以在 phpunit.xml 文件中修改，但是在运行测试之前，请确保使用 config:clear Artisan 命令来清除配置信息的缓存！

此外，你还可以在你的项目根目录下创建一个 .env.testing 文件，在运行单元测试，或者使用带有 --env=testing 选项的 Artisan 命令时，.env 文件中的变量会被这个文件覆盖。

--

3、创建并运行测试

可以使用 Artisan 命令 make:test 创建一个新的测试用例。

//

//// 测试用例创建

// 在 Feature 目录下创建一个测试类...

php artisan make:test UserTest

//

// 在 Unit 目录下创建一个测试类...

php artisan make:test UserTest --unit

//

测试类生成之后，你就可以像平常使用 PHPUnit 一样来定义测试方法。要运行测试只需要在终端上运行 phpunit 命令即可。

<?php

namespace Tests\Unit;

use Tests\TestCase;

use Illuminate\Foundation\Testing\RefreshDatabase;

class ExampleTest extends TestCase

{

/\*\*

\* 一个基本的测试示例

\*

\* @return void

\*/

public function testBasicTest()

{

$this->assertTrue(true);

}

}

?>

注意：如果要在你的测试类中定义自己的 setUp / tearDown 方法，请确保调用了父类中的 parent::setUp() / parent::tearDown() 方法。

--

--

二、HTTP 测试

--

https://learnku.com/docs/laravel/6.x/http-tests/5183

--

1、简介

Laravel 为 HTTP 请求的生成和输出的检查都提供了非常流畅的 API。例如，你可以查看下面的这个测试用例。

<?php

namespace Tests\Feature;

use Tests\TestCase;

use Illuminate\Foundation\Testing\RefreshDatabase;

use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase

{

/\*\*

\* A basic test example.

\*

\* @return void

\*/

public function testBasicTest()

{

$response = $this->get('/');

$response->assertStatus(200);

}

}

?>

get 方法会创建一个 GET 请求来请求你的应用，而 assertStatus 方法断言返回的响应是指定的 HTTP 状态码。 除了这个简单的断言之外， Laravel 也包含检查响应头、响应体、JSON 结构等断言。

(1)、自定义请求头

在请求发送到应用程式之前你可以使用 withHeaders 方法定义请求头。这允许你在请求上添加想要的请求头。

<?php

class ExampleTest extends TestCase

{

/\*\*

\* A basic functional test example.

\*

\* @return void

\*/

public function testBasicExample()

{

$response = $this->withHeaders([

'X-Header' => 'Value',

])->json('POST', '/user', ['name' => 'Sally']);

$response

->assertStatus(201)

->assertJson([

'created' => true,

]);

}

}

?>

Tip：运行测试时，CSRF 中间件会自动禁用。

(2)、响应调试 (调试响应)

测试请求通过应用程序后，dump 和 dumpHeaders 方法将会在控制台打印响应体和响应头，可以用来分析和调试响应。

<?php

namespace Tests\Feature;

use Tests\TestCase;

use Illuminate\Foundation\Testing\RefreshDatabase;

use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase

{

/\*\*

\* A basic test example.

\*

\* @return void

\*/

public function testBasicTest()

{

$response = $this->get('/');

$response->dumpHeaders();

$response->dump();

}

}

?>

--

2、Session / 认证

Laravel 提供了几个可在 HTTP 测试时使用 Session 的辅助函数。首先，你需要传递一个数组给 withSession 方法来设置 Session 数据。这让你在应用程序的测试请求发送之前，先给数据加载 Session 变得简单。

<?php

class ExampleTest extends TestCase

{

public function testApplication()

{

$response = $this->withSession(['foo' => 'bar'])

->get('/');

}

}

?>

当然，一般使用 Session 时都是用于维护用户状态，如认证用户。actingAs 辅助函数提供了简单的方法来指定的用户认证为当前用户。 例如， 我们可以使用 工厂模型( https://learnku.com/docs/laravel/6.x/database-testing#writing-factories ) 来生成并认证用户。

<?php

use App\User;

class ExampleTest extends TestCase

{

public function testApplication()

{

$user = factory(User::class)->create();

$response = $this->actingAs($user)

->withSession(['foo' => 'bar'])

->get('/');

}

}

?>

你也可以通过传递看守器名称作为 actingAs 方法的第二参数以指定用户通过哪种看守器来认证。

$this->actingAs($user, 'api');

--

3、测试 JSON APIs

Laravel 也提供了几个辅助函数来测试 JSON APIs 和他们的响应。 例如，json, get， post， put， patch，和 delete 可以被用于发送各种 HTTP 动作。 你也可以轻松地将数据和请求头传递到这些方法中。让我们写一个 POST 请求到 /user 并断言返回期望的数据来开始使用他们。

<?php

class ExampleTest extends TestCase

{

/\*\*

\* A basic functional test example.

\*

\* @return void

\*/

public function testBasicExample()

{

$response = $this->json('POST', '/user', ['name' => 'Sally']);

$response

->assertStatus(201)

->assertJson([

'created' => true,

]);

}

}

?>

Tip：assertJson 方法将响应转换为数组并利用 PHPUnit::assertArraySubset 来验证给定的数组存在于应用返回的 JSON 响应中。 所以，如果 JSON 响应中有其他属性，测试仍旧会在给定数组存在的情况下通过。

(1)、验证 JSON 完全匹配

如果你想验证给定的数组 完全 匹配应用返回的 JSON 结果，你应该使用 assertExactJson 方法。

<?php

class ExampleTest extends TestCase

{

/\*\*

\* A basic functional test example.

\*

\* @return void

\*/

public function testBasicExample()

{

$response = $this->json('POST', '/user', ['name' => 'Sally']);

$response

->assertStatus(201)

->assertExactJson([

'created' => true,

]);

}

}

?>

--

4、测试文件上传

Illuminate\Http\UploadedFile 提供了一个 fake 方法用于生成虚拟的文件或者图像以供测试之用。它可以和 Storage facade 的 fake 方法相结合， 大幅度简化了文件上传测试。举个例子，你可以结合这两者的功能非常方便地进行头像上传表单测试。

<?php

namespace Tests\Feature;

use Tests\TestCase;

use Illuminate\Http\UploadedFile;

use Illuminate\Support\Facades\Storage;

use Illuminate\Foundation\Testing\RefreshDatabase;

use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase

{

public function testAvatarUpload()

{

Storage::fake('avatars');

$file = UploadedFile::fake()->image('avatar.jpg');

$response = $this->json('POST', '/avatar', [

'avatar' => $file,

]);

// Assert the file was stored...

Storage::disk('avatars')->assertExists($file->hashName());

// Assert a file does not exist...

Storage::disk('avatars')->assertMissing('missing.jpg');

}

}

?>

(1)、虚拟文件制定

在使用 fake 方法创建文件时，你可以指定图像的宽高以及大小，从而更好的验证测试规则。

UploadedFile::fake()->image('avatar.jpg', $width, $height)->size(100);

除创建图像外，你也可以用 create 方法创建其他类型的文件。

UploadedFile::fake()->create('document.pdf', $sizeInKilobytes);

--

5、可用的断言方法

(1)、响应断言

Laravel 给 PHPUnit ( https://phpunit.de/ ) 测试提供了各种各样的常用断言方法。可从 json, get, post, put，和 delete 测试方法中访问这些断言。

//

//// assertCookie

断言响应中包含给定的 cookie。

$response->assertCookie($cookieName, $value = null);

//

//// assertCookieExpired

断言响应中包含了给定的 cookie 且它已过期。

$response->assertCookieExpired($cookieName);

//

//// assertCookieNotExpired

断言响应中包含了给定的 cookie 且它未过期。

$response->assertCookieNotExpired($cookieName);

//

//// assertCookieMissing

断言响应中不包含给定的 cookie。

$response->assertCookieMissing($cookieName);

//

//// assertDontSee

断言响应中不包含给定的字符串。

$response->assertDontSee($value);

//

//// assertDontSeeText

断言给定字符串不包含在响应文本中。

$response->assertDontSeeText($value);

//

//// assertExactJson

断言响应中包含的数据与给定的 JSON 数据完全匹配。

$response->assertExactJson(array $data);

//

//// assertForbidden

断言响应中有禁止状态码。

$response->assertForbidden();

//

//// assertHeader

断言响应中有给定的包头。

$response->assertHeader($headerName, $value = null);

//

//// assertHeaderMissing

断言响应中没有给定的报头。

$response->assertHeaderMissing($headerName);

//

//// assertJson

断言响应包含给定的 JSON 数据。

$response->assertJson(array $data);

//

//// assertJsonCount

断言响应 JSON 中有一个数组，其中包含给定键的预期元素数量。

$response->assertJsonCount($count, $key = null);

//

//// assertJsonFragment

断言响应包含给定 JSON 片段。

$response->assertJsonFragment(array $data);

//

//// assertJsonMissing

断言响应未包含给定的 JSON 片段。

$response->assertJsonMissing(array $data);

//

//// assertJsonMissingExact

断言响应不包含确切的 JSON 片段。

$response->assertJsonMissingExact(array $data);

//

//// assertJsonMissingValidationErrors

断言响应没有给定键的 JSON 验证错误。

$response->assertJsonMissingValidationErrors($keys);

//

//// assertJsonStructure

断言响应具有给定的 JSON 结构。

$response->assertJsonStructure(array $structure);

//

//// assertJsonValidationErrors

断言响应具有给定键的给定 JSON 验证错误。

$response->assertJsonValidationErrors(array $data);

//

//// assertLocation

断言响应在 Location 头部中具有给定的 URI 值。

$response->assertLocation($uri);

//

//// assertNotFound

断言响应具有未找到状态码。

$response->assertNotFound();

//

//// assertOk

断言响应有 200 状态码。

$response->assertOk();

//

//// assertPlainCookie

断言响应包含给定的 cookie （未加密）。

$response->assertPlainCookie($cookieName, $value = null);

//

//// assertRedirect

断言响应会重定向到给定的 URI。

$response->assertRedirect($uri);

//

//// assertSee

断言给定的字符串包含在响应中：断言响应中有序包含了给定的字符串。

$response->assertSee($value);

//

//// assertSeeInOrder

断言响应中有序包含了给定的字符串。

$response->assertSeeInOrder(array $values);

//

//// assertSeeText

断言给定的字符串包含在响应文本中。

$response->assertSeeText($value);

//

//// assertSeeTextInOrder

断言给定的字符串有序包含在响应文本中。

$response->assertSeeTextInOrder(array $values);

//

//// assertSessionHas

断言 session 中包含给定的数据。

$response->assertSessionHas($key, $value = null);

//

//// assertSessionHasInput

断言 session 中有给定值数组。

$response->assertSessionHasInput($key, $value = null);

//

//// assertSessionHasAll

断言 session 中有给定值列表。

$response->assertSessionHasAll(array $data);

//

//// assertSessionHasErrors

断言 session 中包含一个给定字段的错误。

$response->assertSessionHasErrors(array $keys, $format = null, $errorBag = 'default');

//

//// assertSessionHasErrorsIn

断言 session 中具有给定的错误。

$response->assertSessionHasErrorsIn($errorBag, $keys = [], $format = null);

//

//// assertSessionHasNoErrors

断言 session 没有错误。

$response->assertSessionHasNoErrors();

//

//// assertSessionDoesntHaveErrors

断言 session 没有给定键错误。

$response->assertSessionDoesntHaveErrors($keys = [], $format = null, $errorBag = 'default');

//

//// assertSessionMissing

断言 session 中不包含给定键。

$response->assertSessionMissing($key);

//

//// assertStatus

断言响应中具有给定的状态码。

$response->assertStatus($code);

//

//// assertSuccessful

断言响应中有成功的状态码。

$response->assertSuccessful();

//

//// assertUnauthorized

断言响应中有未授权的（401）状态码。

$response->assertUnauthorized();

//

//// assertViewHas

断言响应视图是一段给定的数据。

$response->assertViewHas($key, $value = null);

//

//// assertViewHasAll

断言响应视图具有给定的数据列表。

$response->assertViewHasAll(array $data);

//

//// assertViewIs

断言路由返回给定的视图。

$response->assertViewIs($value);

//

//// assertViewMissing

断言响应视图缺少一段绑定数据。

$response->assertViewMissing($key);

(2)、认证断言 (Authentication Assertions)

Laravel 还为你的 PHPUnit ( https://phpunit.de/ ) 测试提供了各种与身份验证相关的断言。

(Method/Description)

$this->assertAuthenticated($guard = null); 断言此用户已被认证。

$this->assertGuest($guard = null); 断言此用户未被认证。

$this->assertAuthenticatedAs($user, $guard = null); 断言给定的用户被认证。

$this->assertCredentials(array $credentials, $guard = null); 断言给定的凭证有效。

$this->assertInvalidCredentials(array $credentials, $guard = null); 断言给定的凭证无效。

--

--

三、命令行测试 (控制台测试)

--

https://learnku.com/docs/laravel/6.x/console-tests/5184

--

1、简介

除了简化 HTTP 测试之外， Laravel 为测试用户输入的控制台应用提供了简单的 API。

--

2、期望输入 / 输出

Laravel 使用 expectsQuestion 方法为控制台命令「模拟」用户输入。此外，还可以使用 assertExitCode 和 expectsOutput 方法指定控制台命令退出码和期望输出文本。举例如下。

Artisan::command('question', function () {

$name = $this->ask('What is your name?');

$language = $this->choice('Which language do you program in?', [

'PHP',

'Ruby',

'Python',

]);

$this->line('Your name is '.$name.' and you program in '.$language.'.');

});

您可以参考下面的示例代码来测试命令，其中使用了 expectsQuestion ，expectsOutput ，和 assertExitCode 方法。

/\*\*

\* 测试控制台命令

\*

\* @return void

\*/

public function test\_console\_command()

{

$this->artisan('question')

->expectsQuestion('What is your name?', 'Taylor Otwell')

->expectsQuestion('Which language do you program in?', 'PHP')

->expectsOutput('Your name is Taylor Otwell and you program in PHP.')

->assertExitCode(0);

}

--

--

四、数据库测试

--

https://learnku.com/docs/laravel/6.x/database-testing/5185

--

1、简介

Laravel 提供了各种有用的工具，可以更轻松地测试数据库驱动的应用程序。 首先，你可以使用 assertDatabaseHas 辅助函数，来断言数据库中是否存在与指定条件互相匹配的数据。 例如，如果我们想要验证 users 表中是否存在 email 值为 sally@example.com 的数据，你可以执行以下操作。

public function testDatabase()

{

// 调用应该程序...

//

$this->assertDatabaseHas('users', [

'email' => 'sally@example.com',

]);

}

你也可以使用 assertDatabaseMissing 辅助函数断言数据库中不存在给定的数据。

当然，assertDatabaseHas 方法和其他类似的帮助函数只是为了方便起见。你还可以自由使用任何 PHPUnit 的内置断言方法来补充你的测试。

--

2、生成模型工厂

使用 make:factory Artisan command ( https://learnku.com/docs/laravel/6.x/artisan ) 命令可以创建一个模型工厂：

// 模型工厂生成

php artisan make:factory PostFactory

//

//// 模型工厂指定模型

新生成的工厂位置在 database/factories 目录下。--model 选项可用于指示工厂创建的模型的名称。 此选项将使用给定模型预填充生成的工厂文件。

php artisan make:factory PostFactory --model=Post

--

3、每次测试后重置数据库

在每次测试后重置数据库是很有用的，这样前一次测试的数据不会干扰后续测试。 RefreshDatabase trait 会采用最优化的方法来迁移测试数据库，这取决于你使用的是内存数据库还是传统数据库。 在测试类中引用这个 trait ，一切都将为你处理。

<?php

namespace Tests\Feature;

use Tests\TestCase;

use Illuminate\Foundation\Testing\RefreshDatabase;

use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase

{

use RefreshDatabase;

/\*\*

\* 一个基本的功能测试示例。

\*

\* @return void

\*/

public function testBasicExample()

{

$response = $this->get('/');

// ...

}

}

?>

--

4、创建模型工厂

进行测试时，运行测试之前常常需要插入一些数据到数据库中。当你创建测试数据时，除了手动设置每个字段的值，Laravel 还可以使用 Eloquent 模型 ( https://learnku.com/docs/laravel/6.0/eloquent ) 的工厂来设置每个属性的默认值。在开始之前， 你可以先看一下应用程序的 database/factories/UserFactory.php 的文件。 开箱即用，这个文件包含一个模型工厂定义。

use Illuminate\Support\Str;

use Faker\Generator as Faker;

$factory->define(App\User::class, function (Faker $faker) {

return [

'name' => $faker->name,

'email' => $faker->unique()->safeEmail,

'email\_verified\_at' => now(),

'password' => '$2y$10$TKh8H1.PfQx37YgCzwiKb.KjNyWgaHb9cbcoQgdIVFlYg7B77UdFm', // secret

'remember\_token' => Str::random(10),

];

});

在这个模型工厂定义的闭包中，你可以返回模型上所有属性的默认测试值。闭包将接受 PHP 函数库 Faker ( https://github.com/fzaninotto/Faker ) 的一个实例，它允许你便捷的生成各种随机数据来进行测试。

你也可以为每个模型创建一个工厂文件以便实现更好的组织。 例如，你可以自 database/factories 目录下创建 UserFactory.php 和 CommentFactory.php 文件。 Laravel 将自动加载 factories 目录下的所有文件。

Tip：你也可以在 config/app.php 配置文件中添加 faker\_locale 选项来设置 Faker 的语言环境。

(1)、工厂状态

工厂状态可以让你任意组合你的模型工厂，仅需要做出适当差异化的修改，就可以达到让模型拥有多种不同的状态。例如， 你的 User 模型中可以修改某个默认属性值来达到标识一种 delinquent 状态。你可以使用 state 方法来进行这种状态转换。对于简单的工厂状态，你可以传入要修改的属性数组。

$factory->state(App\User::class, 'delinquent', [

'account\_status' => 'delinquent',

]);

如果你的工厂状态需要计算或者使用 $faker 实例。你可以使用闭包方法来实现状态属性的修改。

$factory->state(App\User::class, 'address', function ($faker) {

return [

'address' => $faker->address,

];

});

(2)、工厂回调

工厂回调是使用 afterMaking 和 afterCreating 方法注册的，并且允许你在创建模型之后执行其他任务。例如，可以使用回调将附加模型与创建的模型相关联。

$factory->afterMaking(App\User::class, function ($user, $faker) {

// ...

});

$factory->afterCreating(App\User::class, function ($user, $faker) {

$user->accounts()->save(factory(App\Account::class)->make());

});

你还可以为 工厂状态 ( https://learnku.com/docs/laravel/6.x/database-testing/5185#factory-states ) 定义回调。

$factory->afterMakingState(App\User::class, 'delinquent', function ($user, $faker) {

// ...

});

$factory->afterCreatingState(App\User::class, 'delinquent', function ($user, $faker) {

// ...

});

--

5、在测试中使用模型工厂 (使用模型工厂)

(1)、创建模型

模型工厂定义后，就可以在测试或者是数据库的填充文件中，通过全局的 factory 函数来生成模型实例。因此，先让我们来看看几个模型创建的例子。 首先，我们将使用 make 方法创建模型但不将他们保存至数据库。

public function testDatabase()

{

$user = factory(App\User::class)->make();

// 在测试中使用模型...

}

你也可以创建一个含有多个模型的集合，或创建一个指定类型的模型。

// 创建三个 App\User 实例...

$users = factory(App\User::class, 3)->make();

//

///1/ 应用状态

你也可将任何状态( https://learnku.com/docs/laravel/6.x/database-testing/5185#factory-states )应用于模型。若将多个状态转换应用于模型，你应当为每个状态指定名称。

$users = factory(App\User::class, 5)->states('delinquent')->make();

$users = factory(App\User::class, 5)->states('premium', 'delinquent')->make();

//

///2/ 覆盖属性

如果要覆盖模型的某些默认值，你可以将一组值传递给 make 方法。 这样的话只有指定的值才会被替换，而其余值仍为工厂指定的默认值。

$user = factory(App\User::class)->make([

'name' => 'Abigail',

]);

Tip：使用工厂创建模型时，将自动禁用 批量赋值( https://learnku.com/docs/laravel/6.0/eloquent#mass-assignment )。

(2)、持久化模型

create 方法创建模型实例的同时还调用了把记录写入数据库的 save 方法。

public function testDatabase()

{

// 创建单个 App\User 实例...

$user = factory(App\User::class)->create();

// 创建 3 个 App\User 实例..

$users = factory(App\User::class, 3)->create();

// 在测试中使用模型...

}

传入一个数组给 create 方法重写模型的属性。

$user = factory(App\User::class)->create([

'name' => 'Abigail',

]);

(3)、模型关联 (关联模型)

在这个例子中，我们将为模型创建关联。使用 create 方法创建多模型时，返回一个 Eloquent 实例集合 ( https://learnku.com/docs/laravel/6.0/eloquent-collections )，这样就可以在集合上使用 each 等便捷方法。

$users = factory(App\User::class, 3)

->create()

->each(function ($user) {

$user->posts()->save(factory(App\Post::class)->make());

});

你可以使用 createMany 方法创建多个相关模型。

$user->posts()->createMany(

factory(App\Post::class, 3)->make()->toArray()

);

//

///1/ 关联 & 属性闭包

模型工厂定义时，也可以使用闭包里的属性来给模型添加关联。 例如，创建 Post 实例时，同时创建 User 实例，可以这样做。

$factory->define(App\Post::class, function ($faker) {

return [

'title' => $faker->title,

'content' => $faker->paragraph,

'user\_id' => function () {

return factory(App\User::class)->create()->id;

},

];

});

这些闭包接收一个包含工厂属性的数组。

$factory->define(App\Post::class, function ($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;

},

];

});

--

6、在测试中使用数据填充

如果你希望在测试期间使用 数据填充 ( https://learnku.com/docs/laravel/6.0/seeding ) 填充数据库，则可以使用 seed 方法。默认情况下， seed 方法将返回 DatabaseSeeder, 它会结束其他的数据填充器。 或者，将指定的数据填充器的类名传递给 seed 方法。

<?php

namespace Tests\Feature;

use Tests\TestCase;

use OrderStatusesTableSeeder;

use Illuminate\Foundation\Testing\RefreshDatabase;

use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase

{

use RefreshDatabase;

/\*\*

\* 测试创建新订单。

\*

\* @return void

\*/

public function testCreatingANewOrder()

{

// 运行整个数据填充器...

$this->seed();

// 运行单个数据填充器...

$this->seed(OrderStatusesTableSeeder::class);

// ...

}

}

?>

--

7、可用的断言方法

Laravel 为 PHPUnit ( https://phpunit.de/ ) 测试提供了多个数据库断言方法。

(Method/Description)

$this->assertDatabaseHas($table, array $data); 断言数据库表中包含给定的数据。

$this->assertDatabaseMissing($table, array $data); 断言数据库中的表不包含给定数据。

$this->assertSoftDeleted($table, array $data); 断言给定记录已被软删除。

--

--

五、测试模拟器 Mocking

--

https://learnku.com/docs/laravel/6.x/mocking/5186

--

1、简介

在 Laravel 应用程序测试中，你可能希望「模拟」应用程序的某些功能的行为，从而避免该部分在测试中真正执行。例如：在控制器执行过程中会触发事件，从而避免该事件在测试控制器时真正执行。这允许你在仅测试控制器 HTTP 响应的情况时，而不必担心触发事件。当然，你也可以在单独的测试中测试该事件逻辑。

Laravel 针对事件、任务和 Facades 的模拟，提供了开箱即用的辅助函数。这些函数基于 Mocker 封装而成，使用非常方便，无需手动调用复杂的 Mockery 函数。当然你也可以使用 Mockery ( http://docs.mockery.io/en/latest/ ) 或者使用 PHPUnit 创建自己的模拟器。

--

2、模拟对象

当模拟一个对象将通过 Laravel 的服务容器注入到应用中时，你将需要将模拟实例作为 instance 绑定到容器中。这将告诉容器使用对象的模拟实例，而不是构造对象的真身。

use Mockery;

use App\Service;

$this->instance(Service::class, Mockery::mock(Service::class, function ($mock) {

$mock->shouldReceive('process')->once();

}));

为了让以上过程更加便捷，你可以使用 Laravel 的基本测试用例类提供 mock 方法。

use App\Service;

$this->mock(Service::class, function ($mock) {

$mock->shouldReceive('process')->once();

});

同样，如果你想侦查一个对象，Laravel 的基本测试用例类提供了一个便捷的 spy 方法作为 Mockery::spy 的替代方法。

use App\Service;

$this->spy(Service::class, function ($mock) {

$mock->shouldHaveReceived('process');

});

--

3、任务模拟

作为模拟的替代方式，你可以使用 Bus Facade 的 fake 方法来防止任务被真正分发执行。使用 fake 的时候，断言一般出现在测试代码的后面。

<?php

namespace Tests\Feature;

use Tests\TestCase;

use App\Jobs\ShipOrder;

use Illuminate\Support\Facades\Bus;

use Illuminate\Foundation\Testing\RefreshDatabase;

use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase

{

public function testOrderShipping()

{

Bus::fake();

// 执行订单发货...

Bus::assertDispatched(ShipOrder::class, function ($job) use ($order) {

return $job->order->id === $order->id;

});

// 断言任务并未分发...

Bus::assertNotDispatched(AnotherJob::class);

}

}

?>

--

4、事件模拟

作为 mock 的替代方法，你可以使用 Event Facade 的 fake 方法来模拟事件监听，测试的时候并不会真正触发事件监听器。然后你就可以测试断言事件运行了，甚至可以检查他们接收的数据。使用 fake 的时候，断言一般出现在测试代码的后面。

<?php

namespace Tests\Feature;

use Tests\TestCase;

use App\Events\OrderShipped;

use App\Events\OrderFailedToShip;

use Illuminate\Support\Facades\Event;

use Illuminate\Foundation\Testing\RefreshDatabase;

use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase

{

/\*\*

\* 测试订单发送

\*/

public function testOrderShipping()

{

Event::fake();

// 执行订单发送...

Event::assertDispatched(OrderShipped::class, function ($e) use ($order) {

return $e->order->id === $order->id;

});

// 断言一个事件被发送了两次...

Event::assertDispatched(OrderShipped::class, 2);

// 未分配断言事件...

Event::assertNotDispatched(OrderFailedToShip::class);

}

}

?>

注意：调用 Event::fake() 后不会执行事件监听。所以，你基于事件的测试必须使用工厂模型，例如，在模型的 creating 事件中创建 UUID ，你应该调用 Event::fake() 之后 使用工厂模型。

//

///1/ 模拟事件的子集

如果你只想为特定的一组事件模拟事件监听器，你可以将它们传递给 fake 或 fakeFor 方法。

/\*\*

\* 测试订单流程。

\*/

public function testOrderProcess()

{

Event::fake([

OrderCreated::class,

]);

$order = factory(Order::class)->create();

Event::assertDispatched(OrderCreated::class);

// 其他事件照常发送...

$order->update([...]);

}

(1)、Scoped 事件模拟

如果你只想为部分测试模拟事件监听，则可以使用 fakeFor 方法。

<?php

namespace Tests\Feature;

use App\Order;

use Tests\TestCase;

use App\Events\OrderCreated;

use Illuminate\Support\Facades\Event;

use Illuminate\Foundation\Testing\RefreshDatabase;

use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase

{

/\*\*

\* 测试订单流程

\*/

public function testOrderProcess()

{

$order = Event::fakeFor(function () {

$order = factory(Order::class)->create();

Event::assertDispatched(OrderCreated::class);

return $order;

});

// 事件按正常方式发送，观察者将运行...

$order->update([...]);

}

}

?>

--

5、邮件模拟

你可以是用 Mail Facade 的 fake 方法来模拟邮件发送，测试时不会真的发送邮件，然后你可以断言 mailables ( https://learnku.com/docs/laravel/6.x/mail ) 发送给了用户，甚至可以检查他们收到的内容。使用 fakes 时，断言一般放在测试代码的后面。

<?php

namespace Tests\Feature;

use Tests\TestCase;

use App\Mail\OrderShipped;

use Illuminate\Support\Facades\Mail;

use Illuminate\Foundation\Testing\RefreshDatabase;

use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase

{

public function testOrderShipping()

{

Mail::fake();

// 断言没有发送任何邮件...

Mail::assertNothingSent();

// 执行订单发送...

Mail::assertSent(OrderShipped::class, function ($mail) use ($order) {

return $mail->order->id === $order->id;

});

// 断言一条发送给用户的消息...

Mail::assertSent(OrderShipped::class, function ($mail) use ($user) {

return $mail->hasTo($user->email) &&

$mail->hasCc('...') &&

$mail->hasBcc('...');

});

// 断言邮件被发送两次...

Mail::assertSent(OrderShipped::class, 2);

// 断言没有发送邮件...

Mail::assertNotSent(AnotherMailable::class);

}

}

?>

如果你用后台任务执行邮件发送队列，你应该是用 assertQueued 代替 assertSent。

Mail::assertQueued(...);

Mail::assertNotQueued(...);

--

6、通知模拟

你可以使用 Notification Facade 的 fake 方法来模拟通知的发送，测试时并不会真的发出通知。然后你可以断言 notifications ( https://learnku.com/docs/laravel/6.x/notifications ) 发送给了用户，甚至可以检查他们收到的内容。使用 fakes 时，断言一般放在测试代码后面。

<?php

namespace Tests\Feature;

use Tests\TestCase;

use App\Notifications\OrderShipped;

use Illuminate\Support\Facades\Notification;

use Illuminate\Notifications\AnonymousNotifiable;

use Illuminate\Foundation\Testing\RefreshDatabase;

use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase

{

public function testOrderShipping()

{

Notification::fake();

// 断言没有发送通知...

Notification::assertNothingSent();

// 执行订单发送...

Notification::assertSentTo(

$user,

OrderShipped::class,

function ($notification, $channels) use ($order) {

return $notification->order->id === $order->id;

}

);

// 断言向给定用户发送了通知...

Notification::assertSentTo(

[$user], OrderShipped::class

);

// 断言没有发送通知...

Notification::assertNotSentTo(

[$user], AnotherNotification::class

);

// 断言通过 Notification::route() 方法发送通知...

Notification::assertSentTo(

new AnonymousNotifiable, OrderShipped::class

);

}

}

?>

--

7、队列模拟

为模拟替代方案，你可以使用 Queue Facade 的 fake 方法避免把任务真的放到队列中执行。然后你就可以断言任务已经被推送入队列了，甚至可以检查它们收到的数据。使用 fakes 时，断言一般放在测试代码的后面。

<?php

namespace Tests\Feature;

use Tests\TestCase;

use App\Jobs\ShipOrder;

use Illuminate\Support\Facades\Queue;

use Illuminate\Foundation\Testing\RefreshDatabase;

use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase

{

public function testOrderShipping()

{

Queue::fake();

// 断言没有任务被发送...

Queue::assertNothingPushed();

// 执行订单发送...

Queue::assertPushed(ShipOrder::class, function ($job) use ($order) {

return $job->order->id === $order->id;

});

// 断言任务进入了指定队列...

Queue::assertPushedOn('queue-name', ShipOrder::class);

// 断言任务进入2次...

Queue::assertPushed(ShipOrder::class, 2);

// 断言没有一个任务进入队列...

Queue::assertNotPushed(AnotherJob::class);

// 断言任务是由特定的通道发送的..

Queue::assertPushedWithChain(ShipOrder::class, [

AnotherJob::class,

FinalJob::class

]);

}

}

?>

--

8、Storage 模拟 (存储模拟)

你可以使用 Storage Facade 的 fake 方法，轻松的生成一个模拟磁盘，结合 UploadedFile 类的文件生成工具，极大的简化了文件上传测试。例如。

<?php

namespace Tests\Feature;

use Tests\TestCase;

use Illuminate\Http\UploadedFile;

use Illuminate\Support\Facades\Storage;

use Illuminate\Foundation\Testing\RefreshDatabase;

use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase

{

public function testAlbumUpload()

{

Storage::fake('photos');

$response = $this->json('POST', '/photos', [

UploadedFile::fake()->image('photo1.jpg'),

UploadedFile::fake()->image('photo2.jpg')

]);

// 断言文件已存储...

Storage::disk('photos')->assertExists('photo1.jpg');

Storage::disk('photos')->assertExists(['photo1.jpg', 'photo2.jpg']);

// 断言文件不存在...

Storage::disk('photos')->assertMissing('missing.jpg');

Storage::disk('photos')->assertMissing(['missing.jpg', 'non-existing.jpg']);

}

}

?>

Tip：默认情况下，fake 方法将删除临时目录下所有文件。如果你想保留这些文件，你可以使用「persistentFake」。

--

9、Facades

与传统静态方法调用不同的是，facades 也可以被模拟。相较传统的静态方法而言，它具有很大的优势，即便你使用依赖注入，可测试性不逊半分。在测试中，你可能想在控制器中模拟对 Laravel Facade 的调用。比如下面控制器中的行为。

<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

class UserController extends Controller

{

/\*\*

\* 显示应用里所有用户

\*

\* @return Response

\*/

public function index()

{

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

//

}

}

?>

我们可以通过 shouldReceive 方法来模拟 Cache Facade，此函数会返回一个 Mockery ( https://github.com/padraic/mockery ) 实例。由于 Facade 的调用实际是由 Laravel 的 服务容器 ( https://learnku.com/docs/laravel/6.x/container ) 管理的，所以 Facade 能比传统的静态类表现出更好的可测试性。下面，让我们模拟一下 Cache Facade 的 get 方法。

<?php

namespace Tests\Feature;

use Tests\TestCase;

use Illuminate\Support\Facades\Cache;

use Illuminate\Foundation\Testing\RefreshDatabase;

use Illuminate\Foundation\Testing\WithoutMiddleware;

class UserControllerTest extends TestCase

{

public function testGetIndex()

{

Cache::shouldReceive('get')

->once()

->with('key')

->andReturn('value');

$response = $this->get('/users');

// ...

}

}

?>

注意：你不能模拟 Request Facade 。相反，在运行测试时如果需要传入指定参数，请使用 HTTP 辅助函数，比如 get 和 post 。同理，请在测试时通过调用 Config::set 来模拟 Config Facade。

--

--

第十一卷 官方扩展包

--

--

一、Cashier 交易工具包 (Laravel Cashier)

--

1、简介

Laravel Cashier 提供了直观而流畅的接口来接入 Stripe ( https://stripe.com/ ) 付费订阅服务，它可以处理几乎所有的让你头疼的付费订阅代码。除了基本的订阅管理之外，Cashier 还可以帮你处理优惠券、交换订阅、订阅数量、取消宽限期，甚至还可以生成 PDF 电子发票。

{注意} 为防止破坏性的更改，Cashier 使用固定的 Stripe API 版本。Cashier 10.1 利用 Stripe API 2019-08-14 版本。Stripe API 版本将在次要分发上更新以便使用新的 Stripe 特性和改进。

--

2、升级 Cashier

当你升级到新版本 Cashier 时，请务必仔细阅读 Cashier 升级指南 ( https://github.com/laravel/cashier/blob/master/UPGRADE.md )。

--

3、安装

首先，使用 composer 将 Stripe 的 Cashier 包添加到项目依赖中。

// Cashier包安装

composer require laravel/cashier

{注意} 为确保 Cashier 正确的处理所有的 Stripe 事件，请记得 配置 Cashier webhook 处理 ( https://learnku.com/docs/laravel/6.x/billing/5187#handling-stripe-webhooks )。

(1)、数据库迁移

Cashier 服务提供者注册了自己的数据库迁移目录，请记住在安装完包后迁移你的数据库。Cashier 迁移会在你的 users 表中增加几列并创建一个新的 subscriptions 表来保存你所有客户的订阅。

// Cashier数据迁移

php artisan migrate

//

如果你需要覆盖 Cashier 包附带的迁移，可以使用 vendor:publish Artisan 命令发布它们。

php artisan vendor:publish --tag="cashier-migrations"

//

如果你想阻止 Cashier 迁移完全运行，可以使用 Cashier 提供的 ignoreMigrations 。通常，你应该在 AppServiceProvider 的 register 方法中调用这个方法。

use Laravel\Cashier\Cashier;

Cashier::ignoreMigrations();

{注意} Stripe 建议用来存储 stripe 标识符的所有列应该大小写敏感。因此，以 MySQL 数据库为例，你应该确保 stripe\_id 列的列排序规则设置为 utf8\_bin ，更多信息可以在 Stripe 文档 ( https://stripe.com/docs/upgrades#what-changes-does-stripe-consider-to-be-backwards-compatible ) 中找到。

--

4、配置

(1)、Billable 模型

在使用 Cashier 之前， 添加 Billable Trait 到你的模型定义中。 这个 Trait 提供了多种方法以便你执行常见的支付任务，如创建订阅、申请优惠券以及更新支付方式信息。

use Laravel\Cashier\Billable;

class User extends Authenticatable

{

use Billable;

}

Cashier 暂定你的 Billable 模型是 Laravel 附带的 App\User 类，如果你想修改请在你的 .env 文件中指定一个不同的模型。

CASHIER\_MODEL=App\User

{注意} 如果你使用的不是 Laravel 提供的 App\User 模型，你应该发布并更改提供的 migrations ( https://learnku.com/docs/laravel/6.x/billing/5187#installation ) 以匹配你的代替模型数据表名。

(2)、API Keys (API 密钥)

接下来，您需要在 .env 中配置您的 Stripe 密钥。您可以在 Stripe 的控制面板中获取 Stripe API 密钥。

STRIPE\_KEY=your-stripe-key

STRIPE\_SECRET=your-stripe-secret

(3)、货币配置

Cashier 的默认货币是美元 (USD)。您可以设置 CASHIER\_CURRENCY 来更改默认的货币。

CASHIER\_CURRENCY=eur

为了配置 Cashier 使用的货币，您也许需要指定一个额外的 Locale（本地化） 参数，用于格式化在账单上面显示的金额。Cashier 内部使用 PHP的NumberFormatter类 ( https://www.php.net/manual/en/class.numberformatter.php ) 来格式化数字。

CASHIER\_CURRENCY\_LOCALE=nl\_BE

注意：要使用除了 en 之外的 Locale，确保 ext-intl 扩展在服务器上安装并配置好。

--

5、客户

(1)、创建客户

有时，您可能希望在未订阅的情况下创建 Stripe 客户。您可以使用 createAsStripeCustomer 方法完成此操作。

// 创建客户

$user->createAsStripeCustomer();

一旦在 Stripe 中创建了客户，您可以稍后便可开始订阅。

--

6、支付方式

(1)、存储支付方式

为了使用 Stripe 创建订阅或是一次性扣费，您将需要从 Stripe 获取支付方式并储存他的识别码。基于您计划使用的支付方式是用来一次性扣费还是订阅，我们有两种方案来达成，接下来我们一起来看一下这两种方法。

//

///1/ 用于订阅制的支付方法

为了储存用户的信用卡供以后使用，Stripe 的 “Setup Intent” API 需要安全的获取用户的信用卡信息。“Setup Intent” 指示 Stripe 用户支付方式的意象。Cashier 的 Blillable trait 包含了 createSetupIntent 方法来快速创建一个 “Setup Intent”。

您应该在路由或者控制器调用这个方法来获取用户的支付方式意象。

return view('update-payment-method', [

'intent' => $user->createSetupIntent()

]);

当您创建完 “Setup Intent” 并传递给视图后，您应该将它的密钥元素附加到获取用户支付信息的表单中。如下面这个 “更新支付方式” 表单。

<input id="card-holder-name" type="text">

<!-- Stripe Elements Placeholder -->

<div id="card-element"></div>

<button id="card-button" data-secret="{{ $intent->client\_secret }}">

Update Payment Method

</button>

接下来，Stripe.js 库会调用这个密钥并安全地抓取用户的支付信息。

<script src="https://js.stripe.com/v3/"></script>

<script>

const stripe = Stripe('stripe-public-key');

const elements = stripe.elements();

const cardElement = elements.create('card');

cardElement.mount('#card-element');

</script>

然后，卡片会被校验且您可以使用 [Stripe 的 handleCardSetup 方法] ( https://stripe.com/docs/stripe-js/reference#stripe-handle-card-setup )来获取一个安全的“支付方式识别码“。

const cardHolderName = document.getElementById('card-holder-name');

const cardButton = document.getElementById('card-button');

const clientSecret = cardButton.dataset.secret;

cardButton.addEventListener('click', async (e) => {

const { setupIntent, error } = await stripe.handleCardSetup(

clientSecret, cardElement, {

payment\_method\_data: {

billing\_details: { name: cardHolderName.value }

}

}

);

if (error) {

// 显示错误信息

} else {

// 卡片已成功验证

}

});

当 Stripe 完成对客户支付方式的校验后，您可以将 setupIntent.payment\_method 识别码返回给您的应用并附加给客户。这次支付使用的支付方式可 被添加为新的支付方式 ( https://learnku.com/docs/laravel/6.x/billing/5187#adding-payment-methods ) 或者 用于更新默认支付方式 ( https://learnku.com/docs/laravel/6.x/billing/5187#updating-the-default-payment-method )。您可以立即使用这个识别码来 创建一个订阅 ( https://learnku.com/docs/laravel/6.x/billing/5187#creating-subscriptions )。

Tip：如果您希望了解更多有关 “Setup Intent” 与获取收集用户支付详情的信息，请参阅 Stripe的文档 ( https://stripe.com/docs/payments/cards/saving-cards#saving-card-without-payment )。

//

///2/ 用于一次性收费的支付方式

当然，用于一次性收费的支付识别码我们只需要使用一次。由于 Stripe 的限制，您无法使用客户的一次性收费的支付方式。您需要用 Stripe.js 库来让客户填写他们的支付详情。让我们来看一下下面这个支付表单。

<input id="card-holder-name" type="text">

<!-- Stripe Elements Placeholder -->

<div id="card-element"></div>

<button id="card-button">

处理支付流程

</button>

接下来，Stripe.js 可用来附加一个 Stripe 元素用于加密用户的支付方式详情。

<script src="https://js.stripe.com/v3/"></script>

<script>

const stripe = Stripe('stripe-public-key');

const elements = stripe.elements();

const cardElement = elements.create('card');

cardElement.mount('#card-element');

</script>

然后，卡片会被校验且您可以使用 Stripe 的 createPaymentMethod ( https://stripe.com/docs/stripe-js/reference#stripe-create-payment-method ) 来获取一个安全的 “支付方式识别码”。

const cardHolderName = document.getElementById('card-holder-name');

const cardButton = document.getElementById('card-button');

cardButton.addEventListener('click', async (e) => {

const { paymentMethod, error } = await stripe.createPaymentMethod(

'card', cardElement, {

billing\_details: { name: cardHolderName.value }

}

);

if (error) {

// 错误信息

} else {

// 卡片验证成功

}

});

如果卡片验证成功，您可以将 paymentMethod.id 传递给您的应用并进行 一次性付费( https://learnku.com/docs/laravel/6.x/billing/5187#simple-charge )。

(2)、获取支付方式

Billable 模块实例上的 paymentMethods 方法将返回一组 Laravel\Cashier\PaymentMethod 实例。

$paymentMethods = $user->paymentMethods();

要获取默认的支付方式，可以使用 defaultPaymentMethod 方法。

$paymentMethod = $user->defaultPaymentMethod();

(3)、确认用户是否拥有支付方式

要确认一个 Billable 模型是否拥有一个支付方式，使用 hasPaymentMethod 方法。

if ($user->hasPaymentMethod()) {

//

}

(4)、更新默认支付方式 (更新默认的支付方式)

updateDefaultPaymentMethod 可用来更新用户的默认支付方式。这个方法接受 Stripe 的支付方式识别码并且会将新分配的识别码设为默认的支付方式。

$user->updateDefaultPaymentMethod($paymentMethod);

要在 Stripe 内同步您与客户的默认支付方式，您可以使用 updateDefaultPaymentMethodFromStripe 方法。

$user->updateDefaultPaymentMethodFromStripe();

注意：客户的默认支付方式只能用户创建发票或订阅，由于 Stripe 的限制，某些支付方式不适用于一次性收费。

(5)、添加支付方式

要添加新的支付方式，您可以在（可收费的） Billable 用户上调用 addPaymentMethod 方法，并传递支付方式识别码。

$user->addPaymentMethod($paymentMethod);

Tip：要学习如何获取并储存支付方式识别码，请参阅 支付方式储存( https://learnku.com/docs/laravel/6.x/billing/5187#storing-payment-methods )。

(6)、删除支付方式

要删除一个支付方式，你可以调用 Laravel\Cashier\PaymentMethod 上的 delete 方法进行删除。

$paymentMethod->delete();

deletePaymentMethods 方法将删除 Billable 模型的所有相关付款方式的信息。

$user->deletePaymentMethods();

注意：如果用户的订阅比较活跃，那么应该阻止他们删除默认的付款方式。

--

7、订阅

(1)、创建订阅

创建订阅，首先需要获取到一个 Billable 模型实例，这通常是 App\User 的一个实例。一旦您获取了模型实例，您可以使用 newSubscription 方法创建模型的订阅。

$user = User::find(1);

$user->newSubscription('main', 'premium')->create($paymentMethod);

newSubscription 方法的第一个参数应该是订阅的名称。如果您的应用程序只提供一个订阅，那么您可以将其设置为 main or primary。第二个参数是用户订阅的 Stripe 计划。这个值应该与 Stripe 或 Braintree 中的标识符对应。

create 方法接受一个 支付方法标识符( https://learnku.com/docs/laravel/6.x/billing/5187#storing-payment-methods )或者 Stripe PaymentMethod 对象后将开始订阅，并使用客户 ID 和其他相关的账单信息更新数据库。

注意：直接传递支付方法标识符到 create() 方法也可以将用户的付款方式储存到数据库。

//

///1/ 用户其他的详细信息

如果您想要指定用户其他的详细信息，您可以通过将它们作为第二个参数传递给 create 方法。

$user->newSubscription('main', 'monthly')->create($paymentMethod, [

'email' => $email,

]);

要了解更多关于 Stripe 支持的额外字段，请查看 Stripe 的内容创建客户文档( https://stripe.com/docs/api#create\_customer )。

//

///2/ 优惠券

如果您想在创建订阅时使用优惠券，您可以使用 withCoupon 方法。

$user->newSubscription('main', 'monthly')

->withCoupon('code')

->create($paymentMethod);

(2)、检查订阅状态

一旦用户在您的应用程序订阅了，您可以使用各种方便的方法轻松地检查他们的订阅状态。首先，如果用户有一个激活的订阅，那么 subscribed 的方法将返回 true ，即使订阅当前处于试用阶段。

if ($user->subscribed('main')) {

//

}

这个 subscribed 方法还可以在 路由中间件( https://learnku.com/docs/laravel/6.x/middleware )使用，允许您根据用户的订阅状态对路由和控制器进行访问。

public function handle($request, Closure $next)

{

if ($request->user() && ! $request->user()->subscribed('main')) {

// 该用户不是付费客户......

return redirect('billing');

}

return $next($request);

}

如果您想要确定用户是否仍然处于试用阶段，您可以使用 onTrial 方法。这个方法对于向用户显示他们仍然处于试用期的警告是很有用的。

if ($user->subscription('main')->onTrial()) {

//

}

基于给定的 Stripe 计划 ID，可以使用 subscribedToPlan 方法来确定用户是否订阅了该计划。在本例中，我们将确定用户的 main 订阅是否激活了 monthly 计划。

if ($user->subscribedToPlan('monthly', 'main')) {

//

}

recurring 方法可用于确定用户当前是否已订阅，并且不再处于试用阶段。

if ($user->subscription('main')->recurring()) {

//

}

(3)、取消订阅状态

为了确定用户是否曾经订阅，但是已经取消了他们的订阅，您可以使用 cancelled 方法。

if ($user->subscription('main')->cancelled()) {

//

}

您还可以确定用户是否已经取消了订阅，但是仍然处于订阅的「宽限期」，直到订阅完全过期为止。例如，如果用户在 3 月 5 日取消了原定于 3 月 10 日到期的订阅，那么用户将在 3 月 10 日之前进行「宽限期」。请注意，在此期间 subscribed 方法仍然返回 true。

if ($user->subscription('main')->onGracePeriod()) {

//

}

如果要确定用户取消订阅的时间是否已不在其「宽限期」 内，可以使用 ended 方法。

if ($user->subscription('main')->ended()) {

//

}

(4)、不完整和过期状态

如果订阅需要在创建后进行二次付款操作，则订阅将被标记为 incomplete 。订阅状态存储在 stripe\_status Cashier subscriptions 数据库表的列中。

同样，如果在交换计划时需要进行二次付款操作，则订阅将被标记为 past\_due 。当您的订阅处于这两种状态时，在客户确认付款之前，它将不会处于活动状态。检查订阅是否有不完整的付款可以使用 hasIncompletePayment Billable 模型或订阅实例上的方法完成。

if ($user->hasIncompletePayment('main')) {

//

}

if ($user->subscription('main')->hasIncompletePayment()) {

//

}

如果订阅的付款不完整，您应该将用户定向到收银员的付款确认页面，并传递 latestPayment 标识符。您可以使用 latestPayment 订阅实例上的可用方法来检索此标识符。

<a href="{{ route('cashier.payment', $subscription->latestPayment()->id) }}">

Please confirm your payment.

</a>

注意：订阅处于某种 incomplete 状态时，在确认付款之前无法更改。因此，当订阅处于某种状态时， swap 和 updateQuantity 方法将抛出异常 incomplete。

(3)、修改订阅计划

用户在您的应用程序中订阅了之后，他们可能会偶尔想要更改一个新的订阅计划。要将一个用户切换到一个新的订阅，需将订阅计划的标识符传递给 swap 方法。

$user = App\User::find(1);

$user->subscription('main')->swap('provider-plan-id');

如果用户在试用期，试用期的期限会被保留。另外，如果订阅的数量存在「份额」，那么该份额也将保持。

如果你想在更改用户订阅计划的时候取消用户当前订阅的试用期，可以使用 skipTrial 方法。

$user->subscription('main')

->skipTrial()

->swap('provider-plan-id');

如果你想在更改用户订阅计划的时候就为用户开具发票信息，而不是等待下一个结算周期，你可以使用 swapAndInvoice 方法。

$user = App\User::find(1);

$user->subscription('main')->swapAndInvoice('provider-plan-id');

(4)、订阅量

有些时候订阅是会受「数量」影响的。举个例子，你的应用程序的付费方式可能是每个账户 $10 / 月。你可以使用 incrementQuantity 和 decrementQuantity 方法轻松地增加或减少你的订阅量。

$user = User::find(1);

$user->subscription('main')->incrementQuantity();

$user->subscription('main')->incrementQuantity(5); //对当前的订阅量加5;

$user->subscription('main')->decrementQuantity();

$user->subscription('main')->decrementQuantity(5); //对当前的订阅量减5;

或者，你可以使用 updateQuantity 方法设定一个特定的数量。

$user->subscription('main')->updateQuantity(10);

noProrate 方法可用于更新订阅的数量，而不会对收费进行定价。

$user->subscription('main')->noProrate()->updateQuantity(10);

要获得更多关于订阅量的信息，请参考 Stripe 文档 ( https://stripe.com/docs/subscriptions/quantities )。

(5)、订阅税额

在计费模式上实现 taxPercentage 方法，并且返回一个 0 到 100 不超过 2 位小数的数字，用来指定用户在订阅中支付的税率百分比。

public function taxPercentage()

{

return 20;

}

taxPercentage 方法使你能够在模型的基础上应用税率，这对于一个跨越多个国家和税率的用户群可能有帮助。

注意：taxPercentage 方法只适用于付费订阅模式。如果你用 charges 来做「一次性」收费，你需要同时手工指定税率。

//

///1/ 同步税率百分比

当更改 taxPercentage 方法返回的硬编码值时，用户的任何现有订阅的税率设置将保持不变。如果要用返回的 taxPercentage 值更新现有订阅的税率，应在用户的订阅实例上调用 syncTaxPercentage 方法。

$user->subscription('main')->syncTaxPercentage();

(6)、订阅锚定日期

默认情况下，计费周期锚定是创建订阅的日期，如果使用试用期，则是试用结束的日期。如果要修改账单锚定日期，可以使用 anchorBillingCycleOn 方法。

use App\User;

use Carbon\Carbon;

$user = User::find(1);

$anchor = Carbon::parse('first day of next month');

$user->newSubscription('main', 'premium')

->anchorBillingCycleOn($anchor->startOfDay())

->create($paymentMethod);

有关管理订阅计费周期的详细信息，请参阅 Stripe 计划周期文档( https://stripe.com/docs/billing/subscriptions/billing-cycle )。

(7)、取消订阅

在用户订阅上调用 cancel 方法用来取消订阅。

$user->subscription('main')->cancel();

当一个订阅被取消时，Cashier 将会自动在你的数据库中设置 ends\_at 列。这个列经常被用来获悉 subscribed 字段何时应该开始返回 false 。例如，如果客户在 3 月 1 日取消订阅，但是订阅计划直到 3 月 5 日才结束，subscribed 方法将会继续返回 true 一直到 3 月 5 日。

你可以使用 onGracePeriod 方法确定用户是否确定订阅，但是仍然存在一个「宽限期」。

if ($user->subscription('main')->onGracePeriod()) {

//

}

如果你想马上取消订阅，请在用户的订阅中调用 cancelNow 方法。

$user->subscription('main')->cancelNow();

(8)、恢复订阅

如果一个用已经取消订阅，你可以在你希望恢复它的时候使用 resume 方法。用户 必须 仍然在他们的宽限期内才可以恢复订阅。

$user->subscription('main')->resume();

如果用户已取消订阅，然后在订阅宽限期前恢复该订阅，他们将不会被立即计费。相反，他们的订阅将会被重新激活，需要按照原来的支付流程再次进行支付。

--

8、订阅试用 (试用订阅)

(1)、以信用卡订阅 (预付款方式)

如果你想给你的顾客提供试用期，同时收集支付方法信息，那么你应该在创建订阅使用 trialDays 方法。

$user = User::find(1);

$user->newSubscription('main', 'monthly')

->trialDays(10)

->create($paymentMethod);

该方法会在数据库订阅记录上设置订阅期结束时间，以便告知 Sripe 在此之前不要计算用户的账单信息。当使用 trialDays 方法时，Cashier 将覆盖 Stripe 中配置的所有默认试用期。

注意：如果顾客没有在试用期结束前取消订阅，订阅会被自动结算，所以你应该确保告知你的用户他们的试用结束期。

trialUntil 方法允许提供 DateTime 实例指定试用结束期。

use Carbon\Carbon;

$user->newSubscription('main', 'monthly')

->trialUntil(Carbon::now()->addDays(10))

->create($paymentMethod);

你可以使用用户实例的 onTrial 方法或者订阅实例的 onTrial 方法判断用户是否处于试用期。下面两个示例等价。

if ($user->onTrial('main')) {

//

}

if ($user->subscription('main')->onTrial()) {

//

}

(2)、非信用卡订阅 (非预付款方式)

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 列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([

// 填充其他用户属性……

'trial\_ends\_at' => now()->addDays(10),

]);

注意：确保已添加 trial\_ends\_at 日期修改器 ( https://learnku.com/docs/laravel/6.x/eloquent-mutators#date-mutators ) 到模型定义。

Cashier 把这种类型的引用称为「一般体验」，因为它没有关联任何已存在的订阅。如果当前的日期没有超过 trail\_ends\_at 值， User 实例的 onTrial 方法将会返回 true。

if ($user->onTrial()) {

// 用户在他们的试用期内...

}

如果你希望明确的知道用户处于「一般」试用期，并且还未创建实际的订阅，那么你可以使用 onGenericTrial 方法。

if ($user->onGenericTrial()) {

// 用户在他们「一般」试用期...

}

如果你准备给用户创建实际的订阅，通常你可以使用 newSubsription 方法。

$user = User::find(1);

$user->newSubscription('main', 'monthly')->create($paymentMethod);

--

9、处理 Stripe Webhooks

提示：你可以使用 Laravel Valet ( https://learnku.com/docs/laravel/6.x/valet ) 中的 valet share 命令来帮助你在本地开发期间测试 webhook。

Stripe 可以通过 webhooks 通知应用各种各样的事件。 默认情况下，需要定义一个 Cashier 的 webhook 控制器的路由。这个控制器用来处理所有传入的 webhook 请求。

默认情况下，这个控制器将会自动对支付失败次数过多（这个次数可以在 Stripe 设置中定义）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去处理任何你想要处理的 webhook 事件。

为了确保您的应用可以处理 Stripe webhook，请务必在 Stripe 控制面板中配置 webhook URL。Stripe 控制面板中应该配置的所有 webhook 完整列表如下。

customer.subscription.updated

customer.subscription.deleted

customer.updated

customer.deleted

invoice.payment\_action\_required

注意：请确保使用 Cashier 的 webhook 验签( https://learnku.com/docs/laravel/6.x/billing#verifying-webhook-signatures ) 中间件来保护传入请求。

(1)、Webhooks & CSRF 保护

因为 Stripe webhooks 需要绕过 Laraval 的 CSRF 保护( https://learnku.com/docs/laravel/6.x/csrf ), 请确保在您的 VerifyCsrfToken 中间件含有 URI ，或者将其置于 web 中间件组之外。

protected $except = [

'stripe/\*',

];

(2)、定义 Webhook 事件处理程序 (定义 Webhook 事件句柄)

Cashier 对于失败支付自动进行取消订阅，但是如果您有其他的 Stripe Webhook 事件希望去处理，可以扩展 Webhook 控制器。您的方法名应该与 Cashier 期望的约定相符，更具体的说，您希望处理 Stripe webhook 的方法应该以 handle 和 「驼峰」 名为前缀。举例来说，如果您希望处理 invoice.payment\_succeeded 的 webhook，您应该在控制器添加 handleInvoicePaymentSucceeded 方法。

<?php

namespace App\Http\Controllers;

use Laravel\Cashier\Http\Controllers\WebhookController as CashierController;

class WebhookController extends CashierController

{

/\*\*

\* Handle invoice payment succeeded.

\*

\* @param array $payload

\* @return \Symfony\Component\HttpFoundation\Response

\*/

public function handleInvoicePaymentSucceeded($payload)

{

// 此处处理事件

}

}

?>

接下来，在 routes/web.php 文件中定义 Cashier 控制器的路由，这将会覆盖掉默认的路由。

Route::post(

'stripe/webhook',

'\App\Http\Controllers\WebhookController@handleWebhook'

);

(3)、订阅失败

如果用户的信用卡过期怎么办？不用担心 - Cashier 包含了一个 Webhook 控制器可以轻松为您取消用户的订阅。 该控制器会在 Stripe 判断订阅失败后（通常尝试支付失败 3 次及以上）取消用户的订阅。

(4)、验证 Webhook 签名 (Webhook 验签)

为了保护 Webhook，您需要使用 Stripe 的 Webhook 签名 (https://stripe.com/docs/webhooks/signatures)。为了方便，Cashier 包含一个中间件，用于验证传入 Stripe webhook 的请求是否有效。

如果要启用 Webhook 验证，请确保在 .env 配置文件中设置了 STRIPE\_WEBHOOK\_SECRET 的值。 Webhook 的 secret 可以从 Stripe 用户控制面板中找到。

--

10、一次性支付

(1)、简单支付

注意： charge 方法接收您想支付于 应用程序使用的货币的最小单位 的金额。

如果您想对订阅客户的信用卡收取「一次性」费用，可以在可计费模型实例上使用 charge 方法。 您需要将 支付方式识别码 作为第二个参数。

// Stripe 接收分为单位的费用...

$stripeCharge = $user->charge(100, $paymentMethod);

charge 方法接受一个数组作为它的第三个参数，允许您创建支付时将任何您想要的选项传递给底层的 Stripe。有关在创建支付时可用的选项，请参阅 Stripe 文档。

$user->charge(100, $paymentMethod, [

'custom\_option' => $value,

]);

如果支付失败， charge 方法将会抛出异常。如果支付成功，一个 Laravel\Cashier\Payment 实例会从该方法返回。

try {

$payment = $user->charge(100, $paymentMethod);

} catch (Exception $e) {

//

}

(2)、费用与发票 (发票)

有时您可能需要支付一次性费用同时也需要生成费用发票，以便可以向客户提供 PDF 文件格式的收据。 invoiceFor 方法可以让您做到这一点。 例如，向客户开具 5.00 美元的「一次性费用」发票。

// Stripe 接收分为单位的费用...

$user->invoiceFor('One Time Fee', 500);

账单会立刻向用户的默认支付方式收取费用。invoiceFor 方法接收一个数组作为第三个参数，这个数组包含了所购内容的账单选项。接收的第四个参数也是一个数组， 这最后一个参数包含了发票自身的一些选项。

$user->invoiceFor('Stickers', 500, [

'quantity' => 50,

], [

'tax\_percent' => 21,

]);

注意：invoiceFor 方法将会创建 Stripe 发票，该发票将会在支付失败后重试。如果您不想失败后重试，您需要在第一次支付失败后调用 Stripe API 关闭它。

(3)、关于退款

如果您需要处理退款，您可以使用 refund 方法。此方法接受 Stripe Payment Intent ID 作为其第一个参数。

$payment = $user->charge(100, $paymentMethod);

$user->refund($payment->id);

--

11、发票

您可以使用 invoices 方法轻松获取账单模型的发票数组。

$invoices = $user->invoices();

$invoices = $user->invoicesIncludingPending(); //结果包含处理中的发票;

当列出客户发票清单时，可以使用发票辅助函数来显示相关的发票信息。例如，您可能希望在表格中列出每张发票，从而方便客户下载它们。

<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>

(1)、生成电子发票 (生成 PDF 发票)

在路由或控制器中，使用 downloadInvoice 方法生成一个发票的 PDF 下载。这个方法会自动给浏览器生成合适的 HTTP 下载响应。

use Illuminate\Http\Request;

Route::get('user/invoice/{invoice}', function (Request $request, $invoiceId) {

return $request->user()->downloadInvoice($invoiceId, [

'vendor' => 'Your Company',

'product' => 'Your Product',

]);

});

--

12、强客户身份验证 (SCA Strong Customer Authentication)

如果您的业务立足于欧洲，那么您就需要遵守 Strong Customer Authentication (SCA) 法规。这些法规由欧盟发布于 2019 年 9 月，意在预防支付欺诈。 幸运的是， Stripe 与 Cashier 已经准备好构建符合 SCA 的应用。

注意：在您开始之前， 请先阅读 Stripe's guide on PSD2 and SCA ( https://stripe.com/en-be/guides/strong-customer-authentication ) 与 新 SCA API 文档 ( https://stripe.com/docs/strong-customer-authentication )。

(1)、需求额外验证的支付 (需要额外验证的支付)

SCA 通常在处理支付时需要额外的验证步骤。当这种情况出现时， Cashier 会抛出 IncompletePayment 来提示您需要额外的验证步骤。 当异常抛出时，您有两个选择。

您可以引导用户跳转到 Cashier 自带的支付确认页面完成验证。这个页面已经关联由 Cashier 的服务提供者注册的路由。因此，您可以捕获 IncompletePayment 后跳转到支付确认界面。

use Laravel\Cashier\Exceptions\IncompletePayment;

try {

$subscription = $user->newSubscription('default', $planId)

->create($paymentMethod);

} catch (IncompletePayment $exception) {

return redirect()->route(

'cashier.payment',

[$exception->payment->id, 'redirect' => route('home')]

);

}

在支付确认界面，用户则需要再次输入他们的信用卡信息以及进行任何 Stripe 要求的操作。例如输入信用卡的 “3D 安全信息” 来确认支付。当他们完成确认后，用户会被重定向到上方 redirect 定义的 URL。

或者，您可以让 Stripe 来为您处理此次支付确认。在这种情况下，您可以在 Stripe 仪表板设置 自动账单邮箱 ( https://dashboard.stripe.com/account/billing/automatic )。如果捕获到 IncompletePayment，您还是要告知用户他们会在邮箱内收到处理支付的引导邮件。

IncompletePayment 异常会由以下方法抛出：charge, invoiceFor, 和在 Billable（可收费）用户的 invoice 方法。 当处理订阅时， 在 IncompletePayment 中的 create 方法，与在 IncompletePayment 中的 incrementAndInvoice 和 swapAndInvoice 会抛出异常。

//

///1/ 账单未完成与已逾期

每当账单需要额外确认时，订阅在数据库中的 stripe\_status 列留下 incomplete 或 past\_due 状态。当 Webhook 确认支付已完成时，Cashier 会立即激活用户的订阅。

如果您要了解更多关于 incomplete 与 past\_due 状态的信息，请参考额外文档( https://learnku.com/docs/laravel/6.x/billing/5187#incomplete-and-past-due-status )。

(2)、无会话 (Off-session) 支付通知 (非会话支付通知)

尽管用户订阅已经激活，但 SCA 要求用户时不时验证其支付信息。 Cashier 可以发送一封邮件提醒用户需要非会话支付通知。例如，当用户订阅需要续费时， Cashier 的支付通知可以在环境变量中将 CASHIER\_PAYMENT\_NOTIFICATION 分配给一个类来启用。这默认是禁用的。当然，如果您不想使用 Cashier 提供的这个通知类，您可以使用自己的类来完成。

CASHIER\_PAYMENT\_NOTIFICATION=Laravel\Cashier\Notifications\ConfirmPayment

为了确保非会话支付通知送达， 您需要确认已经在 Stripe 仪表盘完成 Stripe Webhook 配置( https://learnku.com/docs/laravel/6.x/billing/5187#handling-stripe-webhooks )和已启用 invoice.payment\_action\_required Webhook。 您的 Billable 模型应使用 Laravel 的 Illuminate\Notifications\Notifiable trait。

注意：当用户手动支付了一个需要额外确认的订单时，通知仍然会被发送。不幸的是，Stripe 无法确认用户是手动支付的还是使用非会话（Off-Session）方式支付。但是，用户确认支付以后返回支付页面，他们可以看见一个简单的 “支付成功” 提示。这样用户就无法在意外情况下再次确认支付，导致二次付费的情况发生。

--

--

二、Dusk 浏览器测试 (Laravel Dusk)

--

https://learnku.com/docs/laravel/6.x/dusk/5188

--

1、介绍

Laravel Dusk 提供了富有表现力、简单易用的浏览器自动化及测试 API 。默认情况下，Dusk 不需要在你的机器上安装 JDK 或者 Selenium 。而是需要使用单独的 ChromeDriver ( https://sites.google.com/a/chromium.org/chromedriver/home ) 进行安装。当然，你也可以自由使用其他的兼容 Selenium 的驱动程序。

--

2、安装

//

///1/ dusk依赖

你应该先向你的 Composer 添加 laravel/dusk 依赖。

composer require --dev laravel/dusk

注意：如果你是手动注册 Dusk 服务提供者，一定 不能 在你的生产环境中注册，这样可能会导致一些不守规矩的用户拥有控制你应用的权限。

//

///2/ dusk安装

安装好 Dusk 包后，运行 dusk:install 命令。

php artisan dusk:install

//

///3/ dusk运行

Browser 目录将会在 tests 目录下被创建，并且包含一个测试用例。接下来，在你的 .env 文件中设置 APP\_URL 变量。这个值应该与你在浏览器中打开本应用的 URL 匹配。要运行测试，使用 dusk 命令。 dusk 命令可以使用与 phpunit 命令同样的参数。

php artisan dusk

//

///4/ dusk失败运行

如果上次运行 dusk 命令时测试失败，则可以通过使用 dusk:fails 命令重新运行失败的测试来节省时间。

php artisan dusk:fails

(1)、管理 Chromedriver 安装

如果您想安装与 Laravel Dusk 附带版本不同的 ChromeDriver，可以使用 dusk:chrome driver 命令。

//

# 为您的操作系统安装最新版本的 ChromeDriver...

php artisan dusk:chrome-driver

//

# 为您的操作系统安装指定版本的 ChromeDriver...

php artisan dusk:chrome-driver 74

//

# 为所有支持的操作系统安装指定版本的 ChromeDriver...

php artisan dusk:chrome-driver --all

{注意} Dusk 要求 ChromeDriver ( https://sites.google.com/a/chromium.org/chromedriver/home ) 二进制文件是可执行的。如果在 Dusk 运行时遇到问题，可以使用以下命令确保二进制文件是可执行的：chmod -R 0755 vendor/laravel/dusk/bin。

(2)、使用其他浏览器

默认情况下， Dusk 使用 Google Chrome 浏览器和一个单独的 ChromeDriver 的安装来运行你的浏览器测试。当然，你可以运行你自己的 Selenium 服务，用任何你想用的浏览器来进行测试。

如果要这么做，打开 tests/DuskTestCase.php 文件，这个是应用测试用例的基类。在这个文件中，你可以移除对 startChromeDriver 方法的调用。这样 Dusk 就不会自动启动 ChromeDriver 了。

/\*\*

\* 准备执行 Dusk 测试。

\*

\* @beforeClass

\* @return void

\*/

public static function prepare()

{

// static::startChromeDriver();

}

然后，你可以修改 driver 方法来连接到你选定的 URL 和端口。此外，你可以修改 「desired capabilities」（期望能力），它将会被传递给 WebDriver。

/\*\*

\* 创建 RemoteWebDriver 实例。

\*

\* @return \Facebook\WebDriver\Remote\RemoteWebDriver

\*/

protected function driver()

{

return RemoteWebDriver::create(

'http://localhost:4444/wd/hub', DesiredCapabilities::phantomjs()

);

}

--

3、开始使用

(1)、创建测试

要创建一个 Dusk 测试，使用 dusk:make 命令。创建的测试将会被放在 tests/Browser 目录。

php artisan dusk:make LoginTest

(2)、运行测试

使用 dusk Artisan 命令来运行你的浏览器测试。

php artisan dusk

如果上次运行 dusk 命令时测试失败，则可以通过使用 dusk:fails 命令重新运行失败的测试来节省时间。

php artisan dusk:fails

dusk 命令接受任何能让 PHPUnit 正常运行的参数。例如，让你可以在指定 group 中运行测试。

php artisan dusk --group=foo

//

///1/ 手动运行 ChromeDriver

默认情况下，Dusk 会尝试自动运行 ChromeDriver。如果你在特定的系统中不能运行，可以在运行 dusk 命令前通过手动的方式来运行 ChromeDriver。 如果你选择手动运行 ChromeDriver，你需要在你的 tests/DuskTestCase.php 文件中注释掉下面这一行。

/\*\*

\* 为 Dusk 测试做准备。

\*

\* @beforeClass

\* @return void

\*/

public static function prepare()

{

// static::startChromeDriver();

}

另外，如果你的 ChromeDriver 运行在非 9515 端口 ，你需要修改同一个类中的 driver 方法。

/\*\*

\* 创建 RemoteWebDriver 实例。

\*

\* @return \Facebook\WebDriver\Remote\RemoteWebDriver

\*/

protected function driver()

{

return RemoteWebDriver::create(

'http://localhost:9515', DesiredCapabilities::chrome()

);

}

(3)、环境处理

为了让 Dusk 使用自己的环境文件来运行测试，你需要在项目根目录创建一个 .env.dusk.{environment} 文件。简单的说，如果你想用 local 环境来运行 dusk 命令，你需要创建一个 .env.dusk.local 文件。

运行测试的时候，Dusk 会备份你的 .env 文件并且重命名你的 Dusk 环境文件为 .env。当测试结束后，它会恢复你的 .env 文件。

(4)、创建浏览器

让我们先来写一个测试用例，这个例子可以验证我们是否能够登录系统。生成测试例子之后，我们可以修改它并让它可以跳转到登录界面，输入登录信息之后，点击「登录」按钮。我们通过 browse 方法来创建一个浏览器实例。

<?php

namespace Tests\Browser;

use App\User;

use Tests\DuskTestCase;

use Laravel\Dusk\Chrome;

use Illuminate\Foundation\Testing\DatabaseMigrations;

class ExampleTest extends DuskTestCase

{

use DatabaseMigrations;

/\*\*

\* 一个基础的浏览器测试用例

\*

\* @return void

\*/

public function testBasicExample()

{

$user = factory(User::class)->create([

'email' => 'taylor@laravel.com',

]);

$this->browse(function ($browser) use ($user) {

$browser->visit('/login')

->type('email', $user->email)

->type('password', 'password')

->press('Login')

->assertPathIs('/home');

});

}

}

?>

在上面的例子中，browse 方法接收了一个回调参数。Dusk 会自动将这个浏览器实例注入到回调过程中，而且这个浏览器实例可以和你的应用进行交互和断言。

//

///1/ 创建多个浏览器

有时候你可能需要多个浏览器才能正确的进行测试。例如，使用多个浏览器测试通过 websockets 进行通讯的在线聊天页面。想要创建多个浏览器，需要在 browse 方法的回调中，用名字来区分浏览器实例，然后传给回调去「申请」多个浏览器实例。

$this->browse(function ($first, $second) {

$first->loginAs(User::find(1))

->visit('/home')

->waitForText('Message');

$second->loginAs(User::find(2))

->visit('/home')

->waitForText('Message')

->type('message', 'Hey Taylor')

->press('Send');

$first->waitForText('Hey Taylor')

->assertSee('Jeffrey Way');

});

//

///2/ 改变浏览器窗口大小

你可以使用 resize 方法去调整浏览器的窗口大小。

$browser->resize(1920, 1080);

maximize 方法可以将浏览器窗口最大化：

$browser->maximize();

(5)、浏览器宏

如果你想定义一个可以在各种测试中重复使用的自定义浏览器方法，可以在 Browser 类中使用 macro 方法。通常，你应该从 服务提供者( https://learnku.com/docs/laravel/6.x/providers ) 的 boot 方法中调用它。

<?php

namespace App\Providers;

use Laravel\Dusk\Browser;

use Illuminate\Support\ServiceProvider;

class DuskServiceProvider extends ServiceProvider

{

/\*\*

\* 注册Dusk的浏览器宏。

\*

\* @return void

\*/

public function boot()

{

Browser::macro('scrollToElement', function ($element = null) {

$this->script("$('html, body').animate({ scrollTop: $('$element').offset().top }, 0);");

return $this;

});

}

}

?>

macro 方法接收一个名称作为第一个参数，第二个参数则是一个闭包。 当调用浏览器宏作为一个 Browser 的实现的方法时，浏览器宏的闭包将会执行。

$this->browse(function ($browser) use ($user) {

$browser->visit('/pay')

->scrollToElement('#credit-card-details')

->assertSee('Enter Credit Card Details');

});

(6)、用户认证 (认证)

你可能经常会测试一些需要认证的页面。你可以使用 Dusk 的 loginAs 方法来避免每个测试都去登陆页面登陆一次。 loginAs 可以使用用户 ID 或者用户模型实例。

$this->browse(function ($first, $second) {

$first->loginAs(User::find(1))

->visit('/home');

});

注意：使用 loginAs 方法后，该用户的 session 将会持久化的供其他测试用例使用。

(7)、数据库迁移

就像上面的认证例子一样，当你的测试用例需要迁移的时候，你不应该使用 RefreshDatabase trait。 RefreshDatabase trait 使用了不适用于 HTTP 请求的数据库事务。取而代之，我们要用 DatabaseMigrations trait。

<?php

namespace Tests\Browser;

use App\User;

use Tests\DuskTestCase;

use Laravel\Dusk\Chrome;

use Illuminate\Foundation\Testing\DatabaseMigrations;

class ExampleTest extends DuskTestCase

{

use DatabaseMigrations;

}

?>

--

4、与元素交互

(1)、Dusk 选择器

选择一个好的 CSS 选择器用于元素交互是编写 Dush 测试最困难的部分之一。随着时间推移，前端的更改可能会导致类似以下的 CSS 选择器中断测试。

// HTML...

<button>Login</button>

// Test...

$browser->click('.login-page .container div > button');

Dusk 选择器让你专注于编写有效的测试，而不是去记忆 CSS 选择器。要定义一个选择器，只需在你的 HTML 元素中添加一个 dusk 属性。然后，在选择器前面添加 @ 去操作 Dusk 测试中的附加元素。

// HTML...

<button dusk="login-button">Login</button>

// Test...

$browser->click('@login-button');

(2)、点击链接

要点击链接的话，你可以在浏览器实例上使用 clickLink 方法。clickLink 方法将会点击指定显示文本的链接。

$browser->clickLink($linkText);

注意：这个方法可以与 jQuery 进行交互。如果页面上没有 jQuery，Dusk 会自动将其注入页面，保证在测试的期间可用。

(3)、文本、值和属性

//

///1/ 检索和设置值

Dusk 提供了几种与当前显示文本、值和属性进行交互的方法。例如，要获取与指定选择器匹配的元素的「值」，请使用 value 方法。

// 检索值...

$value = $browser->value('selector');

// 设置值...

$browser->value('selector', 'value');

//

///2/ 检索文本

text 这个方法可以用来匹配指定选择器中元素的显示文本。

$text = $browser->text('selector');

//

///3/ 检索属性

最后，attribute 这个方法 可以用来匹配指定选择器中元素的属性。

$attribute = $browser->attribute('selector', 'value');

(4)、使用表单 (表单的使用)

//

///1/ 输入值

Dusk 提供了与表单和 input 元素交互的各种方法。首先让我们看一个在 input 框中输入文本的例子。

$browser->type('email', 'taylor@laravel.com');

注意， 虽然 type 方法可以传递 CSS 选择器做为一个参数，但这并不是强制要求的。如果没有提供 CSS 选择器， Dusk 会搜索与 name 属性相同的 input 。如果还是没有找到，Dusk 会尝试查找传入值与 name 属性相同的 textarea。

要想将文本附加到一个字段之后而且不清除其内容， 你可以使用 append 方法。

$browser->type('tags', 'foo')

->append('tags', ', bar, baz');

你可以使用 clear 方法清除输入值。

$browser->clear('email');

//

///2/ 下拉菜单

需要在下拉菜单中选择值，你可以使用 select 方法。 类似于 type 方法， select 方法并不是一定要传入 CSS 选择器。 当使用 select 方法时，你应该传递选项实际的值而不是它的显示文本。

$browser->select('size', 'Large');

你也可以通过省略第二个参数来随机选择一个选项。

$browser->select('size');

//

///3/ 复选框

使用「check」 复选框时，你可以使用 check 方法。 像其他许多与 input 相关的方法，并不是必须传入 CSS 选择器。 如果准确的选择器无法找到的时候，Dusk 会搜索能够与 name 属性匹配的复选框。

$browser->check('terms');

$browser->uncheck('terms');

//

///4/ 单选按钮

使用 「select」中单选按钮选项时，你可以使用 radio 这个方法。 像很多其他的与输入相关的方法一样， 它也并不是必须传入 CSS 选择器。如果准确的选择器无法被找到的时候， Dusk 会搜索能够与 name 属性或者 value 属性相匹配的单选按钮。

$browser->radio('version', 'php7');

(5)、附加文件 (附件)

attach 方法可以附加一个文件到 file input 元素中。 像很多其他的与输入相关的方法一样，他也并不是必须传入 CSS 选择器。如果准确的选择器没有被找到的时候，Dusk 会搜索与 name 属性匹配的文件输入框。

$browser->attach('photo', \_\_DIR\_\_.'/photos/me.png');

{注意} attach 方法需要使用 PHP Zip 扩展，你的服务器必须安装了此扩展。

(6)、使用键盘

keys 方法让你可以再指定元素中输入比 type 方法更加复杂的输入序列。例如，你可以在输入值的同时按下按键。在这个例子中，输入 taylor 时， shift 键也同时被按下。当 taylor 输入完之后， 将会输入 otwell 而不会按下任何按键。

$browser->keys('selector', ['{shift}', 'taylor'], 'otwell');

你甚至可以在你的应用中选中某个元素之后按下「快捷键」。

$browser->keys('.app', ['{command}', 'j']);

{提示} 所有包在 {} 中的键盘按键， 都对应定义于 Facebook\WebDriver\WebDriverKeys 类中，你可以在 GitHub ( https://github.com/facebook/php-webdriver/blob/community/lib/WebDriverKeys.php ) 中找到。

(7)、使用鼠标

//

///1/ 点击元素

click 方法可用于「点击」与给定选择器匹配的元素。

$browser->click('.selector');

//

///2/ 鼠标悬停

mouseover 方法可用于与给定选择器匹配的元素的鼠标悬停动作。

$browser->mouseover('.selector');

//

///3/ 拖放

drag 方法用于将与指定选择器匹配的元素拖到其它元素。

$browser->drag('.from-selector', '.to-selector');

或者，可以在单一方向上拖动元素。

$browser->dragLeft('.selector', 10);

$browser->dragRight('.selector', 10);

$browser->dragUp('.selector', 10);

$browser->dragDown('.selector', 10);

(8)、JavaScript 对话框

Dusk 提供了几种与 JavaScript 对话框交互的方法。

// 等待对话框显示。

$browser->waitForDialog($seconds = null);

// 断言对话框已经显示，并且其消息与给定值匹配。

$browser->assertDialogOpened('value');

// 在打开的 JavaScript 提示对话框中输入给定值。

$browser->typeInDialog('Hello World');

通过点击确定按钮关闭打开的 JavaScript 对话框。

$browser->acceptDialog();

通过点击取消按钮关闭打开的 JavaScript 对话框（仅对确认对话框有效）。

$browser->dismissDialog();

(9)、作用域选择 (选择器作用范围)

有时可能希望在给定的选择器范围内执行多个操作。比如，可能想要断言表格中存在某些文本，然后点击表格中的一个按钮。可以使用 with 方法实现此需求。回调函数内所有被执行的操作都被限定在原始的选择器上。

$browser->with('.table', function ($table) {

$table->assertSee('Hello World')

->clickLink('Delete');

});

(10)、等待元素

在测试大面积使用 JavaScript 的应用时，在进行测试之前，经常需要「等待」指定元素或数据可用。Dusk 使之更容易。使用一系列方法，可以等到页面元素可用，甚至给定的 JavaScript 表达式执行结果为 true。

//

///1/ 等待

如果需要测试暂停指定的毫秒数，可以使用 pause 方法。

$browser->pause(1000);

//

///2/ 等待选择器

waitFor 方法可以用于暂停执行测试，直到页面上与给定 CSS 选择器匹配的元素被显示。默认情况下，将在暂停超过 5 秒后抛出异常。如果有必要，可以传递自定义超时时长作为其第二个参数。

// 等待选择器 5 秒时间...

$browser->waitFor('.selector');

// 等待选择器 1 秒时间...

$browser->waitFor('.selector', 1);

还可以等待指定选择器从页面消失。

$browser->waitUntilMissing('.selector');

$browser->waitUntilMissing('.selector', 1);

//

///3/ 选择器可用时限定作用域范围

偶尔可能希望等待选择器然后与其互动。例如，可能希望等待模态窗口可用，然后点击模态窗口的「确定」按钮。 whenAvailable 方法能够用于这种情况。给定回调内的所有要执行的元素操作都将被限定在起始选择器上。

$browser->whenAvailable('.modal', function ($modal) {

$modal->assertSee('Hello World')

->press('OK');

});

//

///4/ 等待文本

waitForText 方法可以用于等待页面上给定文字被显示。

// 等待指定文本 5 秒时间...

$browser->waitForText('Hello World');

// 等待指定文本 1 秒时间...

$browser->waitForText('Hello World', 1);

//

///5/ 等待链接

waitForLink 方法用于等待给定链接文字在页面上显示

// 等待指定链接 5 秒时间...

$browser->waitForLink('Create');

//等待给定链接 1 秒时间...

$browser->waitForLink('Create', 1);

//

///6/ 等待页面跳转

在给出类似 $browser->assertPathIs('/home') 路径断言时，如果 window.location.pathname 被异步更新，断言就会失败。可以使用 waitForLocation 方法等待页面跳转到给定路径。

$browser->waitForLocation('/secret');

还可以等待被命名的路由跳转。

$browser->waitForRoute($routeName, $parameters);

//

///7/ 等待页面重新加载

如果要在页面重新加载后断言，可以使用 waitForReload 方法。

$browser->click('.some-action')

->waitForReload()

->assertSee('something');

//

///8/ 等待 JavaScript 表达式

有时会希望暂停执行测试，直到给定的 JavaScript 表达式执行结果为 true。可以使用 waitUntil 方法轻易地达成此目的。传送一个表达式给此方法，不需要包含 return 关键字或者结束分号。

// 等待表达式为 true 5 秒时间...

$browser->waitUntil('App.dataLoaded');

$browser->waitUntil('App.data.servers.length > 0');

// 等待表达式为 true 1 秒时间...

$browser->waitUntil('App.data.servers.length > 0', 1);

//

///9/ 等待 Vue 表达式

下述方法可用于一直等待，直到一个给定的 Vue 组件属性有特定的值。

// 一直等待，直到组件属性有特定的值

$browser->waitUntilVue('user.name', 'Taylor', '@user');

// 一直等待，直到组件不包含给定的值

$browser->waitUntilVueIsNot('user.name', null, '@user');

//

///10/ 使用回调函数来等待

Dusk 中的许多 "wait" 方法都依赖于底层方法 waitUsing。你可以直接用这个方法去等待一个回调函数返回 true。waitUsing 方法接收一个最大的等待秒数，闭包执行的内部时间，闭包，以及一个可选的失败信息。

$browser->waitUsing(10, 1, function () use ($something) {

return $something->isReady();

}, "Something wasn't ready in time.");

(11)、Vue 组件断言 (创建 Vue 断言)

创建 Vue 断言

Dusk 甚至还允许你对 Vue 组件数据的状态进行断言。比如：假设你的应用有如下的 Vue 组件。

// HTML...

<profile dusk="profile-component"></profile>

// 组件的定义

Vue.component('profile', {

template: '<div>{{ user.name }}</div>',

data: function () {

return {

user: {

name: 'Taylor'

}

};

}

});

你可以像这样对 Vue 组件状态进行断言。

/\*\*

\* 一个基础的 Vue 测试示例

\*

\* @return void

\*/

public function testVue()

{

$this->browse(function (Browser $browser) {

$browser->visit('/')

->assertVue('user.name', 'Taylor', '@profile-component');

});

}

5、可用的断言

Dusk 提供了各种你可以对应用使用的断言。所有可用的断言列表如下。

(1)、assertTitle

断言页面的标题（title）为给定值。

$browser->assertTitle($title);

(2)、assertTitleContains

断言页面标题（title）包含给定值。

$browser->assertTitleContains($title);

(3)、assertUrlIs

断言当前的 URL（不包含 query 字符串）满足给定的字符串。

$browser->assertUrlIs($url);

(4)、assertSchemeIs

断言当前的 URL scheme 满足给定的 scheme。

$browser->assertSchemeIs($scheme);

(5)、assertSchemeIsNot

断言当前的 URL scheme 不满足给定的 scheme。

$browser->assertSchemeIsNot($scheme);

(6)、assertHostIs

断言当前的 URL host 满足给定的 host。

$browser->assertHostIs($host);

(7)、assertHostIsNot

断言当前的 URL host 不满足给定的 host。

$browser->assertHostIsNot($host);

(8)、assertPortIs

断言当前的 URL 端口满足给定的端口。

$browser->assertPortIs($port);

(9)、assertPortIsNot

断言当前的 URL 端口不满足给定的端口。

$browser->assertPortIsNot($port);

(10)、assertPathBeginsWith

断言当前的 URL 以给定的字符串开头。

$browser->assertPathBeginsWith($path);

(11)、assertPathIs

断言当前的路径满足给定的路径。

$browser->assertPathIs('/home');

(12)、assertPathIsNot

断言当前的路径不满足给定的路径。

$browser->assertPathIsNot('/home');

(13)、assertRouteIs

断言给定的 URL 不满足给定的命名路由的 URL。

$browser->assertRouteIs($name, $parameters);

(14)、assertQueryStringHas

断言给定的 query 参数存在。

$browser->assertQueryStringHas($name);

断言给定的 query 参数存在，并且是给定的值。

$browser->assertQueryStringHas($name, $value);

(15)、assertQueryStringMissing

断言给定的 query 参数不存在。

$browser->assertQueryStringMissing($name);

(16)、assertFragmentIs

断言当前的 fragment 满足给定的 fragment。

$browser->assertFragmentIs('anchor');

(17)、assertFragmentBeginsWith

断言当前的 fragment 以给定的 fragment 开头。

$browser->assertFragmentBeginsWith('anchor');

(18)、assertFragmentIsNot

断言当前的 fragment 不满足给定的 fragment。

$browser->assertFragmentIsNot('anchor');

(19)、assertHasCookie

断言给定的 cookie 存在。

$browser->assertHasCookie($name);

(20)、assertCookieMissing

断言给定的 cookie 不存在。

$browser->assertCookieMissing($name);

(21)、assertCookieValue

断言 cookie 是给定的值。

$browser->assertCookieValue($name, $value);

(22)、assertPlainCookieValue

断言未加密的 cookie 有给定的值。

$browser->assertPlainCookieValue($name, $value);

(23)、assertSee

断言在页面中有给定的文本。

$browser->assertSee($text);

(24)、assertDontSee

断言在页面中没有给定的文本。

$browser->assertDontSee($text);

(25)、assertSeeIn

断言在选择器中有给定的文本。

$browser->assertSeeIn($selector, $text);

译者注：这里的选择器，应该是 DOM 选择器。

(26)、assertDontSeeIn

断言在选择器中不存在给定的字符串。

$browser->assertDontSeeIn($selector, $text);

(27)、assertSourceHas

断言在页面中存在给定的源码。

$browser->assertSourceHas($code);

(28)、assertSourceMissing

断言页面中没有给定的源码。

$browser->assertSourceMissing($code);

(29)、assertSeeLink

断言在页面中存在指定的链接。

$browser->assertSeeLink($linkText);

(30)、assertDontSeeLink

断言页面中没有指定的链接。

$browser->assertDontSeeLink($linkText);

(31)、assertInputValue

断言输入框（input）有给定的值。

$browser->assertInputValue($field, $value);

(32)、assertInputValueIsNot

断言输入框没有给定的值。

$browser->assertInputValueIsNot($field, $value);

(33)、assertChecked

断言复选框（checkbox）被选中。

$browser->assertChecked($field);

(34)、assertNotChecked

断言复选框没有被选中。

$browser->assertNotChecked($field);

(35)、assertRadioSelected

断言单选框（radio）被选中。

$browser->assertRadioSelected($field, $value);

(36)、assertRadioNotSelected

断言单选框没有被选中。

$browser->assertRadioNotSelected($field, $value);

(37)、assertSelected

断言下拉框有给定的值。

$browser->assertSelected($field, $value);

(38)、assertNotSelected

断言下拉框没有给定的值。

$browser->assertNotSelected($field, $value);

(39)、assertSelectHasOptions

断言给定的数组值是可选的。

$browser->assertSelectHasOptions($field, $values);

(40)、assertSelectMissingOptions

断言给定的数组值是不可选的。

$browser->assertSelectMissingOptions($field, $values);

(41)、assertSelectHasOption

断言给定的值在给定的地方是可供选择的。

$browser->assertSelectHasOption($field, $value);

(42)、assertValue

断言选择器范围内的元素存在指定的值。

$browser->assertValue($selector, $value);

(43)、assertVisible

断言选择器范围内的元素可见。

$browser->assertVisible($selector);

(44)、assertPresent

断言选择器范围内的元素是存在的。

$browser->assertPresent($selector);

(45)、assertMissing

断言选择器范围内的元素不存在。

$browser->assertMissing($selector);

(46)、assertDialogOpened

断言含有指定消息的 JavaScript 对话框已经打开。

$browser->assertDialogOpened($message);

(47)、assertEnabled

断言指定的字段是启用的。

$browser->assertEnabled($field);

(48)、assertDisabled

断言指定的字段是停用的。

$browser->assertDisabled($field);

(49)、assertButtonEnabled

断言指定的按钮是启用的。

$browser->assertButtonEnabled($button);

(50)、assertButtonDisabled

断言指定的按钮是停用的。

$browser->assertButtonDisabled($button);

(51)、assertFocused

断言焦点在于指定的字段。

$browser->assertFocused($field);

(52)、assertNotFocused

断言焦点不在指定的字段。

$browser->assertNotFocused($field);

(53)、assertVue

断言 Vue 组件数据的属性匹配指定的值。

$browser->assertVue($property, $value, $componentSelector = null);

(54)、assertVueIsNot

断言 Vue 组件数据的属性不匹配指定的值。

$browser->assertVueIsNot($property, $value, $componentSelector = null);

(55)、assertVueContains

断言 Vue 组件数据的属性是一个数组，并且该数组包含指定的值。

$browser->assertVueContains($property, $value, $componentSelector = null);

(56)、assertVueDoesNotContain

断言 Vue 组件数据的属性是一个数组，并且该数组不包含指定的值。

$browser->assertVueDoesNotContain($property, $value, $componentSelector = null);

--

6、页面

有时候，需要测试一系列复杂的动作，这会使得测试代码难以阅读和理解。通过页面可以定义出语义化的动作，然后在指定页面中可以使用单个方法。页面还可以定义应用或单个页面通用选择器的快捷方式。

(1)、创建页面 (生成页面)

dusk:page Artisan 命令可以生成页面对象。所有的页面对象都位于 tests/Browser/Pages 目录。

// 创建页面

php artisan dusk:page Login

(2)、配置页面

页面默认拥有 3 个方法： url， assert 和 elements。 在这里我们先详述 url 和 assert 方法， elements 方法将会 选择器简写( https://learnku.com/docs/laravel/6.x/dusk/5188#shorthand-selectors )中详述。

//

///1/ url 方法

url 方法应该返回表示页面 URL 的路径。 Dusk 将会在浏览器中使用这个 URL 来导航到具体页面。

/\*\*

\* 获得页面 URL 路径。

\*

\* @return string

\*/

public function url()

{

return '/login';

}

//

///2/ assert 方法

assert 方法可以作出任何断言来验证浏览器是否在指定页面上。这个方法并不是必须的。你可以根据你自己的需求来做出这些断言。这些断言会在你导航到这个页面的时候自动执行。

/\*\*

\* 断言浏览器当前处于指定页面。

\*

\* @return void

\*/

public function assert(Browser $browser)

{

$browser->assertPathIs($this->url());

}

(3)、导航至页面

一旦页面配置好之后，你可以使用 visit 方法导航至页面。

use Tests\Browser\Pages\Login;

$browser->visit(new Login);

有时候，你可能已经在指定页面了，你需要的只是「加载」当前页面的选择器和方法到当前测试中来。常见的例子有：当你按下一个按钮的时候，你会被重定向至指定页面，而不是直接导航至指定页面。在这种情况下，你需要使用 on 方法来加载页面。

use Tests\Browser\Pages\CreatePlaylist;

$browser->visit('/dashboard')

->clickLink('Create Playlist')

->on(new CreatePlaylist)

->assertSee('@create');

(4)、简写选择器 (选择器简写)

elements 方法允许你为页面中的任何 CSS 选择器定义简单易记的简写。例如，让我们为应用登录页中的 email 输入框定义一个简写。

/\*\*

\* 获取页面的元素简写。

\*

\* @return array

\*/

public function elements()

{

return [

'@email' => 'input[name=email]',

];

}

现在你可以用这个简写来代替之前在页面中使用的完整 CSS 选择器。

$browser->type('@email', 'taylor@laravel.com');

//

///1/ 全局的选择器简写

安装 Dusk 之后，Page 基类存放在你的 tests/Browser/Pages 目录。该类中包含一个 siteElements 方法，这个方法可以用来定义全局的选择器简写，这样在你应用中每个页面都可以使用这些全局选择器简写了。

/\*\*

\* 获取站点全局的选择器简写。

\*

\* @return array

\*/

public static function siteElements()

{

return [

'@element' => '#selector',

];

}

(5)、页面方法

除了页面中已经定义的默认方法之外，你还可以定义在整个测试过程中会使用到的其他方法。例如，假设我们正在开发一个音乐管理应用，在应用中都可能需要一个公共的方法来创建列表，而不是在每一页、每一个测试类中都重写一遍创建播放列表的逻辑，这时候你可以在你的页面类中定义一个 createPlaylist 方法。

<?php

namespace Tests\Browser\Pages;

use Laravel\Dusk\Browser;

class Dashboard extends Page

{

// 其他页面方法...

/\*\*

\* 创建一个新的播放列表。

\*

\* @param \Laravel\Dusk\Browser $browser

\* @param string $name

\* @return void

\*/

public function createPlaylist(Browser $browser, $name)

{

$browser->type('name', $name)

->check('share')

->press('Create Playlist');

}

}

?>

方法被定义之后，你可以在任何使用到该页的测试中使用这个方法了。浏览器实例会自动传递该页面方法。

use Tests\Browser\Pages\Dashboard;

$browser->visit(new Dashboard)

->createPlaylist('My Playlist')

->assertSee('My Playlist');

--

7、组件

组件类似于 Dusk 的 「页面对象」，不过它更多的是贯穿整个应用程序中频繁重用的 UI 和功能片断，比如说导航条或信息通知弹窗。因此，组件并不会绑定于某个明确的 URL。

(1)、创建组件 (组件的生成)

为了生成一个组件，使用 Artisan 命令 dusk:component 即可生成组件。新生成的组件位于 test/Browser/Components 目录中。

// 组件创建

php artisan dusk:component DatePicker

如上所示，这是生成一个 「日期选择器」（date picker） 组件的示例，这个组件可能会贯穿使用在你应用程序的许多页面中。在整个测试套件的大量测试页面中，手动编写日期选择的浏览器自动化逻辑会非常麻烦。 更方便的替代办法是，定义一个表示日期选择器的 Dusk 组件，然后把自动化逻辑封装在该组件内。

<?php

namespace Tests\Browser\Components;

use Laravel\Dusk\Browser;

use Laravel\Dusk\Component as BaseComponent;

class DatePicker extends BaseComponent

{

/\*\*

\* 获取组件的 root selector

\*

\* @return string

\*/

public function selector()

{

return '.date-picker';

}

/\*\*

\* 浏览器包含组件的断言

\*

\* @param Browser $browser

\* @return void

\*/

public function assert(Browser $browser)

{

$browser->assertVisible($this->selector());

}

/\*\*

\* 读取组件的元素快捷方式

\*

\* @return array

\*/

public function elements()

{

return [

'@date-field' => 'input.datepicker-input',

'@month-list' => 'div > div.datepicker-months',

'@day-list' => 'div > div.datepicker-days',

];

}

/\*\*

\* 选择给定日期

\*

\* @param \Laravel\Dusk\Browser $browser

\* @param int $month

\* @param int $day

\* @return void

\*/

public function selectDate($browser, $month, $day)

{

$browser->click('@date-field')

->within('@month-list', function ($browser) use ($month) {

$browser->click($month);

})

->within('@day-list', function ($browser) use ($day) {

$browser->click($day);

});

}

}

?>

(2)、使用组件 (组件的使用)

组件定义一旦完成，在任何测试页面的日期选择器中选定一个日期就很轻松了。并且，如果需要修改选定日期的逻辑，仅修改该组件即可。

<?php

namespace Tests\Browser;

use Tests\DuskTestCase;

use Laravel\Dusk\Browser;

use Tests\Browser\Components\DatePicker;

use Illuminate\Foundation\Testing\DatabaseMigrations;

class ExampleTest extends DuskTestCase

{

/\*\*

\* 基本的组件测试示例

\*

\* @return void

\*/

public function testBasicExample()

{

$this->browse(function (Browser $browser) {

$browser->visit('/')

->within(new DatePicker, function ($browser) {

$browser->selectDate(1, 2018);

})

->assertSee('January');

});

}

}

?>

--

8、持续集成

(1)、CircleCI

如果使用 CircleCI 运行 Dusk 测试，则可以参考此配置文件作为起点。与 TravisCI 一样，我们将使用 php artisan serve 命令启动 PHP 的内置 Web 服务器。

//

//// 启动PHP内置Web服务器

version: 2

jobs:

build:

steps:

- run: sudo apt-get install -y libsqlite3-dev

- run: cp .env.testing .env

- run: composer install -n --ignore-platform-reqs

- run: npm install

- run: npm run production

- run: vendor/bin/phpunit

- run:

name: Start Chrome Driver

command: ./vendor/laravel/dusk/bin/chromedriver-linux

background: true

- run:

name: Run Laravel Server

command: php artisan serve

background: true

- run:

name: Run Laravel Dusk Tests

command: php artisan dusk

(2)、Codeship

在 Codeship ( https://codeship.com/ ) 中运行 Dusk 测试，需要在你的 Codeship 项目中添加以下命令。当然，这些命令只是作为范例，你可以根据需要随意添加额外的命令。

phpenv local 7.2

cp .env.testing .env

mkdir -p ./bootstrap/cache

composer install --no-interaction --prefer-dist

php artisan key:generate

nohup bash -c "php artisan serve 2>&1 &" && sleep 5

php artisan dusk

(3)、Heroku CI

在 Heroku CI ( https://www.heroku.com/continuous-integration ) 中运行 Dusk 测试时，请将下列 Google Chrome 构建包和脚本添加到你的 Heroku app.json 文件中。

{

"environments": {

"test": {

"buildpacks": [

{ "url": "heroku/php" },

{ "url": "https://github.com/heroku/heroku-buildpack-google-chrome" }

],

"scripts": {

"test-setup": "cp .env.testing .env",

"test": "nohup bash -c './vendor/laravel/dusk/bin/chromedriver-linux > /dev/null 2>&1 &' && nohup bash -c 'php artisan serve > /dev/null 2>&1 &' && php artisan dusk"

}

}

}

}

(4)、Travis CI

在 Travis CI ( https://travis-ci.org/ ) 中运行 Dusk 测试时，您可以参考 .travis.yml 的配置。由于 Travis CI 并不是一个图形化的环境，我们还需要一些额外的操作步骤以便启动 Chrome 浏览器。 此外，我们将会使用 php artisan serve 来启动 PHP 自带的 Web 服务器。

~~~ S ~~~

language: php

php:

- 7.3

addons:

chrome: stable

install:

- cp .env.testing .env

- travis\_retry composer install --no-interaction --prefer-dist --no-suggest

- php artisan key:generate

before\_script:

- google-chrome-stable --headless --disable-gpu --remote-debugging-port=9222 http://localhost &

- php artisan serve &

script:

- php artisan dusk

~~~ e ~~~

(5)、GitHub Actions

如果您使用 Github Actions ( https://github.com/features/actions ) 来运行 Dusk 测试时，则可以使用此配置文件作为配置起点。像 TravisCI 一样。我们将使用 php artisan serve 命令来启动 PHP 的 Web 服务器。

~~~ s ~~~

name: CI

on: [push]

jobs:

dusk-php:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v1

- name: Prepare The Environment

run: cp .env.example .env

- name: Install Composer Dependencies

run: composer install --no-progress --no-suggest --prefer-dist --optimize-autoloader

- name: Generate Application Key

run: php artisan key:generate

- name: Upgrade Chrome Driver

run: php artisan dusk:chrome-driver

- name: Start Chrome Driver

run: ./vendor/laravel/dusk/bin/chromedriver-linux > /dev/null 2>&1 &

- name: Run Laravel Server

run: php artisan serve > /dev/null 2>&1 &

- name: Run Dusk Tests

run: php artisan dusk

~~~ e ~~~

在 .env.testing 文件中，调整 APP\_URL 的配置。

APP\_URL=http://127.0.0.1:8000

--

--

三、Envoy 部署工具

--

https://learnku.com/docs/laravel/6.x/envoy/5189

--

1、简介

Laravel Envoy ( https://github.com/laravel/envoy ) 为定义远程服务器的日常任务，提供了一套简洁、轻量的语法。Blade 风格语法即可实现部署任务的配置、Artisan 命令的执行等。目前，Envoy 仅支持 Mac 和 Linux 操作系统。

(1)、安装

首先，运行 Composer 的 global require 命令进行全局安装 Envoy。

composer global require laravel/envoy

由于全局 Composer 库有时会导致包版本冲突，因此您可能希望考虑使用 cgr ，它是 composer global require 命令的直接替代品。 cgr 库的安装说明可参阅 found on GitHub ( https://github.com/consolidation-org/cgr )。

注意：确保将 〜/ .composer / vendor / bin 目录放在 PATH 中，以便在终端中运行 envoy 命令时找到 envoy 可执行文件。

(2)、更新 Envoy

您也可以使用 Composer 使您的 Envoy 安装保持最新。 使用 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

你的所有 Envoy 任务都应该在项目根目录中的 Envoy.blade.php 文件中定义。 这是一个让你入门的例子。

@servers(['web' => ['user@192.168.1.1']])

@task('foo', ['on' => 'web'])

ls -la

@endtask

如您所见，在文件顶部定义了一个 @server 数组，允许您在任务声明的 on 选项中引用这些服务器。 在你的 @ task 声明中，你应该放置在任务执行时应该在你的服务器上运行的 Bash 代码。

您可以通过将服务器的 IP 地址指定为 127.0.0.1 来强制脚本在本地运行。

@servers(['localhost' => '127.0.0.1'])

(1)、配置

有时，您可能需要在执行 Envoy 任务之前执行一些 PHP 代码。 您可以使用 @ setup 指令声明变量，并在执行任何其他任务之前执行其他常规 PHP 工作。

@setup

$now = new DateTime();

$environment = isset($env) ? $env : "testing";

@endsetup

如果在执行任务之前需要其他 PHP 文件，可以在 Envoy.blade.php 文件的顶部使用 @include 指令。

@include('vendor/autoload.php')

@task('foo')

# ...

@endtask

(2)、变量

如果需要，您可以使用命令行将选项值传递到 Envoy 任务。

envoy run deploy --branch=master

您可以通过 Blade 的「echo」 语法访问任务中的选项。您也可以在任务中使用 if 语句和循环。 例如，在执行 git pull 命令之前，让我们验证 $branch 变量的存在。

@servers(['web' => '192.168.1.1'])

@task('deploy', ['on' => 'web'])

cd site

@if ($branch)

git pull origin {{ $branch }}

@endif

php artisan migrate

@endtask

(3)、Stories

Stories 将一组任务分组在一个方便的名称下，允许您将小型，集中的任务分组到大型任务中。例如，deploy 故事可以通过在其定义中列出任务名称来运行 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

一旦编写了 story ，你可以像一个典型的任务一样运行它。

envoy run deploy

(4)、多服务器

Envoy 允许您轻松跨多个服务器运行任务。 首先，在 @server 声明中添加其他服务器。 应为每个服务器分配一个唯一的名称。 定义了其他服务器后，列出任务的 on 数组中的每个服务器。

@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2']])

cd site

git pull origin {{ $branch }}

php artisan migrate

@endtask

//

///1/ 并行运行

默认情况下，将在每个服务器上串行执行任务。 换句话说，任务将在继续在第二台服务器上执行之前在第一台服务器上完成运行。 如果您想并行运行多个服务器上的任务，请在任务声明中添加 parallel 选项。

@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

要运行在 Envoy.blade.php 文件中定义的任务或故事，请执行 Envoy 的 run 命令，传递您要执行的任务或故事的名称。 当任务运行时， Envoy 将运行任务并显示服务器的输出。

// 运行任务

envoy run deploy

(1)、任务确认

如果您希望在服务器上运行给定任务之前提示您进行确认，则应将 confirm 指令添加到任务声明中。 此选项对于破坏性操作特别有用。

@task('deploy', ['on' => 'web', 'confirm' => true])

cd site

git pull origin {{ $branch }}

php artisan migrate

@endtask

--

4、消息通知

(1)、Slack

Envoy 还支持在执行每个任务后向 Slack ( https://slack.com/ ) 发送通知。 @slack 指令接受 Slack 钩子 URL 和通道名称。 您可以通过在 Slack 控制面板中创建 Incoming WebHooks 集成来检索您的 webhook URL 。 您应该将整个 webhook URL 传递给 @slack 指令。

@finished

@slack('webhook-url', '#bots')

@endfinished

你可以提供以下其中一项作为频道参数：给频道发送通知，#channel；给用户发送通知，@user。

(2)、Discord

Envoy 还支持在执行每项任务后向 Discord ( https://discord.com/ ) 发送通知。 @ discord 指令接受 Discord 钩子 URL 和消息。 您可以通过在 Webhook 中创建 Webhook 并选择 webhook 应发布到哪个频道来检索您的 webhook URL。 您应该将整个 Webhook URL 传递到 @ discord 指令。

@finished

@discord('discord-webhook-url')

@endfinished

--

四、 Horizon 队列管理工具 (Laravel Horizon)

--

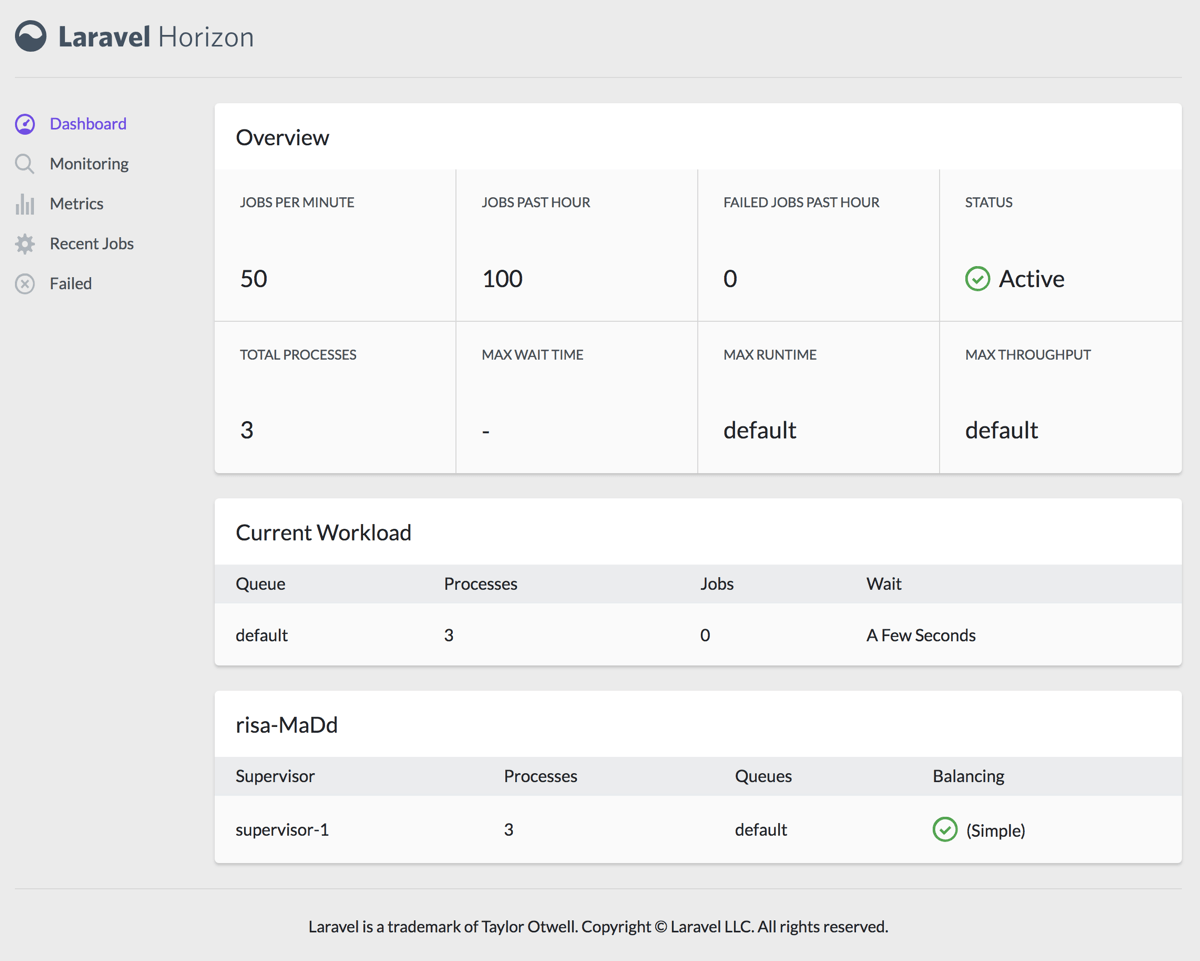
https://learnku.com/docs/laravel/6.x/horizon/5190

--

1、简介

Horizon 提供了一个漂亮的仪表盘，并且可以通过代码配置你的 Laravel Redis 队列，同时它允许你轻易的监控你的队列系统中诸如任务吞吐量，运行时间和失败任务等关键指标。

所有的配置项都存放在一个简单的配置文件中，从而允许你将它放在团队的协同版本控制中，方便团队协作。



( 图 https://cdn.learnku.com/uploads/images/201909/04/4760/wfych538yj.png!large )

--

2、安装

注意：你应该确保 queue 配置文件中设置了 redis 队列驱动。

你应该使用 Composer 来为你的 Laravel 项目安装 Horizon。

///1/ 安装Horizon (需要php的pcntl的扩展)

// composer require laravel/horizon //^4.3版本安装错误;

composer require "laravel/horizon:~3"

//

///2/ 发布Horizon

安装完成后，使用 horizon:install 发布 Artisan 命令。

php artisan horizon:install

//

///3/ 创建 failed\_jobs 表

你还应该创建 failed\_jobs 表，Laravel 将使用该表来存储任何 failed queue jobs ( https://learnku.com/docs/laravel/6.x/queues#dealing-with-failed-jobs )。

php artisan queue:failed-table

php artisan migrate

(0)、升级 Horizon

当升级到 Horizon 的一个新的主要版本时，你必须仔细检查 the upgrade guide( https://github.com/laravel/horizon/blob/master/UPGRADE.md )。

此外，你应该重新发布 Horizon 的资源文件。

php artisan horizon:assets

(1)、配置

发布 Horizon 相关文件后，他的主要配置文件会放在 config/horizon.php。你可以在这个文件中配置队列相关选项，并且每个配置项都有详细的使用说明，请详细阅读此文件。

注意：你应确保 horizon 配置文件的 environments 部分包含你计划在其上运行 Horizon 的每个环境的条目。

//

///1/ 均衡配置

Horizon 提供了三种均衡策略： simple ， auto ，和 false 。默认的是 simple , 会将收到的任务均分给队列进程。

'balance' => 'simple',

auto 策略会根据当前的工作量调整每个队列的工作进程任务数量。例如：如果 notifications 队列有 1000 个待执行任务，但是你的 render 队列是空的，Horizon 会分配更多的工作进程给 notifications 队列，直到 notifications 队列中所有任务执行完成。当配置项 balance 配置为 false ，Horizon 会使用 Laravel 默认执行行为，它将按照配置中列出的顺序处理队列任务。

当使用 auto 策略时，您可以定义 minProcesses 和 maxProcesses 配置选项，以控制最小和最大进程数范围应向上和向下扩展到。

'environments' => [

'production' => [

'supervisor-1' => [

'connection' => 'redis',

'queue' => ['default'],

'balance' => 'auto',

'minProcesses' => 1,

'maxProcesses' => 10,

'tries' => 3,

],

],

],

//

///2/ 任务修整

horizon 配置文件允许你配置应保留最近和失败任务的时间（以分钟为单位）。 默认情况下，最近的任务保留一小时，而失败的任务保留一周。

'trim' => [

'recent' => 60,

'failed' => 10080,

],

(2)、仪表盘授权 (仪表盘权限验证)

Horizon 仪表盘路由是 /horizon。 默认情况下，你只能在 local 环境下访问仪表盘。在你的 app/Providers/HorizonServiceProvider.php 文件中，有一个 gate 方法。这里授权控制 非本地 环境中对 Horizon 的访问。 你可以根据需要随意修改此门面，以限制对 Horizon 安装的访问。

/\*\*

\* Register the Horizon gate.

\*

\* This gate determines who can access Horizon in non-local environments.

\*

\* @return void

\*/

protected function gate()

{

Gate::define('viewHorizon', function ($user) {

return in\_array($user->email, [

'taylor@laravel.com',

]);

});

}

注意：记住，Laravel 会自动将 authenticated 用户注入到 gate 方法。如果您的应用程序通过其他方法（如 IP 限制）提供 Horizon 安全，那么 Horizon 用户可能不需要 “登录”。因此，您需要将上面的 function ($user) 更改为 function ($user = null) 以使 Laravel 不需要进行身份验证。

--

4、运行 Horizon

当在 config/horizon.php 文件中配置好了你的队列执行进程后，你就可以使用 horizon Artisan 命令启动 Horizon。只需要一条命令语句即可启动所有配置好的队列进程。

//

///1/ 运行Horizon

php artisan horizon

//

你也可以使用 horizon:pause 和 horizon:continue Artisan 命令来暂停或继续执行队列任务。

//

///2/ 暂停Horizon

php artisan horizon:pause

//

///3/ 继续Horizon

php artisan horizon:continue

//

你可以使用 horizon:terminate Artisan 命令优雅的终止 Horizon 主进程。Horizon 会把正在执行的任务处理完毕后退出。

//

///4/ 终止Horizon

php artisan horizon:terminate

(1)、部署 Horizon

如果你将 Horizon 部署到线上服务器时，则需要配置一个进程监控器来检测 php artisan horizon 命令，在它意外退出时自动重启。上线新代码时则需要该进程监控器终止 Horizon 进程并以修改后的代码重启 Horizon。

//

///1/ Supervisor 配置

如果你使用 Supervisor 进程监控器管理你的 horizon 进程，那么以下配置文件则可满足需求。

[program:horizon]

process\_name=%(program\_name)s

command=php /home/forge/app.com/artisan horizon

autostart=true

autorestart=true

user=forge

redirect\_stderr=true

stdout\_logfile=/home/forge/app.com/horizon.log

Tip：如果你不喜欢自己维护服务器，可以考虑使用 Laravel Forge ( https://forge.laravel.com/ )。Forge 提供了运行一个带有 Horizon 的现代、强大的 Laravel 应用所需要的 PHP7+ 及其他所有环境。

--

5、标签

Horizon 允许你对任务分配「标签」，包括邮件，事件广播，通知和队列的事件监听器。事实上，Horizon 会智能并且自动根据任务携带 Eloquent 模型给大多数任务标记标签，如下任务示例。

<?php

namespace App\Jobs;

use App\Video;

use Illuminate\Bus\Queueable;

use Illuminate\Queue\SerializesModels;

use Illuminate\Queue\InteractsWithQueue;

use Illuminate\Contracts\Queue\ShouldQueue;

use Illuminate\Foundation\Bus\Dispatchable;

class RenderVideo implements ShouldQueue

{

use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

/\*\*

\* video 实例

\*

\* @var \App\Video

\*/

public $video;

/\*\*

\* 创建工作实例

\*

\* @param \App\Video $video

\* @return void

\*/

public function \_\_construct(Video $video)

{

$this->video = $video;

}

/\*\*

\* 执行任务

\*

\* @return void

\*/

public function handle()

{

//

}

}

?>

如果该队列任务是一个携带 id 为 1 的 App\Video 实例，那么它将自动被标记上 App\Video:1 标签。因为 Horizon 会检查任务属性是否具有 Eloquent 模型，如果发现 Eloquent 模型，Horizon 将会智能的用该模型的类名和主键为任务标记上标签。

$video = App\Video::find(1);

App\Jobs\RenderVideo::dispatch($video);

(1)、自定义标签

如果你想手动的为队列执行的对象定义标签，你可以给这个类定义一个 tags 方法。

class RenderVideo implements ShouldQueue

{

/\*\*

\* 获取分配给这个任务的标签

\*

\* @return array

\*/

public function tags()

{

return ['render', 'video:'.$this->video->id];

}

}

--

6、通知

Note: 配置 Horizon 以发送 Slack 或 SMS 通知时，应查看 相关通知驱动程序的前提条件( https://learnku.com/docs/laravel/6.x/notifications )。

如果需要在队列等待时间过长时发起通知，可以在应用的 HorizonServiceProvider 中调用 Horizon::routeMailNotificationsTo, Horizon::routeSlackNotificationsTo , 和 Horizon::routeSmsNotificationsTo 方法。

Horizon::routeMailNotificationsTo('example@example.com');

Horizon::routeSlackNotificationsTo('slack-webhook-url', '#channel');

Horizon::routeSmsNotificationsTo('15556667777');

(1)、配置等待时间过长的阈值

你可以在 config/horizon.php 配置文件中设置等待时间过长的具体秒数。 waits 配置项可以针对每一个 链接 / 队列 配置阈值。

'waits' => [

'redis:default' => 60,

],

--

7、Metrics

Horizon 包含一个 Metrics 仪表盘，它可以提供任务和队列等待时间和吞吐量信息，为了填充此仪表盘，你需要通过应用的任务调度器配置 Horizon 的 snapshot Artisan 命令每五分钟运行一次。

/\*\*

\* 定义应用程序的任务调度

\*

\* @param \Illuminate\Console\Scheduling\Schedule $schedule

\* @return void

\*/

protected function schedule(Schedule $schedule)

{

$schedule->command('horizon:snapshot')->everyFiveMinutes();

}

--

--

五、Passport OAuth 认证 (Laravel Passport)

--

https://learnku.com/docs/laravel/6.x/passport/5152

--

--

1、简介

在 Laravel 中，实现基于传统表单的登陆和授权已经非常简单，但是如何满足 API 场景下的授权需求呢？在 API 场景里通常通过令牌来实现用户授权，而非维护请求之间的 Session 状态。在 Laravel 项目中使用 Passport 可以轻而易举地实现 API 授权认证，Passport 可以在几分钟之内为你的应用程序提供完整的 OAuth2 服务端实现。Passport 是基于由 Andy Millington 和 Simon Hamp 维护的 League OAuth2 server( https://github.com/thephpleague/oauth2-server ) 建立的。

注意：本文档假定你已熟悉 OAuth2 。如果你并不了解 OAuth2 ，阅读之前请先熟悉下 OAuth2 的 常用术语( https://oauth2.thephpleague.com/terminology/ ) 和特性。

--

--

2、安装

在开始之前，请通过 Composer 包管理器安装 Passport。

//

///1/ 安装Passport

composer require laravel/passport

//

///2/ 执行数据迁移

Passport 服务提供器使用框架注册自己的数据库迁移目录，因此在注册提供器后，就应该运行 Passport 的迁移命令来自动创建存储客户端和令牌的数据表。

php artisan migrate

//

///3/ 创建客户端 (个人访问客户端和密码授权客户端)

接下来，运行 passport:install 命令来创建生成安全访问令牌时所需的加密密钥，同时，这条命令也会创建用于生成访问令牌的「个人访问」客户端和「密码授权」客户端。

php artisan passport:install

//

///4/ 添加HasApiTokens Trait类

// 用于检查已认证用户令牌和使用范围。

上面命令执行后，请将 Laravel\Passport\HasApiTokens Trait 添加到 App\User 模型中，这个 Trait 会给你的模型提供一些辅助函数，用于检查已认证用户的令牌和使用范围。

<?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;

}

?>

//

///5/ 注册Passport路由

接下来，在 AuthServiceProvider 的 boot 方法中调用 Passport::routes 函数。这个函数会注册发出访问令牌并撤销访问令牌、客户端和个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;

class AuthServiceProvider extends ServiceProvider

{

/\*\*

\* 应用程序的策略映射。

\*

\* @var array

\*/

protected $policies = [

'App\Model' => 'App\Policies\ModelPolicy',

];

/\*\*

\* 注册认证 / 授权服务

\*

\* @return void

\*/

public function boot()

{

$this->registerPolicies();

Passport::routes();

}

}

?>

//

///6/ 改用passport授权看守器

最后，将配置文件 config/auth.php 中授权看守器 guards 的 api 的 driver 选项改为 passport。此调整会让你的应用程序在在验证传入的 API 的请求时使用 Passport 的 TokenGuard 来处理。

'guards' => [

'web' => [

'driver' => 'session',

'provider' => 'users',

],

'api' => [

'driver' => 'passport',

'provider' => 'users',

],

],

(0)、自定义迁移

如果你不打算使用 Passport 的默认迁移，你应该在 AppServiceProvider 的 register 方法中调用 Passport::ignoreMigrations 方法。 你可以用这个命令 php artisan vendor:publish --tag=passport-migrations 导出默认迁移。

默认情况下，Passport 使用字段 user\_id 标识用户。如果想使用不同的字段标识用户 (例如：UUIDs)，可以修改默认的 Passport 迁移文件。

(1)、前端快速上手

注意：为了使用 Passport 的 Vue 组件，你必须使用 Vue( https://vuejs.org/ ) JavaScript 框架。这些组件也使用了 Bootstrap CSS 框架。然而，如果你不打算使用这些工具，这些组件对于你自己的前端组件编写也十分有价值。

Passport 提供了一系列 JSON API ，你可以用它们来允许你的用户创建客户端和个人访问令牌。然而，编写与这些 API 交互的前端代码可能是很占用时间的。因此，Passport 也包括了预编译的 Vue( https://vuejs.org/ ) 组件，你可以直接使用或将其作为你自己的前端参考。

//

///1/ 使用Passport Vue组件

要使用 Passport 的 Vue 组件，使用 vendor:publish Artisan 命令。

php artisan vendor:publish --tag=passport-components

//

///2/ 注册Vue组件

被发布的组件将会被放到 resources/js/components 目录下。当组件被发布后，你应该在你的 resources/js/app.js 文件中注册它们。

Vue.component(

'passport-clients',

require('./components/passport/Clients.vue').default

);

Vue.component(

'passport-authorized-clients',

require('./components/passport/AuthorizedClients.vue').default

);

Vue.component(

'passport-personal-access-tokens',

require('./components/passport/PersonalAccessTokens.vue').default

);

注意：在 Laravel v5.7.19 之前，在注册组件时添加 .default 会导致控制台错误。有关此更改的解释，请参阅 Laravel Mix v4.0.0 发布说明( https://github.com/JeffreyWay/laravel-mix/releases/tag/v4.0.0 )。

//

///3/ 编译资源

npm run dev

//

///4/ 创建客户端和个人访问令牌

在注册了组件后，请确保运行 npm run dev 来重新编译你的资源。 当你重编译你的资源后，你可以将组件放到你应用的模板中以开始创建客户端和个人访问令牌。

<passport-clients></passport-clients>

<passport-authorized-clients></passport-authorized-clients>

<passport-personal-access-tokens></passport-personal-access-tokens>

(2)、部署 Passport

第一次在你的生产环境部署 Passport 时，你大概需要运行 passport:keys 命令。这个命令生成 Passport 生成访问令牌所需的密钥。生成的密钥一般情况下不应放在版本控制中。

//

///1/ 生成Passport密钥

// 生成 Passport 生成访问令牌所需的密钥。

php artisan passport:keys

生成 storage/oauth-private.key 和 storage/oauth-public.key。

//

///2/ 自定义 Passport 密钥加载路径

可以使用 Passport::loadKeysFrom 方法来自定义 Passport 密钥的加载路径。

/\*\*

\* 注册认证 / 授权服务

\*

\* @return void

\*/

public function boot()

{

$this->registerPolicies();

Passport::routes();

Passport::loadKeysFrom('/secret-keys/oauth');

}

//

///3/ 发布Passport配置文件

另外，您也可以使用 php artisan vendor:publish --tag=passport-config 发布 Passport 的配置文件，然后从您的环境变量提供加载加密密钥的选项。

PASSPORT\_PRIVATE\_KEY="-----BEGIN RSA PRIVATE KEY-----

<此处填写您的私钥>

-----END RSA PRIVATE KEY-----"

PASSPORT\_PUBLIC\_KEY="-----BEGIN PUBLIC KEY-----

<此处填写您的公钥>

-----END PUBLIC KEY-----"

--

--

3、配置

(1)、令牌的使用期限 (令牌的有效期)

默认情况下，Passport 发放的访问令牌是有一年有效期的。但是如果你想自定义访问令牌的有效期，可以使用 tokensExpireIn 、 refreshTokensExpireIn 和 personalAccessTokensExpireIn 方法。上述方法同样需要在 AuthServiceProvider 的 boot 方法中调用。

/\*\*

\* 注册认证 / 授权服务

\*

\* @return void

\*/

public function boot()

{

$this->registerPolicies();

Passport::routes();

Passport::tokensExpireIn(now()->addDays(15));

Passport::refreshTokensExpireIn(now()->addDays(30));

Passport::personalAccessTokensExpireIn(now()->addMonths(6));

}

(2)、覆盖默认的模型 (覆盖默认模型)

use App\Models\Passport\Client;

use App\Models\Passport\Token;

use App\Models\Passport\AuthCode;

use App\Models\Passport\PersonalAccessClient;

/\*\*

\* 注册认证 / 授权服务

\*

\* @return void

\*/

public function boot()

{

$this->registerPolicies();

Passport::routes();

Passport::useTokenModel(Token::class);

Passport::useClientModel(Client::class);

Passport::useAuthCodeModel(AuthCode::class);

Passport::usePersonalAccessClientModel(PersonalAccessClient::class);

}

--

--

4、发放访问令牌

熟悉 OAuth2 的开发者一定知道， OAuth2 中必不可少的部分就是授权码。当使用授权码时，客户端应用程序会将用户重定向到你的服务器，他们将批准或拒绝向客户端发出访问令牌的请求。

(1)、管理客户端

首先，构建需要与应用程序 API 交互的应用程序，开发人员将需要通过创建一个「客户端」来注册自己的应用程序。一般来说，这包括在用户批准其授权请求后，提供其应用程序的名称和应用程序可以重定向到的 URL。

//

///1/ passport:client 命令

创建客户端最简单的方式是使用 Artisan 命令 passport:client，你可以使用此命令创建自己的客户端，用于测试你的 OAuth2 的功能。在你执行 client 命令时，Passport 会提示你输入有关客户端的信息，最终会给你提供客户端的 ID 和 密钥。

php artisan passport:client

//

///2/ Redirect URLs

当有多个重定向 URL 白名单时，可以在 passport:client 命令提示输入 URL 时，使用逗号分隔来指定。

http://example.com/callback,http://examplefoo.com/callback

//

///3/ JSON API

考虑到你的用户无法使用 client 命令，Passport 为此提供了可用于创建「客户端」的 JSON API。这样你就不用再花时间编写控制器来创建、更新和删除客户端。

然而，你仍旧需要基于 Passport 的 JSON API 开发一套前端界面，为你的用户提供管理客户端的面板。下面我们会列出所有用于管理客户端的 API，为了方便起见，我们使用 Axios( https://github.com/mzabriskie/axios ) 来演示对端口发出 HTTP 请求。

这个 JSON API 由 web 和 auth 两个中间件保护，所以只能从应用程序中调用，不能从外部调用。

Tip：如果你不想自己实现整个客户端管理的前端界面，可以使用 前端快速上手( https://learnku.com/docs/laravel/6.x/passport/5152#frontend-quickstart ) 在几分钟内组建一套功能齐全的前端界面。

//

///3-1/ GET /oauth/clients

此路由会返回认证用户的所有客户端。主要用途是列出所有用户的客户端，以便他们可以编辑或删除它们。

axios.get('/oauth/clients')

.then(response => {

console.log(response.data);

});

//

///3-2/ POST /oauth/clients

此路由用于创建新客户端。它需要两个参数：客户端的名称 name 和授权后回调的 URL redirect。在批准或拒绝授权请求后，用户会被重定向到 redirect 参数提供的链接。

当客户端创建后，会返回客户端的 ID 和密钥。客户端可以使用这两个值从你的授权服务请求访问令牌 (Access token) 。该路由会返回新的客户端实例。

const data = {

name: 'Client Name',

redirect: 'http://example.com/callback'

};

axios.post('/oauth/clients', data)

.then(response => {

console.log(response.data);

})

.catch (response => {

// List errors on response...

});

//

///3-3/ PUT /oauth/clients/{client-id}

此路由用于更新客户端信息。它需要两个参数：客户端的名称 name 和授权后回调的 URL redirect。在批准或拒绝授权请求后，用户会被重定向 redirect 到这个链接。此路由会返回更新后的客户端实例。

const data = {

name: 'New Client Name',

redirect: 'http://example.com/callback'

};

axios.put('/oauth/clients/' + clientId, data)

.then(response => {

console.log(response.data);

})

.catch (response => {

// List errors on response...

});

//

///3-4/ DELETE /oauth/clients/{client-id}

此路由用于删除客户端 (client)。

axios.delete('/oauth/clients/' + clientId)

.then(response => {

//

});

(2)、请求令牌

//

///1/ 授权时的重定向

客户端创建之后，开发者会使用此客户端的 ID 和密钥来请求授权代码，并从应用程序访问令牌。首先，接入应用的用户向你应用程序的 /oauth/authorize 路由发出重定向请求，示例如下。

Route::get('/redirect', function (Request $request) {

$request->session()->put('state', $state = Str::random(40));

$query = http\_build\_query([

'client\_id' => 'client-id',

'redirect\_uri' => 'http://example.com/callback',

'response\_type' => 'code',

'scope' => '',

'state' => $state,

]);

return redirect('http://your-app.com/oauth/authorize?'.$query);

});

Tip：注意，路由 /oauth/authorize 已经在 Passport::routes 方法中定义。你不需要手动定义此路由。

//

///2/ 批准请求

接收到授权请求时，Passport 会自动向用户显示一个模版页面，允许用户批准或拒绝授权请求。如果用户批准请求，他们会被重定向回接入的应用程序指定的 redirect\_uri。redirect\_uri 必须和客户端创建时指定的 redirect 链接完全一致。

/// 授权页面

如果你想自定义授权确认页面，可以使用 Artisan 命令 vendor:publish 发布 Passport 的视图。发布后的视图文件存放在 resources/views/vendor/passport。

php artisan vendor:publish --tag=passport-views

/// 无需授权提示设置

有时您可能希望跳过授权提示，例如授权第一方客户端时。 您可以通过在客户端模型上定义 skipsAuthorization 方法来实现此目的。 如果 skipsAuthorization 返回 true，客户端将被批准，用户将被立即重定向回 redirect\_uri。

<?php

namespace App\Models\Passport;

use Laravel\Passport\Client as BaseClient;

class Client extends BaseClient

{

/\*\*

\* 确定客户端是否应跳过授权提示。

\*

\* @return bool

\*/

public function skipsAuthorization()

{

return $this->firstParty();

}

}

?>

//

///3/ 将授权码转换为访问令牌

如果用户批准授权请求，他们将被重定向回应用程序。 应首先根据重定向之前存储的值验证 state 参数。 如果 state 参数匹配，则应向您的应用程序发出 POST 请求以请求访问令牌。 该请求应包括用户批准授权请求时由应用程序颁发的授权代码。 在下面的例子中，我们将使用 Guzzle HTTP 库来实现这次 POST 请求。

Route::get('/callback', function (Request $request) {

$state = $request->session()->pull('state');

throw\_unless(

strlen($state) > 0 && $state === $request->state,

InvalidArgumentException::class

);

//// 安装GuzzleHttp：composer require guzzlehttp/guzzle；

// sudo vim /etc/host 添加 "127.0.0.1 project-laravel-laravelv6x.test"；

$http = new GuzzleHttp\Client;

$response = $http->post('http://your-app.com/oauth/token', [

'form\_params' => [

'grant\_type' => 'authorization\_code',

'client\_id' => 'client-id',

'client\_secret' => 'client-secret',

'redirect\_uri' => 'http://example.com/callback',

'code' => $request->code,

],

]);

return json\_decode((string) $response->getBody(), true);

});

路由 /oauth/token 返回的 JSON 响应中会包含 access\_token 、refresh\_token 和 expires\_in 属性。expires\_in 属性包含访问令牌的有效期（单位：秒）。

Tip：像 /oauth/authorize 路由一样，/oauth/token 路由在 Passport::routes 方法中定义了，你没必要手动去定义它。默认情况下，此路由使用 ThrottleRequests 中间件的设置进行限流。

(3)、刷新令牌

如果你的应用程序发放了短期的访问令牌，用户将需要通过在发出访问令牌时提供给他们的刷新令牌来刷新其访问令牌。在下面的例子中，我们使用 Guzzle HTTP 库来刷新令牌。

//// 安装GuzzleHttp：composer require guzzlehttp/guzzle；

// sudo vim /etc/host 添加 "127.0.0.1 project-laravel-laravelv6x.test"；

$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);

路由 /oauth/token 会返回一个 JSON 响应，其中包含 access\_token 、refresh\_token 和 expires\_in 属性。expires\_in 属性包含访问令牌的有效时间（单位：秒）。

--

--

5、密码授权令牌

OAuth2 密码授权机制可以让你自己的客户端（如移动应用程序）使用邮箱地址或者用户名和密码获取访问令牌。如此一来你就可以安全地向自己的客户端发出访问令牌，而不需要遍历整个 OAuth2 授权代码重定向流程。

(1)、创建密码授权客户端

在应用程序通过密码授权机制来发布令牌之前，在 passport:client 命令后加上 --password 参数来创建密码授权的客户端。如果你已经运行了 passport:install 命令，则不需要再运行此命令.

///1/ 创建密码授权客户端

php artisan passport:client --password

(2)、请求密码授权令牌 (请求令牌)

创建密码授权的客户端后，就可以使用用户的电子邮件地址和密码向 /oauth/token 路由发出 POST 请求来获取访问令牌。而该路由已经由 Passport::routes 方法注册，因此不需要手动定义它。如果请求成功，会在服务端返回的 JSON 响应中收到一个 access\_token 和 refresh\_token。

//// 安装GuzzleHttp：composer require guzzlehttp/guzzle；

// sudo vim /etc/host 添加 "127.0.0.1 project-laravel-laravelv6x.test"；

$http = new GuzzleHttp\Client;

$response = $http->post('http://your-app.com/oauth/token', [

'form\_params' => [

'grant\_type' => 'password',

'client\_id' => 'client-id',

'client\_secret' => 'client-secret',

'username' => 'taylor@laravel.com',

'password' => 'my-password',

'scope' => '',

],

]);

return json\_decode((string) $response->getBody(), true);

Tip：默认情况下，访问令牌是长期有效的。你可以根据需要 配置访问令牌的有效时间( https://learnku.com/docs/laravel/6.x/passport/5152#configuration )。

(3)、请求所有作用域

使用密码授权机制时，可以通过请求 scope 参数 \* 来授权应用程序支持的所有范围的令牌。如果你的请求中包含 scope 为 \* 的参数，令牌实例上的 can 方法会始终返回 true。这种作用域的授权只能分配给使用 password 或者 client\_credentials 授权时发出的令牌。

$response = $http->post('http://your-app.com/oauth/token', [

'form\_params' => [

'grant\_type' => 'password',

'client\_id' => 'client-id',

'client\_secret' => 'client-secret',

'username' => 'taylor@laravel.com',

'password' => 'my-password',

'scope' => '\*',

],

]);

(4)、自定义用户名字段

当你使用密码授权进行身份验证时，Passport 在模型里用 email 属性作为「username」。不过，你仍然可以通过在你的模型里面定义一个 findForPassport 方法来自定义验证行为。

<?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;

/\*\*

\* 通过给定的username获取用户实例。

\*

\* @param string $username

\* @return \App\User

\*/

public function findForPassport($username)

{

return $this->where('username', $username)->first();

}

}

?>

(5)、自定义密码验证

当你使用密码授权进行身份验证时，Passport 会在你的模型里使用 password 字段来验证给定的密码。如果你的模型里面没有 password 属性或者你期望能够自动密码验证逻辑，你可以在你的模型里面定义一个 validateForPassportPasswordGrant 方法。

<?php

namespace App;

use Laravel\Passport\HasApiTokens;

use Illuminate\Support\Facades\Hash;

use Illuminate\Notifications\Notifiable;

use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable

{

use HasApiTokens, Notifiable;

/\*\*

\* 通过Passport的密码授权验证用户使用的密码。

\*

\* @param string $password

\* @return bool

\*/

public function validateForPassportPasswordGrant($password)

{

return Hash::check($password, $this->password);

}

}

?>

--

--

6、隐式授权令牌

隐式授权类似于授权码授权；但是，令牌将在不交换授权码的情况下返回给客户端。这种授权最常用于无法安全存储客户端凭据的 javascript 或移动应用程序。通过调用 AuthServiceProvider 中的 enableImplicitGrant 方法来启用这种授权。

/\*\*

\* 注册任何身份验证/授权服务。

\*

\* @return void

\*/

public function boot()

{

$this->registerPolicies();

Passport::routes();

Passport::enableImplicitGrant();

}

启用上述授权后，开发者们可以使用他们的客户端 ID 从你的应用程序请求访问令牌。接入的应用程序应该向你的应用程序的 /oauth/authorize 路由发出重定向请求，如下所示。

Route::get('/redirect', function (Request $request) {

$request->session()->put('state', $state = Str::random(40));

$query = http\_build\_query([

'client\_id' => 'client-id',

'redirect\_uri' => 'http://example.com/callback',

'response\_type' => 'token',

'scope' => '',

'state' => $state,

]);

return redirect('http://your-app.com/oauth/authorize?'.$query);

});

{提示} 请记住，/oauth/authorize 路由已经在 Passport::routes 方法中定义好了，所以无需再次手动定义此路由。

--

--

7、客户端授权令牌 (客户端凭证授予令牌)

客户端凭证授予令牌适用于计算机到计算机的身份验证。例如，你可以在通过 API 执行维护任务的计划作业中使用此授权。

(1)、创建客户端凭证授权客户端

在应用程序可以通过客户端凭证授予令牌之前，需要先创建一个客户端凭证授权的客户端。你可以通过 passport:client 命令行的 --client 选项来完成此操作。

php artisan passport:client --client

(2)、注册中间件

接下来，要使用这种授权，你首先需要在 app/Http/Kernel.php 的 $routeMiddleware 属性中添加 CheckClientCredentials 中间件。

use Laravel\Passport\Http\Middleware\CheckClientCredentials;

protected $routeMiddleware = [

'client' => CheckClientCredentials::class,

];

(3)、路由附加中间件

之后，在路由上附加中间件。

Route::get('/orders', function (Request $request) {

...

})->middleware('client');

(4)、令牌作用域

要将对路由的访问限制在特定作用域内，可以在将 client 中间件附加到路由并提供以逗号分隔的所需作用域列表。

Route::get('/orders', function (Request $request) {

...

})->middleware('client:check-status,your-scope');

(5)、获取令牌

要获取此授权类型的令牌，请向 oauth/token 发出请求。

$guzzle = new GuzzleHttp\Client;

$response = $guzzle->post('http://your-app.com/oauth/token', [

'form\_params' => [

'grant\_type' => 'client\_credentials',

'client\_id' => 'client-id',

'client\_secret' => 'client-secret',

'scope' => 'your-scope',

],

]);

return json\_decode((string) $response->getBody(), true)['access\_token'];

--

--

8、个人访问令牌

有时候，你的用户要在不经过传统的授权码重定向流程的情况下向自己颁发访问令牌。允许用户通过应用程序用户界面对自己发出令牌，有助于用户体验你的 API，或者也可以将其作为一种更简单的发布访问令牌的方式。

(1)、创建个人访问令牌的客户端 (创建个人访问客户端)

在你的应用程序发布个人访问令牌之前，你需要在 passport:client 命令后带上 --personal 参数来创建对应的客户端。如果你已经运行了 passport:install 命令，则无需再运行此命令。

php artisan passport:client --personal

(2)、启用个人访问令牌

如果您已经定义了个人访问客户端，则可以调用 Passport 的 personalAccessClientId 方法来使用它。通常，应该从 AuthServiceProvider 的 boot 方法调用此方法。

/\*\*

\* 注册认证 / 授权服务

\*

\* @return void

\*/

public function boot()

{

$this->registerPolicies();

Passport::routes();

Passport::personalAccessClientId('client-id');

}

(2)、管理个人访问令牌

创建个人访问客户端后，你可以使用 User 模型实例上的 createToken 方法来为给定用户发布令牌。createToken 方法接受令牌的名称作为其第一个参数和可选的 作用域( https://learnku.com/docs/laravel/6.x/passport/5152#token-scopes ) 数组作为其第二个参数。

$user = App\User::find(1);

// 创建没有作用域的访问令牌...

$token = $user->createToken('Token Name')->accessToken;

// 创建有作用域的访问令牌...

$token = $user->createToken('My Token', ['place-orders'])->accessToken;

//

///1/ JSON API

Passport 中也有用来管理个人访问令牌的 JSON API，你可以将其与自己的前端配对，为用户提供管理个人访问令牌的仪表板。下面我们会介绍用于管理个人访问令牌的所有 API 接口。方便起见，我们使用 Axios( https://github.com/mzabriskie/axios ) 来演示对 API 的接口发出 HTTP 请求。

JSON API 由 web 和 auth 中间件保护；因此，只能从您自己的应用程序中调用它。无法从外部源调用它。

Tip：如果你不想实现自己的个人访问令牌管理的前端界面，可以根据 前端快速上手( https://learnku.com/docs/laravel/6.x/passport/5152#frontend-quickstart ) 在几分钟内组建功能齐全的前端界面。

//

//1-1/ GET /oauth/scopes

此路由会返回应用程序中定义的所有 作用域。你可以使用此路由列出用户可能分配给个人访问令牌的范围。

axios.get('/oauth/scopes')

.then(response => {

console.log(response.data);

});

//

//1-2/ GET /oauth/personal-access-tokens

此路由返回认证用户创建的所有个人访问令牌。这主要用于列出所有用户的令牌，以便他们可以编辑或删除它们。

axios.get('/oauth/personal-access-tokens')

.then(response => {

console.log(response.data);

});

//

//1-3/ POST /oauth/personal-access-tokens

此路由用于创建新的个人访问令牌。它需要两个数据：令牌的 name 和 scpoe。

const data = {

name: 'Token Name',

scopes: []

};

axios.post('/oauth/personal-access-tokens', data)

.then(response => {

console.log(response.data.accessToken);

})

.catch (response => {

// 列出响应中错误...

});

//

//1-4/ DELETE /oauth/personal-access-tokens/{token-id}

此路由可用于删除个人访问令牌。

axios.delete('/oauth/personal-access-tokens/' + tokenId);

--

--

9、路由保护

(1)、通过中间件

Passport 包含一个 验证保护机制( https://learnku.com/docs/laravel/6.0/authentication#adding-custom-guards ) 可以验证请求中传入的访问令牌。配置 api 的看守器使用 passport 驱动程序后，只需要在需要有效访问令牌的任何路由上指定 auth:api 中间件。

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

//

})->middleware('auth:api');

(2)、传递访问令牌

当调用 Passport 保护下的路由时，接入的 API 应用需要将访问令牌作为 Bearer 令牌放在请求头 Authorization 中。例如，使用 Guzzle HTTP 库时。

$response = $client->request('GET', '/api/user', [

'headers' => [

'Accept' => 'application/json',

'Authorization' => 'Bearer '.$accessToken,

],

]);

--

--

10、令牌作用域

作用域可以让 API 客户端在请求账户授权时请求特定的权限。例如，如果你正在构建电子商务应用程序，并不是所有接入的 API 应用都需要下订单的功能。你可以让接入的 API 应用只被允许授权访问订单发货状态。换句话说，作用域允许应用程序的用户限制第三方应用程序执行的操作。

(1)、定义作用域

你可以在 AuthServiceProvider 的 boot 方法中使用 Passport::tokensCan 方法来定义 API 的作用域。tokensCan 方法接受一个包含作用域名称和描述的数组作为参数。作用域描述将会在授权确认页中直接展示给用户，你可以将其定义为任何你需要的内容。

use Laravel\Passport\Passport;

Passport::tokensCan([

'place-orders' => 'Place orders',

'check-status' => 'Check order status',

]);

(2)、默认作用域

如果客户端没有请求任何特定的范围，你可以在 AuthServiceProvider 的 boot 方法中使用 Passport::setDefaultScope 方法来定义默认的作用域。

use Laravel\Passport\Passport;

Passport::setDefaultScope([

'check-status',

'place-orders',

]);

(3)、给令牌分配作用域

//

///1/ 请求授权码

使用授权码请求访问令牌时，接入的应用需为 scope 参数指定所需作用域。 scope 参数包含多个作用域时，名称之间使用空格分割。

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);

});

//

///2/ 分发个人访问令牌

使用 User 模型的 createToken 方法发放个人访问令牌时，可以将所需作用域的数组作为第二个参数传给此方法。

$token = $user->createToken('My Token', ['place-orders'])->accessToken;

(4)、检查作用域

Passport 包含两个中间件，可用于验证传入的请求是否包含访问指定作用域的令牌。 使用之前，需要将下面的中间件添加到 app/Http/Kernel.php 文件的 $routeMiddleware 属性中。

'scopes' => \Laravel\Passport\Http\Middleware\CheckScopes::class,

'scope' => \Laravel\Passport\Http\Middleware\CheckForAnyScope::class,

//

///1/ 检查所有作用域

路由可以使用 scopes 中间件来检查当前请求是否拥有指定的 所有 作用域。

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

// 访问令牌具有 "check-status" 和 "place-orders" 作用域...

})->middleware('scopes:check-status,place-orders');

//

///2/ 检查任意作用域

路由可以使用 scope 中间件来检查当前请求是否拥有指定的 任意 作用域。

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

// 访问令牌具有 "check-status" 或 "place-orders" 作用域...

})->middleware('scope:check-status,place-orders');

//

///3/ 检查令牌实例上的作用域

就算含有访问令牌验证的请求已经通过应用程序的验证，你仍然可以使用当前授权 User 实例上的 tokenCan 方法来验证令牌是否拥有指定的作用域。

use Illuminate\Http\Request;

Route::get('/orders', function (Request $request) {

if ($request->user()->tokenCan('place-orders')) {

//

}

});

//

///4/ 附加作用域方法

// scopeIds 方法将返回所有已定义 ID / 名称的数组。

Laravel\Passport\Passport::scopeIds();

// scopes 方法将返回一个包含所有已定义作用域数组的 Laravel\Passport\Scope 实例。

Laravel\Passport\Passport::scopes();

// scopesFor 方法将返回与给定 ID / 名称匹配的 Laravel\Passport\Scope 实例数组。

Laravel\Passport\Passport::scopesFor(['place-orders', 'check-status']);

// 你可以使用 hasScope 方法确定是否已定义给定作用域。

Laravel\Passport\Passport::hasScope('place-orders');

--

--

11、使用 JavaScript 接入 API

在构建 API 时， 如果能通过 JavaScript 应用接入自己的 API 将会给开发过程带来极大的便利。这种 API 开发方法允许你使用自己的应用程序的 API 和别人共享的 API 。你的 Web 应用程序、移动应用程序、第三方应用程序以及可能在各种软件包管理器上发布的任何 SDK 都可能会使用相同的 API。

通常，如果要在 JavaScript 应用程序中使用 API ，需要手动向应用程序发送访问令牌，并将其传递给应用程序。但是， Passport 有一个可以处理这个问题的中间件。将 CreateFreshApiToken 中间件添加到 app/Http/Kernel.php 文件中的 web 中间件组就可以了。

'web' => [

// Other middleware...

\Laravel\Passport\Http\Middleware\CreateFreshApiToken::class,

],

注意：你应该确保在您的中间件堆栈中 CreateFreshApiToken 中间件之前列出了 EncryptCookies 中间件。

这个 Passport 中间件将在你所有的对外请求中添加一个 laravel\_token cookie 。该 cookie 将包含一个加密后的 JWT ， Passport 将用来验证来自 JavaScript 应用程序的 API 请求。至此，您可以在不明确传递访问令牌的情况下向应用程序的 API 发出请求。

axios.get('/api/user')

.then(response => {

console.log(response.data);

});

//

///1/ 自定义 Cookie 名称

如果需要，你可以在 AuthServiceProvider 的 boot 方法中使用 Passport::cookie 方法来自定义 laravel\_token cookie 的名称。

/\*\*

\* 注册认证 / 授权服务

\*

\* @return void

\*/

public function boot()

{

$this->registerPolicies();

Passport::routes();

Passport::cookie('custom\_name');

}

//

///2/ CSRF 保护

当使用这种授权方法时，您需要确认请求中包含有效的 CSRF 令牌。默认的 Laravel JavaScript 脚手架会包含一个 Axios 实例，该实例是自动使用加密的 XSRF-TOKEN cookie 值在同源请求上发送 X-XSRF-TOKEN 请求头。

注意：如果您选择发送 X-CSRF-TOKEN 请求头而不是 X-XSRF-TOKEN ，则需要使用 csrf\_token() 提供的未加密令牌。

--

--

12、事件

Passport 在发出访问令牌和刷新令牌时引发事件。 您可以使用这些事件来修改或撤消数据库中的其他访问令牌。 您可以在应用程序的 EventServiceProvider 中将监听器附加到这些事件。

/\*\*

\* 应用程序事件监听映射

\*

\* @var array

\*/

protected $listen = [

'Laravel\Passport\Events\AccessTokenCreated' => [

'App\Listeners\RevokeOldTokens',

],

'Laravel\Passport\Events\RefreshTokenCreated' => [

'App\Listeners\PruneOldTokens',

],

];

--

--

13、测试

Passport 的 actingAs 方法可以指定当前已认证用户及其作用域。actingAs 方法的第一个参数是用户实例，第二个参数是用户令牌作用域数组。

use App\User;

use Laravel\Passport\Passport;

public function testServerCreation()

{

Passport::actingAs(

factory(User::class)->create(),

['create-servers']

);

$response = $this->post('/api/create-server');

$response->assertStatus(201);

}

?>

--

--

六、Scout 全文搜索 (Laravel Scout)

--

https://learnku.com/docs/laravel/6.x/scout/5191

--

1、简介

Laravel Scout 为 Eloquent ( https://learnku.com/docs/laravel/6.x/eloquent ) 模型的全文搜索提供了基于驱动的简单的解决方案。通过使用模型观察者， Scout 会自动同步 Eloquent 记录的搜索索引。

目前， Scout 自带一个 Algolia ( https://www.algolia.com/ ) 驱动。不过，编写自定义驱动也很简单，你可以轻松的通过自己的搜索实现来扩展 Scout。

--

2、安装

首先，通过 Composer 包管理器来安装 Scout。

///1/ 安装Scout

composer require laravel/scout

//

Scout 安装完成后，使用 vendor:publish Artisan 命令来生成 Scout 配置文件。这个命令将在你的 config 目录下生成一个 scout.php 配置文件。

///2/ 发布Scout

php artisan vendor:publish --provider="Laravel\Scout\ScoutServiceProvider"

//

///3/ 添加Trait

最后，在你要做搜索的模型中添加 Laravel\Scout\Searchable trait。这个 trait 会注册一个模型观察者来保持模型和所有驱动的同步。

<?php

namespace App;

use Laravel\Scout\Searchable;

use Illuminate\Database\Eloquent\Model;

class Post extends Model

{

use Searchable;

}

?>

(1)、队列

虽然并不强制使用 Scout，但是在使用这个库之前，强烈建议你配置一个 队列驱动，使用它运行一个队列来处理允许 Scout 将模型信息同步到搜索索引的所有操作，为你的应用的 web 接口提供更快的响应。

一旦你配置了队列驱动程序，你的 config/scout.php 配置文件中 queue 选项的值要设置为 true。

'queue' => true,

(2)、驱动必要条件

//

///1/ Algolia

使用 Algolia 驱动时，需要在 config/scout.php 配置文件配置你的 Algolia id 和 secret 凭证。配置好凭证之后，还需要使用 Composer 包管理器安装 Algolia PHP SDK。

composer require algolia/algoliasearch-client-php:^2.2

--

3、配置

(1)、配置模型索引

每个 Eloquent 模型都是通过给定的 「索引」 进行同步，该 「索引」 包含所有可搜索的模型记录。换句话说，你可以把每一个 「索引」 设想为一张 MySQL 数据表。默认情况下，每个模型都会被持久化到与模型的 「表」 名（通常是模型名称的复数形式）相匹配的索引。你也可以通过重写模型上的 searchableAs 方法来自定义模型的索引。

<?php

namespace App;

use Laravel\Scout\Searchable;

use Illuminate\Database\Eloquent\Model;

class Post extends Model

{

use Searchable;

/\*\*

\* 获取索引名称

\*

\* @return string

\*/

public function searchableAs()

{

return 'posts\_index';

}

}

?>

(2)、配置可搜索数据

默认情况下，模型以完整的 toArray 格式持久化到搜索索引。如果要自定义同步到搜索索引的数据，可以覆盖模型上的 toSearchableArray 方法。

<?php

namespace App;

use Laravel\Scout\Searchable;

use Illuminate\Database\Eloquent\Model;

class Post extends Model

{

use Searchable;

/\*\*

\* 获取模型的可搜索数据

\*

\* @return array

\*/

public function toSearchableArray()

{

$array = $this->toArray();

// 自定义数组...

return $array;

}

}

?>

(3)、配置模型 ID

默认情况下，Scout 将使用模型的主键作为搜索索引中存储的唯一 ID 。 可以通过模型上的 getScoutKey 方法自定义。

<?php

namespace App;

use Laravel\Scout\Searchable;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

use Searchable;

/\*\*

\* 获取模型主键

\*

\* @return mixed

\*/

public function getScoutKey()

{

return $this->email;

}

}

?>

--

4、索引

(1)、批量导入

如果你想安装 Scout 到已存在的项目中，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 用来导入所有已存在的记录到搜索索引。

//

///1/ 批量导入数据

php artisan scout:import "App\Post"

//

///2/ 删除所有数据

flush 命令可用于从搜索索引中删除所有模型的记录。

php artisan scout:flush "App\Post"

(2)、添加记录

当你将 Laravel\Scout\Searchable trait 添加到模型中，你需要做的就是 save 一个模型实例，它将自动被添加到搜索索引。如果你已经将 Scout 配置为 使用队列( https://learnku.com/docs/laravel/5.8/scout/3946#queueing )，那这个操作会在后台由你的队列工作进程来执行。

$order = new App\Order;

// ...数据赋值

$order->save();

//

///1/ 通过查询添加

如果你想通过 Eloquent 查询构造器将模型集合添加到搜索索引中，你也可以在 Eloquent 查询构造器上链式调用 searchable 方法。searchable 会把构造器的查询 结果分块 ( https://learnku.com/docs/laravel/5.8/eloquent#chunking-results ) 并且将记录添加到你的搜索索引里。同样的，如果你已经配置 Scout 为使用队列，则所有的数据块将在后台由你的队列工作进程添加。

// 通过 Eloquent 查询构造器增加...

App\Order::where('price', '>', 100)->searchable();

// 你也可以通过模型关系增加记录...

$user->orders()->searchable();

// 你也可以通过集合增加记录...

$orders->searchable();

searchable 方法可以被看做是「更新插入」的操作。换句话说，如果模型记录已经在你的索引里了，它就会被更新。如果搜索索引中不存在，则将其添加到索引中。

(3)、更新记录

要更新可搜索的模型，只需要更新模型实例的属性并将模型 save 到数据库。Scout 会自动将更新同步到你的搜索索引中。

$order = App\Order::find(1);

// 更新订单...

$order->save();

你也可以在 Eloquent 查询语句上使用 searchable 方法来更新一个模型的集合。如果这个模型不存在你检索的索引里，就会被创建。

// 通过 Eloquent 查询更新...

App\Order::where('price', '>', 100)->searchable();

// 你也可以通过数据间的关联进行更新...

$user->orders()->searchable();

// 你也可以通过数据集合进行更新...

$orders->searchable();

(4)、删除记录

使用 delete 从数据库中删除该模型就可以移除索引里的记录。这种删除形式甚至与 软删除 的模型兼容。

$order = App\Order::find(1);

$order->delete();

如果你不希望记录在删除之前被检索到，可以在 Eloquent 查询实例或集合上使用 unsearchable 方法。

// 通过 Eloquent 查询删除...

App\Order::where('price', '>', 100)->unsearchable();

// 你可以通过数据间的关系进行删除...

$user->orders()->unsearchable();

// 你可以通过数据集合进行删除...

$orders->unsearchable();

(5)、暂停索引

你可能需要在执行一批 Eloquent 操作的时候，不同步模型数据到搜索索引。此时你可以使用 withoutSyncingToSearch 方法来执行此操作。这个方法接受一个立即执行的回调。该回调中所有的操作都不会同步到模型的索引。

App\Order::withoutSyncingToSearch(function () {

// 执行模型操作...

});

(6)、有条件的可搜索模型实例

有时候你可能需要在某些条件下模型是可搜索的。例如，假设你有 App\Post 模型可能两种状态之一：「草稿」和「发布」。你可能只允许搜索 「发布」过的帖子。为了实现这一点，你需要在模型中定义一个 shouldBeSearchable 方法。

public function shouldBeSearchable()

{

return $this->isPublished();

}

只有在通过 save 方法、查询或关联模型操作时，才应使用 shouldBeSearchable 方法。直接使用 searchable 方法将使模型或集合的可搜索结果覆盖 shouldBeSearchable 方法的结果。

// 此处将遵循 "shouldBeSearchable" 结果...

App\Order::where('price', '>', 100)->searchable();

$user->orders()->searchable();

$order->save();

// 此处将覆盖 "shouldBeSearchable" 结果...

$orders->searchable();

$order->searchable();

--

5、搜索

你可以使用 search 方法来搜索模型。search 方法接受一个用于搜索模型的字符串。你还需要在搜索查询上链式调用 get 方法，才能用给定的搜索语句查询与之匹配的 Eloquent 模型。

$orders = App\Order::search('Star Trek')->get();

Scout 搜索返回 Eloquent 模型的集合，因此你可以直接从路由或控制器返回结果，它们会被自动转换成 JSON 格式。

use Illuminate\Http\Request;

Route::get('/search', function (Request $request) {

return App\Order::search($request->search)->get();

});

如果你想在它们返回 Eloquent 模型前得到原结果，你应该使用 raw 方法。

$orders = App\Order::search('Star Trek')->raw();

搜索查询通常会在模型的 searchableAs 方法指定的索引上执行。当然，你也可以使用 within 方法指定应该搜索的自定义索引。

$orders = App\Order::search('Star Trek')

->within('tv\_shows\_popularity\_desc')

->get();

(1)、Where 条件 (Where 语句)

允许你在搜索查询中增加简单的 "where" 语句。目前，这些语句只支持基本的数值等式检查，并且主要是用于根据租户 ID 进行的范围搜索查询。由于搜索索引不是关系型数据库，因此当前不支持更高级的 "where" 语句。

$orders = App\Order::search('Star Trek')->where('user\_id', 1)->get();

(2)、分页

除了检索模型的集合，你也可以使用 paginate 方法对搜索结果进行分页。这个方法会返回一个就像 传统的 Eloquent 查询分页 ( https://learnku.com/docs/laravel/5.8/pagination ) 一样的 Paginator 实例。

$orders = App\Order::search('Star Trek')->paginate();

你可以通过将数量作为第一个参数传递给 paginate 方法来指定每页检索多少个模型。

$orders = App\Order::search('Star Trek')->paginate(15);

获取到检索结果后，就可以使用 Blade ( https://learnku.com/docs/laravel/5.8/blade ) 来渲染分页链接和显示检索结果，就像传统的 Eloquent 查询分页一样。

<div class="container">

@foreach ($orders as $order)

{{ $order->price }}

@endforeach

</div>

{{ $orders->links() }}

(3)、软删除

如果你的索引模型是 软删除 ( https://learnku.com/docs/laravel/5.8/eloquent#soft-deleting )，并且你需要搜索软删除的模型，设置 config/scout.php 配置文件的 soft\_delete 选项的值为 true。

'soft\_delete' => true,

当这个配置选项是 true 的时候， Scout 不会从搜索索引中移除软删除模型。相反，它会在索引记录中设置一个隐藏 \_\_soft\_deleted 属性。然后，在搜索的时候，你可以使用 withTrashed 或 onlyTrashed 方法检索软删除记录。

// 搜索结果包括已删除的记录...

$orders = App\Order::withTrashed()->search('Star Trek')->get();

// 搜索结果只含已删除的记录...

$orders = App\Order::onlyTrashed()->search('Star Trek')->get();

{提示} 要永久删除模型可以使用 forceDelete 来删除，Scout 将自动的从搜索索引中移除模型。

(4)、自定义引擎的搜索 (自定义搜索引擎)

如果需要自定义引擎的搜索行为，可以将回调作为第二个参数传递给 search 方法。例如，在将搜索查询传递给 Algolia 之前，可以使用这个回调将地理位置数据添加到搜索请求中。

use Algolia\AlgoliaSearch\SearchIndex;

App\Order::search('Star Trek', function (SearchIndex $algolia, string $query, array $options) {

$options['body']['query']['bool']['filter']['geo\_distance'] = [

'distance' => '1000km',

'location' => ['lat' => 36, 'lon' => 111],

];

return $algolia->search($query, $options);

})->get();

--

6、自定义引擎

(1)、写引擎

如果内置的 Scout 搜索引擎不能满足你的需求，你可以编写自定义的引擎并且将它注册到 Scout。你的引擎需要继承 Laravel\Scout\Engines\Engine 抽象类，这个抽象类包含了你自定义的引擎必须要实现的 8 个方法。

use Laravel\Scout\Builder;

abstract public function update($models);

abstract public function delete($models);

abstract public function search(Builder $builder);

abstract public function paginate(Builder $builder, $perPage, $page);

abstract public function mapIds($results);

abstract public function map($results, $model);

abstract public function getTotalCount($results);

abstract public function flush($model);

在 Laravel\Scout\Engines\AlgoliaEngine 类里查看这些方法的实现会对你有较大的帮助。这个类会为你在学习如何在自定义引擎中实现这些方法提供一个好的起点。

(2)、注册引擎

一旦你写好了自定义引擎，你可以用 Scout 引擎管理的 extend 方法将它注册到 Scout。你只需要从 AppServiceProvider 下的 boot 方法或者应用中使用的任何一个服务提供器中调用 extend 方法。举个例子，如果你写好了一个 MySqlSearchEngine，你可以像这样去注册它。

use Laravel\Scout\EngineManager;

/\*\*

\* 启动任何的服务

\*

\* @return void

\*/

public function boot()

{

resolve(EngineManager::class)->extend('mysql', function () {

return new MySqlSearchEngine;

});

}

引擎注册后，你可以在 config/scout.php 配置文件中指定它为默认的 Scout driver。

'driver' => 'mysql',

--

7、生成器宏 (生成宏命令)

如果你想要自定义生成器方法，你可以使用 Laravel\Scout\Builder 类下的 macro 方法。通常，定义 "macros" 时， 需要实现 service provider's boot 方法。

<?php

namespace App\Providers;

use Laravel\Scout\Builder;

use Illuminate\Support\ServiceProvider;

use Illuminate\Support\Facades\Response;

class ScoutMacroServiceProvider extends ServiceProvider

{

/\*\*

\* 注册应用的Scout 宏命令.

\*

\* @return void

\*/

public function boot()

{

Builder::macro('count', function () {

return $this->engine->getTotalCount(

$this->engine()->search($this)

);

});

}

}

?>

macro 函数接受一个名字作为第一个参数，第二个参数为一个闭包函数。当调用 Laravel\Scout\Builder 宏命令时，调用这个函数。

App\Order::search('Star Trek')->count();

--

--

七、Socialite 社会化登录 (Laravel 社会化登录)

--

https://learnku.com/docs/laravel/5.8/socialite/3947

--

1、简介

除了典型的基于表单的身份验证之外，Laravel 还提供了一种使用 Laravel Socialite ( https://github.com/laravel/socialite ) 对 OAuth providers 进行身份验证的简单方便的方法。 Socialite 目前支持 Facebook，Twitter，LinkedIn，Google，GitHub，GitLab 和 Bitbucket 的身份验证。

其他平台的驱动器可以在 Socialite Providers ( https://socialiteproviders.github.io/ ) 社区驱动网站查找。

--

2、升级社会化登录

升级到 Socialite 的新主要版本时，请务必仔细查看 升级指南( https://github.com/laravel/socialite/blob/master/UPGRADE.md )。

--

3、安装

在开始使用社会化登录功能之前，通过 Composer 将 laravel/socialite 包添加到你的项目依赖里面。

composer require laravel/socialite

--

4、配置

在使用 Socialite 之前，您还需要为应用程序使用的 OAuth 服务添加凭据。 这些凭证应该放在你的 config / services.php 配置文件中，并且应该使用密钥 facebook，twitter，linkedin，google，github，gitlab 或 bitbucket， 取决于您的应用程序所需的提供商。例如。

'github' => [

'client\_id' => env('GITHUB\_CLIENT\_ID'),

'client\_secret' => env('GITHUB\_CLIENT\_SECRET'),

'redirect' => 'http://your-callback-url',

],

如果 redirect 项的值是个相对路径，它会自动解析为全称 URL。

--

5、路由

接下来，就要对用户认证了！这需要两个路由：一个路由用于把用户重定向到 OAuth 提供器，另一个则用于在认证完成后接收相应提供器的回调请求。可以通过 facade 方式 Socialite:: 来访问 Socialite。

<?php

namespace App\Http\Controllers\Auth;

use Socialite;

class LoginController extends Controller

{

/\*\*

\* Redirect the user to the GitHub authentication page.

\*

\* @return \Illuminate\Http\Response

\*/

public function redirectToProvider()

{

// redirect 方法负责将用户发送到 OAuth 提供程序。

return Socialite::driver('github')->redirect();

}

/\*\*

\* Obtain the user information from GitHub.

\*

\* @return \Illuminate\Http\Response

\*/

public function handleProviderCallback()

{

// user 方法将读取传入请求并从提供程序检索用户的信息。

$user = Socialite::driver('github')->user();

// $user->token;

}

}

?>

redirect 方法负责将用户发送到 OAuth 提供程序，而 user 方法将读取传入请求并从提供程序检索用户的信息。

当然，还需要在你的控制器方法中定义好路由规则。

Route::get('login/github', 'Auth\LoginController@redirectToProvider');

Route::get('login/github/callback', 'Auth\LoginController@handleProviderCallback');

--

6、可选参数

许多 OAuth 提供程序支持重定向请求中的可选参数。 要在请求中包含任何可选参数，请使用关联数组调用 with 方法。

return Socialite::driver('google')

->with(['hd' => 'example.com'])

->redirect();

使用 with 方法时，注意不要传递任何保留的关键字，如 state 或 response\_type。

--

7、访问范围 (访问作用域)

在重定向用户之前，您还可以使用 scopes 方法在请求中添加其他「作用域」。此方法将所有现有范围与您提供的范围合并。

return Socialite::driver('github')

->scopes(['read:user', 'public\_repo'])

->redirect();

您可以使用 setScopes 方法覆盖所有现有范围。

return Socialite::driver('github')

->setScopes(['read:user', 'public\_repo'])

->redirect();

--

8、无认证状态

stateless 方法可用于禁用会话状态验证。 在向 API 添加社交身份验证时，这非常有用。

return Socialite::driver('google')->stateless()->user();

--

9、检索用户详细信息 (获取用户实例)

有了用户实例之后，就可以获取更多用户详情。

$user = Socialite::driver('github')->user();

//

// OAuth2 Providers

$token = $user->token;

$refreshToken = $user->refreshToken; // not always provided

$expiresIn = $user->expiresIn;

//

// OAuth1 Providers

$token = $user->token;

$tokenSecret = $user->tokenSecret;

//

// 获取所有 Providers

$user->getId();

$user->getNickname();

$user->getName();

$user->getEmail();

$user->getAvatar();

(1)、从令牌中检索用户详细信息 (OAuth2)

如果你已经有了一个用户的有效访问令牌，你可以使用 userFromToken 方法检索用户的详细信息。

$user = Socialite::driver('github')->userFromToken($token);

(2)、从令牌和秘钥中检索用户详细信息 (OAuth1)

如果你已经有了一个有效的用户令牌 / 秘钥，你可以使用 userFromTokenAndSecret 方法检索他们的详细信息。

$user = Socialite::driver('twitter')->userFromTokenAndSecret($token, $secret);

--

--

八、Telescope 应用调试工具 (Laravel Telescope)

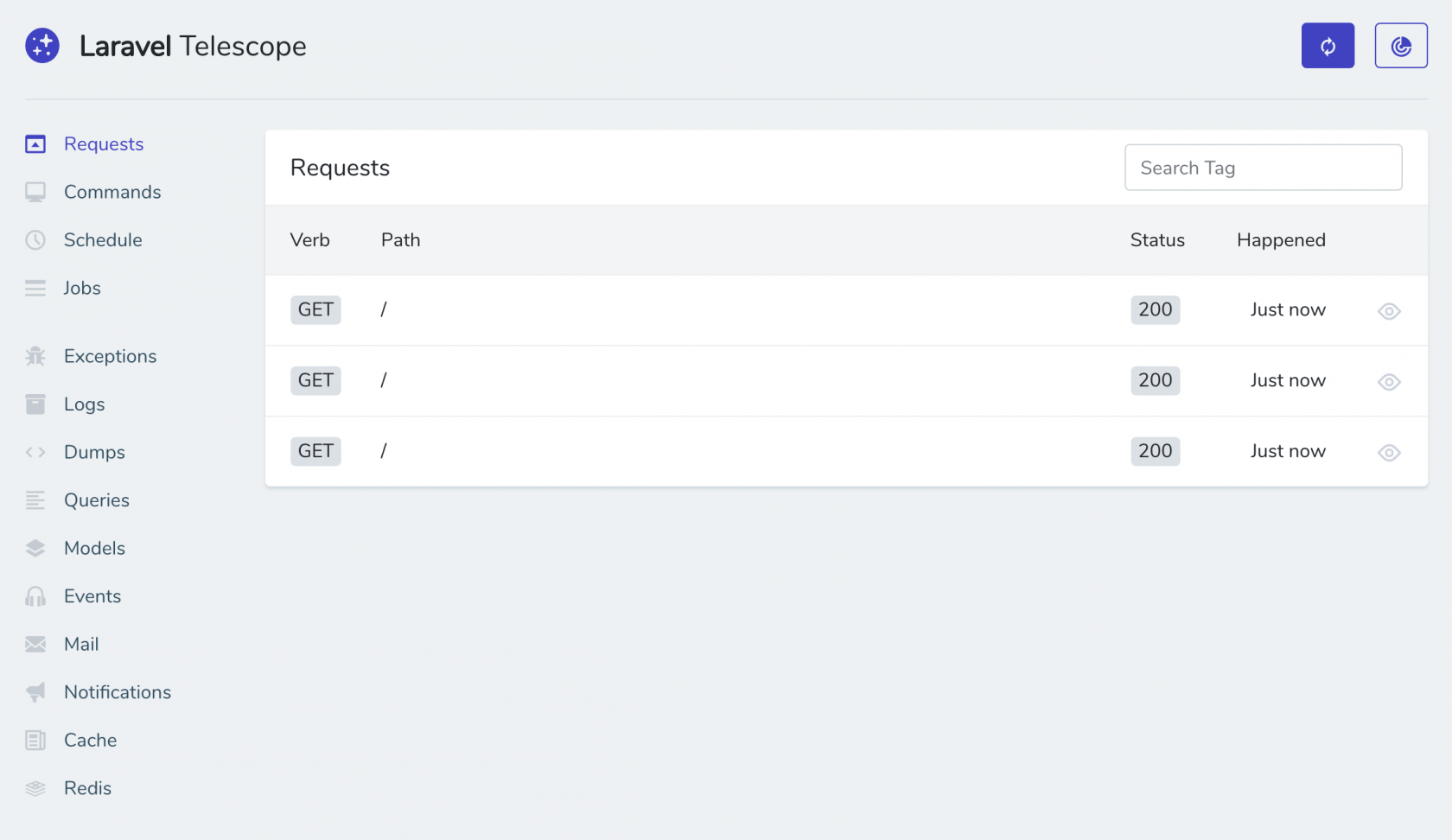
--

https://learnku.com/docs/laravel/5.8/telescope/3948

--

1、简介

Larave Telescope 是 Laravel 框架的优雅调试助手。Telescope 可深入了解进入应用程序的请求、异常、日志条目、数据库查询、排队作业、邮件、通知、缓存操作、计划任务、变量转储等。Telescope 是您本地 Laravel 开发环境的绝佳伴侣。



( 图 https://cdn.learnku.com/uploads/images/201904/22/29333/Brtb1DNyg7.png!large )

--

2、安装

你可以使用 Composer 在 Laravel 项目中安装 Telescope 扩展。

///1/ Telescope依赖

composer require laravel/telescope

//

///2/

安装 Telescope 后，可以在 Artisan 使用 telescope:install 命令来配置扩展实例。安装 Telescope 后，还应运行 migrate 命令。

// Telescope安装

php artisan telescope:install

php artisan migrate

(1)、更新 Telescope

更新 Telescope 时，您应该重新配置加载 Telescope 实例。

///1/ Telescope更新

php artisan telescope:publish

(2)、仅在特定环境中安装

如果您打算仅使用 Telescope 来协助您的本地开发。可以使用 --dev 标志安装 Telescope。

// 开发环境安装

composer require laravel/telescope --dev

运行 telescope:install 后，您应该从 app 配置文件中删除 TelescopeServiceProvider 服务提供注册。相反，在 AppServiceProvider 的 register 方法中手动注册服务。

use Laravel\Telescope\TelescopeServiceProvider;

/\*\*

\* 注册应用服务。

\*

\* @return void

\*/

public function register()

{

if ($this->app->isLocal()) {

$this->app->register(TelescopeServiceProvider::class);

}

}

(3)、定制数据迁移 (数据迁移)

如果您不打算使用 Telescope 的默认迁移，则应该在 AppServiceProvider 的 register 方法中调用 Telescope::ignoreMigrations 方法。您可以使用 php artisan vendor:publish --tag=telescope-migrations 命令导出默认迁移。

(4)、配置

使用 Telescope，其主要配置文件将位于 config/telescope.php。此配置文件允许您配置监听程序选项，每个配置选项都包含其用途说明，因此请务必彻底浏览此文件。

如果需要，您可以使用 enabled 配置选项完全禁用 Telescope 的数据收集。

'enabled' => env('TELESCOPE\_ENABLED', true),

(5)、数据修改

有了数据修改，telescope\_entries 表可以非常快速地累积记录。为了缓解这个问题，你应该使用 Artisan 每天运行 telescope:prune 命令。

$schedule->command('telescope:prune')->daily();

默认情况下，会清除超过 24 小时的所有数据。在调用命令时可以使用 hours 选项来确定保留 Telescope 数据的时间。例如，以下命令将删除 48 小时前创建的所有记录。

$schedule->command('telescope:prune --hours=48')->daily();

--

3、仪表板授权

Telescope 在 /telescope 处显示仪表板。默认情况下，您只能在 本地 环境中访问此仪表板。在你的 app/Providers/TelescopeServiceProvider.php 文件中，有一个 gate 方法。此授权能控制在 非本地 环境中对 Telescope 的访问。您可以根据需要随意修改此权限限制以对 Telescope 安装和访问。

/\*\*

\* 註冊 Telescope Gate。

\*

\* 使用 Gate 决定谁可以在非本地环境中访问 Telescope。

\*

\* @return void

\*/

protected function gate()

{

Gate::define('viewTelescope', function ($user) {

return in\_array($user->email, [

'taylor@laravel.com',

]);

});

}

--

4、过滤

(1)、单项过滤

您可以通过在 TelescopeServiceProvider 中注册的 filter 回调来过滤 Telescope 记录的数据。默认情况下，此回调会记录 本地 环境中的所有数据以及所有其他环境中的异常、进程中断、计划任务和带有受监控标记的数据。

/\*\*

\* 注册应用服务。

\*

\* @return void

\*/

public function register()

{

$this->hideSensitiveRequestDetails();

Telescope::filter(function (IncomingEntry $entry) {

if ($this->app->isLocal()) {

return true;

}

return $entry->isReportableException() ||

$entry->isFailedJob() ||

$entry->isScheduledTask() ||

$entry->hasMonitoredTag();

});

}

(2)、批量过滤

虽然 filter 回调过滤单个条目的数据，但您可以使用 filterBatch 方法注册一个回调，该回调过滤给定请求或控制台命令的所有数据。如果回调返回 true ，则所有数据都由 Telescope 记录。

use Illuminate\Support\Collection;

/\*\*

\* 注册应用服务。

\*

\* @return void

\*/

public function register()

{

$this->hideSensitiveRequestDetails();

Telescope::filterBatch(function (Collection $entries) {

if ($this->app->isLocal()) {

return true;

}

return $entries->contains(function ($entry) {

return $entry->isReportableException() ||

$entry->isFailedJob() ||

$entry->isScheduledTask() ||

$entry->hasMonitoredTag();

});

});

}

--

5、可用的监听

当在控制台执行命令或处理请求时，Telescope 监听器会收集应用程序数据。您可以在 config/telescope.php 配置文件中自定义要启用监听项的列表。

'watchers' => [

Watchers\CacheWatcher::class => true,

Watchers\CommandWatcher::class => true,

...

],

一些监听器还允许您提供其他自定义选项。

'watchers' => [

Watchers\QueryWatcher::class => [

'enabled' => env('TELESCOPE\_QUERY\_WATCHER', true),

'slow' => 100,

],

...

],

(1)、缓存监听

当缓存键被命中、遗漏、更新和遗忘时，缓存监听器会记录数据。

(2)、命令监听

只要执行 Artisan 命令，命令监听器就会记录参数、选项、退出代码和输出。如果您想排除监听器记录的某些命令，您可以在 config/telescope.php 文件的 ignore 选项中指定命令：

'watchers' => [

Watchers\CommandWatcher::class => [

'enabled' => env('TELESCOPE\_COMMAND\_WATCHER', true),

'ignore' => ['key:generate'],

],

...

],

(3)、储存监听 (数据监听)

数据监听器在 Telescope 中记录并显示您的数据变量。使用 Laravel 时，可以使用全局 dump 函数输出变量。必须在浏览器中打开数据监听器选项卡，才能进行输出变量，否则监听器将忽略此次输出。

(4)、事件监听

事件监听器记录应用程序调度的任何事件的有效负载、监听器和广播数据。事件监听器忽略了 Laravel 框架的内部事件。

(5)、异常监听

异常监听器记录应用程序引发的任何可报告异常的数据和堆栈跟踪。

(6)、Gate 监听

Gate 监听器记录您的应用程序的 Gate 和策略检查的数据和结果。如果您希望将某些能力排除在监听器的记录之外，您可以在 config/telescope.php 文件的 ignore\_abilities 选项中指定它们。

'watchers' => [

Watchers\GateWatcher::class => [

'enabled' => env('TELESCOPE\_GATE\_WATCHER', true),

'ignore\_abilities' => ['viewNova'],

],

...

],

(7)、进程监听

进程监听器记录应用程序分派的任何作业的数据和状态。

(8)、日志监听

日志监视器记录应用程序写入的任何日志的日志数据。

(9)、邮件监听

邮件监视器允许您查看电子邮件的浏览器内预览及其相关数据。您也可以将该电子邮件下载为 .eml 文件。

(10)、模型监听

只要调度了模型的 create、updated、restored 或 deleted 事件，模型观察器就会记录模型更改。您可以通过监听器的 events 选项指定应记录哪些模型事件。

'watchers' => [

Watchers\ModelWatcher::class => [

'enabled' => env('TELESCOPE\_MODEL\_WATCHER', true),

'events' => ['eloquent.created\*', 'eloquent.updated\*'],

],

...

],

(11)、消息通知监听

消息通知监听器记录您的应用程序发送的所有通知。如果通知触发了电子邮件并且您启用了邮件监听器，则电子邮件也可以在邮件监视器屏幕上进行预览。

(12)、数据查询监听

数据查询监听器记录应用程序执行的所有查询的原始 SQL、绑定和执行时间。观察者还将任何慢于 100 毫秒的查询标记为 slow。您可以使用观察者的 slow 选项自定义慢查询阈值。

'watchers' => [

Watchers\QueryWatcher::class => [

'enabled' => env('TELESCOPE\_QUERY\_WATCHER', true),

'slow' => 50,

],

...

],

(13)、Redis 监听

必须启用 Redis 事件才能使 Redis 监听器正常运行。您可以通过在 app/Providers/AppServiceProvider.php 文件的 boot 方法中调用 Redis::enableEvents() 来启用 Redis 事件。

Redis 监听器记录您的应用程序执行的所有 Redis 命令。如果您使用 Redis 进行缓存，Redis 监听器也会记录缓存命令。

(14)、请求监听

请求监听器记录与应用程序处理的任何请求相关联的请求、标头、会话和响应数据。您可以通过 size\_limit （以 KB 为单位）选项限制响应数据。

'watchers' => [

Watchers\RequestWatcher::class => [

'enabled' => env('TELESCOPE\_REQUEST\_WATCHER', true),

'size\_limit' => env('TELESCOPE\_RESPONSE\_SIZE\_LIMIT', 64),

],

...

],

(15)、时间表监听 (Schedule 监听)

Schedule 监听器记录应用程序运行的任何计划任务的命令和输出。

--

--

--

1. (^2\_3^ 2020-06-26)