说明：本 Lumen 中文文档基于 Lumen 5.1.0，更多 Laravel/Lumen 中文学习资源，请访问 Laravel 学院：http://laravelacademy.org。

# 一、起步

# 安装&配置

## 1、安装

### 1.1 服务器要求

Lumen 框架有少许的服务器要求，当然，Laravel Homestead 虚拟机满足所有这些要求：

- PHP >= 5.5.9
- OpenSSL PHP Extension
- Mbstring PHP Extension
- Tokenizer PHP Extension

### 1.2 安装 Lumen

Lumen 使用 Composer 来管理依赖，所以，在使用之前，确保你已经在机器上安装了 Composer。

**通过 Lumen 安装器**

首先，使用 Composer 下载 Lumen 安装器：

```
composer global require "laravel/lumen-installer=~1.0"
```

确保`~/.composer/vendor/bin`在系统路径 PATH 中，否则不能在命令行调用`lumen`命令。安装完成后，只需简单通过`lumen new`命令就可以在当前目录下创建一个新的 Lumen 应用，例如，`lumen new blog`将会创建一个名为`blog`的 Lumen 安装目录，该目录中已经包含了所有 Lumen 依赖。该安装方法比通过 Composer 安装要快很多：

```
lumen new blog
```

**通过 Composer 安装**

你还可以在终端中通过 Composer 的`create-project`目录来安装 Lumen：

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

# 2、配置

## 2.1 基本配置

和完整 Laravel 框架有着多个配置文件不同，Lumen 框架的所有配置项都放在单个`.env`配置文件中。

**应用 APP_KEY**

安装完 Lumen 后，需要设置应用 APP_KEY 为 32 位长的随机字符串，该 key 被配置在`.env`环境文件中（APP_KEY），如果你还没有将`.env.example`文件重命名为`.env`，现在立即这样做。如果应用 key 没有被设置，用户 sessions 和其它加密数据将会有安全隐患！

注意：为了让配置值被加载，你需要取消`bootstrap/app.php`文件中`Dotenv::load()`方法前面的注释。

**更多配置**

Lumen 几乎不再需要其它任何配置就可以使用了，你可以自由地开始开发了！

你可能还想要配置 Lumen 的一些其它组件，比如：

- 缓存
- 数据库

**美化 URL**

- **Apache**

框架中自带的`public/.htaccess`文件支持 URL 中隐藏`index.php`，如过你的 Lumen 应用使用 Apache 作为服务器，需要先确保 Apache 启用了`mod_rewrite`模块以支持`.htaccess`解析。

如果 Lumen 自带的`.htaccess`文件不起作用，试试将其中内容做如下替换：

```
Options +FollowSymLinks
RewriteEngine On

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

- **Nginx**

在 Nginx 中，使用如下站点配置指令就可以支持 URL 美化：

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

当然，使用 Homestead 的话，以上配置已经为你配置好以支持 URL 美化。

## 2.2 环境配置

基于应用运行环境拥有不同配置值能够给我们开发带来极大的方便，比如，我们想在本地和线上环境配置不同的缓存驱动，在 Lumen 中这很容易实现。

Lumen 使用了 Vance Lucas 开发的 PHP 库 DotEnv 来实现这一目的，在新安装的 Lumen 中，根目录下有一个`.env.example`文件，如果 Lumen 是通过 Composer 安装的，那么该文件已经被重命名为`.env`，否则的话你要自己手动重命名该文件。

在每次应用接受请求时，`.env`中列出的所有变量都会被载入到 PHP 超全局变量`$_ENV`中，然后你就可以在应用中通过帮助函数`env`来获取这些变量值。实际上，如果你去查看 Lumen 的配置文件，就会发现很多选项已经在使用这些帮助函数了。

你可以尽情的按你所需对本地服务器上的环境变量进行修改，线上环境也是一样。但不要把`.env`文件提交到源码控制（svn 或 git 等）中，因为每个使用你的应用的不同开发者或服务器可能要求不同的环境配置。

如果你是在一个团队中进行开发，你可能需要将`.env.example`文件随你的应用一起提交到源码控制中，通过将一些配置值以占位符的方式放置在`.env.example`文件中，其他开发者可以很清楚明了的知道运行你的应用需要配置哪些环境变量。

**配置文件**

你可以使用 Laravel 风格的配置文件，这些默认文件存放在`vendor/laravel/lumen-framework/config`目录下，如果你将这些文件拷贝并粘贴到应用根目录的`config`目录下，Lumen 将会使用这些配置文件。

使用完整的配置文件你将能够对 Lumen 的配置有着更多的控制权，例如配置多个存储“硬盘”或者数据库读/写连接。

**自定义配置文件**

你还可以创建自定义的配置文件并使用`$app->configure()`方法来加载它们。例如，如果你的配置文件位于`config/options.php`，你可以像这样加载它：

```
$app->configure('options');
```

**访问当前应用环境**

你可以通过`App`门面的`environment`方法来访问应用当前环境：

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

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

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

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

应用实例也可以通过帮助函数`app`来访问：

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

### 2.3 访问配置值

你可以使用全局的帮助函数 `config` 来访问配置值，配置值可以通过”.”来分隔配置文件和配置选项，如果配置选项不存在的话则会返回默认值：

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

如果要在运行时设置配置值，传递一个数组到 `config` 帮助函数：

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

# 发行版本说明

## 1、5.1.0

Lumen 5.1.0 将框架升级为使用 Laravel 5.1，新特性如事件广播、中间价参数，以及测试优化在新版的 Lumen 中可以使用。想要了解完整的 Laravel 5.1 发行版本说明，查看 Laravel 文档。
以前版本发行说明，查看 Lumen 官网。

# 二、基础

# HTTP 路由

## 1、基本路由

大部分路由都定义在 `bootstrap/app.php` 文件载入的 `app/Http/routes.php` 中。
最基本的 Lumen 路由接收一个 URI 和一个闭包：

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

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

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

```
});

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

**生成指向路由对应的 URL**

可以使用帮助函数 `url` 来生成路由对应的 URL：

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

# 2、路由参数

## 2.1 必选参数

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

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

可以按需要定义在路由中定义多个路由参数：

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

路由参数总是通过花括号进行包裹，参数在路由被执行时会被传递到路由的闭包。

注意：路由参数不能包含'-'字符，需要的话可以使用_替代。

## 2.2 正则约束

可以通过在路由中定义定义正则表达式来约束路由参数格式：

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

# 3、命名路由

命名路由使生成 URL 或者重定向到指定路由变得很方便，在定义路由时指定路由名称，然后使用数组键 `as` 指定路由别名：

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

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

还可以为控制器动作指定路由名称：

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

**生成指向命名路由的 URL**

一旦你为给定路由分配了名字，通过 route 函数生成 URL 时就可以使用路由名字：

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

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

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

# 4、路由分组

路由分组允许我们在多个路由中共享路由属性，比如中间件和命名空间等，这样的话一大波共享属性的路由就不必再各自定义这些属性。共享属性以数组的形式被作为第一个参数传递到 `Route::group` 方法中。

想要了解更多路由分组，我们希望通过几个简单的应用实例来展示其特性。

## 4.1 中间件

要分配中间件给分组中的所有路由，可以在分组属性数组中使用 `middleware` 键。中间件将会按照数组中定义的顺序依次执行：

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

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

### 4.2 命名空间

另一个通用的例子是路由分组分配同一个 PHP 命名空间给多个控制器，可以在分组属性数组中使用 `namespace` 参数来指定分组中控制器的命名空间：

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

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

默认情况下，`RouteServiceProvider` 包含 `routes.php` 并指定其所在命名空间为 App\Http\Controllers，因此，我们只需要指定 `App\Http\Controllers` 之后的相对命名空间即可。

### 4.3 路由前缀

属性 `prefix` 可以用来为分组中每个给定 URI 添加一个前缀，比如，你想要为所有路由 URI 前面添加前缀 `admin`：

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

你还可以使用 `prefix` 参数为分组路由指定公共参数：

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

## 5、CSRF 攻击及保护

注意：使用 Lumen 的该特性之前必须开启 session。

### 5.1 简介

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

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

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

帮助函数 `csrf_field` 生成如下 HTML：

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

当然还可以使用 Blade 模板引擎提供的方式：

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

你不需要了解在 POST、PUT 或者 DELETE 请求时 CSRF 令牌是如何进行验证的，HTTP 中间件 `VerifyCsrfToken` 会为我们做这项工作：将请求中输入的 `token` 值和 session 中的存储的作对比。

### 5.2 X-CSRF-Token

除了将 CSRF 令牌作为一个 POST 参数进行检查，Lumen 的 `VerifyCsrfToken` 中间件还会检查 `X-CSRF-TOKEN` 请求头，你可以将令牌保存在”meta”标签中：

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

创建完这个 meta 标签后，就可以在 js 库如 jQuery 中添加该令牌到所有请求头，这为基于 AJAX 的应用提供了简单、方便的方式来避免 CSRF 攻击：

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

### 5.3 X-XSRF-Token

Lumen 还将 CSRF 令牌保存到了名为 `XSRF-TOKEN` 的 cookie 中，你可以使用该 cookie 值来设置 `X-XSRF-TOKEN` 请求头。一些 JavaScript 框架，比如 Angular，将会为你自动进行设置，基本上你不太会手动设置这个值。

## 6、表单方法伪造

HTML 表单不支持 `PUT`、`PATCH` 或者 `DELETE` 动作，因此，当定义被 HTML 表单调用的 `PUT`、`PATCH` 或 `DELETE` 路由时，需要添加一个隐藏的 `_method` 字段到给表单中，其值被用作 HTTP 请求方法名：

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

## 7、抛出 404 错误

有两者方法手动从路由触发 404 错误。

第一种，使用帮助函数 `abort`，`abort` 函数会抛出一个指定状态码的 `Symfony\Component\HttpFoundation\Exception\HttpException`：

```
abort(404);
```

第二种，手动抛出 `Symfony\Component\HttpKernel\Exception\NotFoundHttpException`.的实例。

更多关于处理 404 异常的信息以及如何自定义视图显示这些错误信息，请查看错误文档一节。

# HTTP 中间件

## 1、简介

HTTP 中间件提供了一个便利的机制来过滤进入应用的 HTTP 请求。例如，Lumen 包含了一个中间件来验证用户是否经过授权，如果用户没有经过授权，中间件会将用户重定向到登录页面，否则如果用户经过授权，中间件就会允许请求继续往前进入下一步操作。

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

## 2、定义中间件

中间件通常都放在 `app/Http/Middleware` 目录下。想要创建一个新的中间件，需要在新创建的中间件中重写 `handle` 方法。在下面中间件中，我们只允许提供的 age 大于 200 的访问路由，否则，我们将用户重定向到主页：

```
<?php

namespace App\Http\Middleware;

use Closure;

class OldMiddleware
{
    /**
     * 返回请求过滤器
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
```

```
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if ($request->input('age') <= 200) {
            return redirect('home');
        }

        return $next($request);
    }

}
```

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

## 2.1 中间件之前/之后

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

```
<?php

namespace App\Http\Middleware;

use Closure;

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

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

然而，下面这个中间件则会在请求处理后执行其任务：

```
<?php

namespace App\Http\Middleware;
```

```
use Closure;

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

        // 执行动作

        return $response;
    }
}
```

# 3、注册中间件

## 3.1 全局中间件

如果你想要中间件在每一个 HTTP 请求期间被执行，只需要将相应中间件类放到 `bootstrap/app.php` 文件的`$app->middleware()`调用中即可。

## 3.2 分配中间件到路由

如果你想要分配中间件到指定路由，首先应该在 `bootstrap/app.php` 文件中分配给该中间件一个简写的 key，默认情况下，`$app->routeMiddleware()`方法包含了 Lumen 自带的入口中间件，添加你自己的中间件只需要将其追加到后面并为其分配一个 key：

```
$app->routeMiddleware([
    'old' => 'App\Http\Middleware\OldMiddleware',]);
```

中间件在入口文件中被定义好了之后，可以在路由选项数组中使用 `middleware` 键来指定中间件：

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

# 4、中间件参数

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

```
<?php

namespace App\Http\Middleware;
```

```
use Closure;

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

        return $next($request);
    }

}
```

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

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

# 5、中止中间件

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

```
<?php

namespace Illuminate\Session\Middleware;

use Closure;
```

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

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

`terminate` 方法将会接收请求和响应作为参数。一旦你定义了一个终结中间件，应该将其加入到入口文件的全局中间件列表中。

# HTTP 控制器

## 1、简介

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

## 2、基本控制器

下面是一个基本控制器类的例子。所有的 Lumen 控制器应该继承自 Lumen 安装默认的基本控制器：

```
<?php

namespace App\Http\Controllers;

use App\User;

class UserController extends Controller
{
    /**
     * 为指定用户显示详情
     *
     * @param  int  $id
     * @return Response
```

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

我们可以像这样路由到控制器动作：

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

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

### 2.1 控制器&命名空间

你应该注意到我们在定义控制器路由的时候没有指定完整的控制器命名空间，我们只需要定义 `App\Http\Controllers` 之后的类名部分。默认情况下，`bootstrap/app.php` 将会在一个路由分组中载入 `routes.php` 文件，该路由分组包含了控制器的根命名空间。
如果你在 `App\Http\Controllers` 目录下选择使用 PHP 命名空间嵌套或组织控制器，只需要使用相对于 `App\Http\Controllers` 根命名空间的指定类名即可。因此，如果你的完整控制器类是 `App\Http\Controllers\Photos\AdminController`，你可以像这样注册路由：

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

### 2.2 命名控制器路由

和闭包路由一样，可以指定控制器路由的名字：

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

一旦你为控制器路由分配了名字，那么你就可以使用帮助函数 `action` 很方便的生成 URL 到 action，这里我们也只需要指定相对 `App\Http\Controllers` 的命名空间即可：

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

你还可以使用帮助函数 `route` 来为已命名的控制器路由生成 URL：

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

## 3、控制器中间件

中间件可以像这样分配给控制器路由：

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

```
]);
```

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

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

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

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

# 4、依赖注入&控制器

## 4.1 构造函数注入

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

```
<?php

namespace App\Http\Controllers;

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

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

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

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

## 4.2 方法注入

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

```
<?php

namespace App\Http\Controllers;

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

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

        //
    }
}
```

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

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

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

```
<?php

namespace App\Http\Controllers;

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

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

# HTTP 请求

## 1、访问请求

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

```
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller
```

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

        //
    }
}
```

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

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

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

```
<?php

namespace App\Http\Controllers;

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

classUser Controller extends Controller
{
    /**
     * 更新指定用户
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
     public function update(Request $request,$id)
    {
        //
    }
}
```

## 1.1 基本请求信息

`Illuminate\Http\Request` 实例提供了多个方法来检测应用的 HTTP 请求，Lumen 的 `Illuminate\Http\Request` 继承自 `Symfony\Component\HttpFoundation\Request` 类，这里列出了一些该类中的有用方法：

**获取请求 URI**

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

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

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

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

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

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

**获取请求方法**

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

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

## 1.2 PSR-7 请求

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

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

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

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

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

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

# 2、获取输入

**获取输入值**

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

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

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

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

处理表单数组输入时，可以使用”.”来访问数组：

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

**判断输入值是否出现**

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

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

**获取所有输入数据**

你还可以通过 `all` 方法获取所有输入数据：

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

**获取输入的部分数据**

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

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

## 2.1 老的输入

注意：在使用该特性前必须开启 session。

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

**将输入存储到一次性 Session**

`Illuminate\Http\Request` 实例的 `flash` 方法会将当前输入存放到一次性 session 中，这样在下一次请求时数据依然有效：

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

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

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

**将输入存储到一次性 Session 然后重定向**

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

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

**取出老数据**

要从上次请求取出一次性存储的输入数据，可以使用 Request 实例的 `old` 方法。`old` 方法提供了便利的方式从 session 中取出一次性数据：

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

Lumen 还提供了一个全局的帮助函数 `old`，如果你是在 Blade 模板中显示老数据，使用帮助函数 `old` 更方便：

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

## 2.2 Cookies

要强制所有 cookies 被加密和签名，需要取消 bootstrap/app.php 文件中 EncryptCookies 中间件前面的注释。Lumen 及 Laravel 框架创建的所有签名 cookies 通过一个认证码进行加密和签名，这样如果客户端试图篡改 cookies 则该 cookies 将会失效。

**从请求中取出 Cookies**

我们使用 `Illuminate\Http\Request` 实例的 cookie 方法从请求中获取 cookie 的值：

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

**新增 Cookie 到响应**

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

```
$response = new Illuminate\Http\Response('Hello World');
$response->withCookie(cookie('name', 'value', $minutes));
return $response;
```

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

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

## 2.3 文件上传

**获取上传的文件**

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

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

**验证文件是否存在**

使用 `hasFile` 方法判断文件在请求中是否存在：

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

**验证文件是否上传成功**

使用 `isValid` 方法判断文件在上传过程中是否出错：

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

**保存上传的文件**

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

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

**其它文件方法**

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

# HTTP 响应

## 1、基本响应

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

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

给定的字符串会被框架自动转化为 HTTP 响应。

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

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

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

为方便起见，还可以使用帮助函数 `response`：

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

注意：查看完整的 `Response` 方法列表，请移步相应的 API 文档以及 Symfony AP 文档

### 1.1 添加响应头

大部分响应方法都是可以链式调用的，从而使得可以平滑的构建响应。例如，可以使用 `header` 方法来添加一系列响应头：

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

### 1.2 添加 Cookies

使用 response 实例的帮助函数 `withCookie` 可以轻松添加 cookie 到响应，比如，可以使用 `withCookie` 方法来生成 cookie 并将其添加到 response 实例：

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

`withCookie` 方法接收额外的可选参数从而允许对 cookie 属性更多的自定义：

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

## 2、其它响应类型

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

### 2.1 视图响应

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

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

当然，如果你不需要传递一个自定义的 HTTP 状态码或者自定义头，只需要简单使用全局的帮助函数 `view` 即可。

### 2.2 JSON 响应

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

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

如果你想要创建一个 JSONP 响应，可是添加 setCallback 到 json 方法后面：

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

### 2.3 文件下载

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

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

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

## 3、重定向

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

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

有时候你想要将用户重定向到前一个位置，比如，表单提交后，验证不通过，你就可以使用 back 帮助函数返回前一个 URL：

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

## 3.1 重定向到命名路由

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

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

如果路由中有参数，可以将其作为第二个参数传递到 route 方法：

```
// For a route with the following URI: profile/{id}
return redirect()->route('profile', [1]);
```

如果要重定向到带 ID 参数的路由，并从 Eloquent 模型中取数据填充表单，可以传递模型本身，ID 会被自动解析出来：

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

## 3.2 带一次性 Session 数据的重定向

注意：使用此特性需要开启 session。

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

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

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

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

# 视图

## 1、基本使用

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

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

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

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

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

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

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

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

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

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

### 1.1 视图数据

**传递数据到视图**
在上述例子中可以看到，我们可以简单通过数组方式将数据传递到视图：

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

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

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

**在视图间共享数据**

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

```
<?php

namespace App\Providers;

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

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

# 三、更多特性

# 缓存

## 1、配置

Lumen 为不同的缓存系统提供了统一的 API。缓存配置项位于 env 文件。在该文件中你可以指定在应用中默认使用哪个缓存驱动。Lumen 目前支持流行的缓存后端如 Memcached 和 Redis 等。对于大型应用，推荐使用内存缓存如 Memcached 或 APC。

### 1.1 缓存预备知识

**数据库**

使用 `database` 缓存驱动时，你需要设置一张表包含缓存缓存项。下面是该表的 `Schema` 声明：

```
Schema::create('cache', function($table) {
    $table->string('key')->unique();
    $table->text('value');
    $table->integer('expiration');
});
```

**Memcached**

使用 Memcached 缓存要求安装了 Memcached PECL 包，即 PHP Memcached 扩展。Memcached::addServer 默认配置使用 TCP/IP 协议。

**Redis**

使用 Lumen 的 Redis 缓存之前，你需要通过 Composer 安装 `predis/predis` 包（~1.0）。

## 2、缓存使用

### 2.1 获取缓存实例

`Illuminate\Contracts\Cache\Factory` 和 `Illuminate\Contracts\Cache\Repository` 契约提供了访问 Laravel 的缓存服务的方法。`Factory` 契约提供了所有访问应用定义的缓存驱动的方法。`Repository` 契约通常是应用中 `cache` 配置文件中指定的默认缓存驱动的一个实现。

然而，你还可以使用 `Cache` 门面，这也是我们在整个文档中使用的方式，`Cache` 门面提供了简单方便的方式对底层 Lumen 缓存契约实现进行访问。

例如，让我们在控制器中导入 `Cache` 门面：

```
<?php

namespace App\Http\Controllers;

use Cache;
```

```
class UserController extends Controller{
    /**
     * 显示应用所有用户列表
     *
     * @return Response
     */
    public function index()
    {
        $value = Cache::get('key');

        //
    }
}
```

**访问多个缓存存储**

使用 `Cache` 门面，你可以使用 `store` 方法访问不同的缓存存储器，传入 `store` 方法的键就是 cache 配置文件中 `stores` 配置数组里列出的相应的存储器：

```
$value = Cache::store('file')->get('foo');
Cache::store('redis')->put('bar', 'baz', 10);
```

## 2.2 从缓存中获取数据

`Cache` 门面的 `get` 方法用于从缓存中获取缓存项，如果缓存项不存在，返回 null。如果需要的话你可以传递第二个参数到 `get` 方法指定缓存项不存在时返回的自定义默认值：

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

你甚至可以传递一个闭包作为默认值，如果缓存项不存在的话闭包的结果将会被返回。传递闭包允许你可以从数据库或其它外部服务获取默认值：

```
$value = Cache::get('key', function() {
    return DB::table(...)->get();
});
```

**检查缓存项是否存在**

`has` 方法用于判断缓存项是否存在：

```
if (Cache::has('key')) {
    //
}
```

**数值增加/减少**

`increment` 和 `decrement` 方法可用于调整缓存中的整型数值。这两个方法都可以接收第二个参数来指明缓存项数值增加和减少的数目：

```
Cache::increment('key');
```

```
Cache::increment('key', $amount);

Cache::decrement('key');
Cache::decrement('key', $amount);
```

**获取或更新**

有时候你可能想要获取缓存项，但如果请求的缓存项不存在时给它存储一个默认值。例如，你可能想要从缓存中获取所有用户，或者如果它们不存在的话，从数据库获取它们并将其添加到缓存中，你可以通过使用 `Cache::remember` 方法实现：

```
$value = Cache::remember('users', $minutes, function() {
    return DB::table('users')->get();});
```

如果缓存项不存在，传递给 `remember` 方法的闭包被执行并且将结果存放到缓存中。

你还可以联合 `remember` 和 `forever` 方法：

```
$value = Cache::rememberForever('users', function() {
    return DB::table('users')->get();
});
```

**获取并删除**

如果你需要从缓存中获取缓存项然后删除，你可以使用 `pull` 方法，和 `get` 方法一样，如果缓存项不存在的话返回 null：

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

## 2.3 存储缓存项到缓存

你可以使用 `Cache` 门面上的 `put` 方法在缓存中存储缓存项。当你在缓存中存储缓存项的时候，你需要指定数据被缓存的时间（分钟数）：

```
Cache::put('key', 'value', $minutes);
```

除了传递缓存项失效时间，你还可以传递一个代表缓存项有效时间的 PHP `Datetime` 实例：

```
$expiresAt = Carbon::now()->addMinutes(10);
Cache::put('key', 'value', $expiresAt);
```

`add` 方法只会在缓存项不存在的情况下添加缓存项到缓存，如果缓存项被添加到缓存返回 `true`，否则，返回 `false`：

```
Cache::add('key', 'value', $minutes);
```

`forever` 方法用于持久化存储缓存项到缓存，这些值必须通过 `forget` 方法手动从缓存中移除：

```
Cache::forever('key', 'value');
```

### 2.4 从缓存中移除数据

你可以使用 `Cache` 门面上的 `forget` 方法从缓存中移除缓存项：

```
Cache::forget('key');
```

# 数据库

## 1、配置

在 Lumen 中连接数据库和运行查询都非常简单，目前 Lumen 支持四种数据库系统：MySQL、Postgres、SQLite 和 SQL Server。

你可以在配置文件 `.env` 中使用配置选项 `DB_*` 来配置数据库设置，例如驱动、主机、用户名和密码。

注意：为了让配置值被加载，你需要取消 `bootstrap/app.php` 文件中 `Dotenv::load()` 调用前的注释。

## 2、基本使用

注意：如果你想要使用 `DB` 门面，应该取消 `bootstrap/app.php` 文件中 `$app->withFacades()` 调用前的注释

举个例子，不起用门面，你可以通过帮助函数 `app` 来访问数据库连接：

```
$results = app('db')->select("SELECT * FROM users");
```

或者，开启了门面的话，你可以使用 `DB` 门面来访问数据库连接：

```
$results = DB::select("SELECT * FROM users");
```

**基本查询**

要了解如何通过数据库组件执行基本、原生 SQL 查询，查看 Laravel 数据库文档。

**查询构建器**

Lumen 还可以使用 Laravel 的查询构建器。要了解更多该特性，查看 Laravel 查询构建器文档。

**Eloquent ORM**

如果你想要使用 Eloquent ORM，应该取消 `bootstrap/app.php` 文件中 `$app->withEloquent()` 调用前的注释。

当然，你可以在 Lumen 中轻松使用完整的 Eloquent ORM。要学习如何使用 Eloquent ORM，查看 Laravel 相应文档。

## 3、迁移

要了解更多如何创建数据表并运行迁移的知识，查看 Laravel 迁移文档。

# 加密

## 1、配置

在使用 Lumen 的加密器之前，应该在 `bootstrap/app.php` 文件中设置 APP_KEY 选项为 32 位随机字符串。如果这个值没有被设置，所有 Lumen 加密过的值都是不安全的。

## 2、基本使用

### 2.1 加密

你可以使用 Crypt 门面对数据进行加密，所有加密值都使用 OpenSSL 和 `AES-256-CBC` 密码进行加密。此外，所有加密值都通过一个消息认证码（MAC）来检测对加密字符串的任何修改。
例如，我们可以使用 `encrypt` 方法加密 `secret` 属性并将其存储到 Eloquent 模型：

```
<?php

namespace App\Http\Controllers;

use Crypt;
use App\User;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserController extends Controller{
    /**
     * Store a secret message for the user.
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function storeSecret(Request $request, $id)
    {
        $user = User::findOrFail($id);

        $user->fill([
            'secret' => Crypt::encrypt($request->secret)
        ])->save();
    }
```

```
}
```

### 2.2 解密

当然，你可以使用 `Crypt` 门面上的 `decrypt` 方法进行解密。如果该值不能被解密，例如 MAC 无效，将会抛出一个 `Illuminate\Contracts\Encryption\DecryptException` 异常：

```
use Illuminate\Contracts\Encryption\DecryptException;

try {
    $decrypted = Crypt::decrypt($encryptedValue);
} catch (DecryptException $e) {
    //
}
```

# 错误&日志

## 1、简介

开始一个新的 Lumen 项目的时候，错误和异常处理已经默认为你配置好了。此外，Lumen 还集成了提供各种功能强大日志处理器的 Monolog 日志库。

## 2、配置

### 2.1 错误详情

配置文件`.env` 中的 APP_DEBUG 配置选项控制浏览器显示的错误详情数量。
对本地开发而言，你应该设置环境变量 `APP_DEBUG` 值为 `true`。在生产环境，该值应该被设置为 `false`。

## 3、异常处理器

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

### 3.1 report 方法

`report` 方法用于记录异常并将其发送给外部服务如 Bugsnag。默认情况下，`report` 方法只是将异常传递给异常被记录的基类，你可以随心所欲的记录异常。
例如，如果你需要以不同方式报告不同类型的异常，可使用 PHP 的 `instanceof` 比较操作符：

```
/**
```

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

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

**通过类型忽略异常**

异常处理器的`$dontReport`属性包含一个不会被记录的异常类型数组，默认情况下，`404`错误异常不会被写到日志文件，如果需要的话你可以添加其他异常类型到这个数组。

### 3.2 render 方法

`render`方法负责将给定异常转化为发送给浏览器的 HTTP 响应，默认情况下，异常被传递给为你生成响应的基类。然而，你可以随心所欲地检查异常类型或者返回自定义响应：

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

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

## 4、HTTP 异常

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

```
abort(404);
```

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

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

该方法可在请求生命周期的任何时间点使用。

# 5、日志

Lumen 日志工具基于强大的 Monolog 库，默认情况下，Lumen 被配置为在 `storage/logs` 目录下每日为应用生成日志文件，你可以使用 `Log` 门面编写日志信息到日志中：

```
<?php

namespace App\Http\Controllers;

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

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

该日志记录器提供了 RFC 5424 中定义的七种日志级别：
**alert**, **critical**, **error**,**warning**, **notice**, **info** 和 **debug**。

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

Log::critical($error);

Log::error($error);

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

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

Log::info($error);

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

### 5.1 上下文信息

上下文数据数组也会被传递给日志方法。上下文数据将会和日志消息一起被格式化和显示：

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

# 事件

## 1、简介

Lumen 事件提供了简单的观察者模式实现，允许你订阅和监听应用中的事件。事件类通常存放在 `app/Events` 目录，监听器存放在 `app/Listeners`。

## 2、注册事件/监听器

Lumen 自带的 `EventServiceProvider` 为事件注册提供了方便之所。其中的 `listen` 属性包含了事件（键）和对应监听器（值）数组。如果应用需要，你可以添加多个事件到该数组。例如，让我们添加 `PodcastWasPurchased` 事件：

```
/**
 * 事件监听器映射
 *
 * @var array
 */
protected $listen = [
    'App\Events\PodcastWasPurchased' => [
        'App\Listeners\EmailPurchaseConfirmation',
    ],
];
```

### 3、定义事件

事件类是一个处理与事件相关的简单数据容器，例如，假设我们生成的 `PodcastWasPurchased` 事件接收一个 Eloquent ORM 对象：

```
<?php

namespace App\Events;

use App\Podcast;
use App\Events\Event;
use Illuminate\Queue\SerializesModels;

class PodcastWasPurchased extends Event{
    use SerializesModels;

    public $podcast;

    /**
     * 创建新的事件实例
     *
     * @param  Podcast  $podcast
     * @return void
     */
    public function __construct(Podcast $podcast)
    {
        $this->podcast = $podcast;
    }
}
```

正如你所看到的，该事件类不包含任何特定逻辑，只是一个存放被购买的 `Podcast` 对象的容器，如果事件对象被序列化的话，事件使用的 `SerializesModels` trait 将会使用 PHP 的 `serialize` 函数序列化所有 Eloquent 模型。

# 4、定义监听器

接下来，让我们看看我们的示例事件的监听器，事件监听器在 `handle` 方法中接收事件实例。在 `handle` 方法内，你可以执行任何需要的逻辑以响应事件。

```
<?php

namespace App\Listeners;

use App\Events\PodcastWasPurchased;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class EmailPurchaseConfirmation{
    /**
```

```
     * 创建事件监听器
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * 处理事件
     *
     * @param  PodcastWasPurchased  $event
     * @return void
     */
    public function handle(PodcastWasPurchased $event)
    {
        // Access the podcast using $event->podcast...
    }
}
```

你的事件监听器还可以在构造器中类型提示任何需要的依赖，所有事件监听器通过服务容器解析，所以依赖会自动注入。

**停止事件继续往下传播**

有时候，你希望停止事件被传播到其它监听器，你可以通过从监听器的 `handle` 方法中返回 `false` 来实现。

## 4.1 事件监听器队列

需要将事件监听器放到队列中？没有比这更简单的了，只需要让监听器类实现 `ShouldQueue` 接口即可，通过 Artisan 命令 `event:generate` 生成的监听器类已经将接口导入当前命名空间，所有你可以立即拿来使用：

```
<?php

namespace App\Listeners;

use App\Events\PodcastWasPurchased;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class EmailPurchaseConfirmation implements ShouldQueue{
    //
}
```

就是这么简单，当监听器被事件调用，将会使用 Lumen 的队列系统通过队列分发器自动队列化。如果通过队列执行监听器的时候没有抛出任何异常，队列任务在执行完成后被自动删除。

**手动访问队列**

如果你需要手动访问底层队列任务的 `delete` 和 `release` 方法，在生成的监听器中默认导入的 `Illuminate\Queue\InteractsWithQueue` trait 提供了访问这两个方法的权限：

```
<?php

namespace App\Listeners;

use App\Events\PodcastWasPurchased;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class EmailPurchaseConfirmation implements ShouldQueue{
    use InteractsWithQueue;

    public function handle(PodcastWasPurchased $event)
    {
        if (true) {
            $this->release(30);
        }
    }
}
```

# 5、触发事件

要触发一个事件，可以使用 Event 门面，传递一个事件实例到 `fire` 方法，`fire` 方法会分发事件到所有监听器：

```
<?php

namespace App\Http\Controllers;

use Event;
use App\Podcast;
use App\Events\PodcastWasPurchased;
use App\Http\Controllers\Controller;

class UserController extends Controller{
    /**
     * 显示指定用户属性
     *
```

```
     * @param  int  $userId
     * @param  int  $podcastId
     * @return Response
     */
    public function purchasePodcast($userId, $podcastId)
    {
        $podcast = Podcast::findOrFail($podcastId);

        // Purchase podcast logic...

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

此外，你还可以使用全局的帮助函数 `event` 来触发事件：

```
event(new PodcastWasPurchased($podcast));
```

# 6、广播事件

在很多现代 web 应用中，web 套接字被用于实现实时更新的用户接口。当一些数据在服务器上被更新，通常一条消息通过 websocket 连接被发送给客户端处理。

为帮助你构建这样的应用，Lumen 让通过 websocket 连接广播事件变得简单。广播 Lumen 事件允许你在服务端和客户端 JavaScript 框架之间共享同一事件名。

## 6.1 配置

Lumen 支持多种广播驱动：Pusher、Redis 以及一个服务于本地开发和调试的日志驱动。每一个驱动都有一个配置示例。BROADCAST_DRIVER 配置选项可用于设置默认驱动。

**广播预备知识**

事件广播需要以下两个依赖：

- Pusher: `pusher/pusher-php-server ~2.0`
- Redis: `predis/predis ~1.0`

**队列预备知识**

在开始介绍广播事件之前，还需要配置并运行一个队列监听器。所有事件广播都通过队列任务来完成以便应用的响应时间不受影响。

## 6.2 将事件标记为广播

要告诉 Lumen 给定事件应该被广播，需要在事件类上实现 `Illuminate\Contracts\Broadcasting\ShouldBroadcast` 接口。`ShouldBroadcast` 接口要求你实现一个方法：`broadcastOn`。该方法应该返回事件广播"频道"名称数组：

```
<?php

namespace App\Events;
```

```
use App\User;
use App\Events\Event;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class ServerCreated extends Event implements ShouldBroadcast{
    use SerializesModels;

    public $user;

    /**
     * 创建新的事件实例
     *
     * @return void
     */
    public function __construct(User $user)
    {
        $this->user = $user;
    }

    /**
     * 获取事件广播频道
     *
     * @return array
     */
    public function broadcastOn()
    {
        return ['user.'.$this->user->id];
    }
}
```

然后，你只需要和正常一样触发该事件，事件被触发后，一个队列任务将通过指定广播驱动自动广播该事件。

## 6.3 广播数据

如果某个事件被广播，其所有的 `public` 属性都会按照事件负载自动序列化和广播，从而允许你从 JavaScript 中访问所有 `public` 数据，因此，举个例子，如果你的事件有一个单独的包含 Eloquent 模型的 `$user` 属性，广播负载定义如下：

```
{
    "user": {
        "id": 1,
        "name": "Jonathan Banks"
        ...
```

```
    }
}
```

然而，如果你希望对广播负载有更加细粒度的控制，可以添加 `broadcastWith` 方法到事件，该方法应该返回你想要通过事件广播的数组数据：

```
/**
 * 获取广播数据
 *
 * @return array
 */
public function broadcastWith(){
    return ['user' => $this->user->id];
}
```

## 6.4 消费事件广播

**Pusher**

你可以通过 Pusher 的 JavaScript SDK 方便地使用 Pusher 驱动消费事件广播。例如，让我们从之前的例子中消费 App\Events\ServerCreated 事件：

```
this.pusher = new Pusher('pusher-key');

this.pusherChannel = this.pusher.subscribe('user.' + USER_ID);

this.pusherChannel.bind('App\\Events\\ServerCreated', function
(message) {
    console.log(message.user);
});
```

**Redis**

如果你在使用 Redis 广播，你将需要编写自己的 Redis pub/sub 消费者来接收消息并使用自己选择的 websocket 技术将其进行广播。例如，你可以选择使用使用 Node 编写的流行的 Socket.io 库。

使用 Node 库 `socket.io` 和 `ioredis`，你可以快速编写事件广播发布所有广播事件：

```
var app = require('http').createServer(handler);
var io = require('socket.io')(app);

var Redis = require('ioredis');
var redis = new Redis();

app.listen(6001, function() {
    console.log('Server is running!');});

function handler(req, res) {
    res.writeHead(200);
```

```
    res.end('');}

io.on('connection', function(socket) {
    //
});

redis.psubscribe('*', function(err, count) {
    //
});

redis.on('pmessage', function(subscribed, channel, message) {
    message = JSON.parse(message);
    io.emit(channel + ':' + message.event, message.data);
});
```

## 7、事件订阅者

事件订阅者是指那些在类本身中订阅到多个事件的类，从而允许你在单个类中定义一些事件处理器。订阅者应该定义一个 `subscribe` 方法，该方法中传入一个事件分发器实例：

```
<?php

namespace App\Listeners;

class UserEventListener{
    /**
     * 处理用户登录事件
     */
    public function onUserLogin($event) {}

    /**
     * 处理用户退出事件
     */
    public function onUserLogout($event) {}

    /**
     * 为订阅者注册监听器
     *
     * @param  Illuminate\Events\Dispatcher  $events
     * @return array
     */
    public function subscribe($events)
    {
```

```
        $events->listen(
            'App\Events\UserLoggedIn',
            'App\Listeners\UserEventListener@onUserLogin'
        );

        $events->listen(
            'App\Events\UserLoggedOut',
            'App\Listeners\UserEventListener@onUserLogout'
        );
    }

}
```

## 7.1 注册一个事件订阅者

订阅者被定义后，可以通过事件分发器进行注册，你可以使用 `EventServiceProvider` 上的 `$subcribe` 属性来注册订阅者。例如，让我们添加 `UserEventListener`：

```
<?php

namespace App\Providers;

use Illuminate\Contracts\Events\Dispatcher as DispatcherContrac
t;
use Illuminate\Foundation\Support\Providers\EventServiceProvide
r as ServiceProvider;

class EventServiceProvider extends ServiceProvider{
    /**
     * 事件监听器映射数组
     *
     * @var array
     */
    protected $listen = [
        //
    ];

    /**
     * 要注册的订阅者
     *
     * @var array
     */
    protected $subscribe = [
        'App\Listeners\UserEventListener',
    ];
```

```
}
```

# 队列

## 1、简介

Lumen 队列服务为各种不同的后台队列提供了统一的 API。队列允许你推迟耗时任务（例如发送邮件）的执行，从而大幅提高 web 请求速度。

### 1.1 配置

.env 文件的 QUEUE_DRIVER 选项决定应用使用的队列“驱动”。

### 1.2 队列驱动预备知识

**数据库**

为了使用 `database` 队列驱动，需要一张数据库表来存放任务，要生成创建该表的迁移，运行 Artisan 命令 `queue:table`，迁移被创建好了之后，使用 `migrate` 命令运行迁移：

```
php artisan queue:table
php artisan migrate
```

**其它队列依赖**

下面是以上列出队列驱动需要安装的依赖：

- Amazon SQS: `aws/aws-sdk-php ~3.0`
- Beanstalkd: `pda/pheanstalk ~3.0`
- IronMQ: `iron-io/iron_mq ~2.0`
- Redis: `predis/predis ~1.0`

## 2、编写任务类

### 2.1 任务类结构

默认情况下，应用的所有队列任务都存放在 `app/Jobs` 目录。任务类非常简单，正常情况下只包含一个当队列处理该任务时被执行的 `handle` 方法，让我们看一个任务类的例子：

```
<?php

namespace App\Jobs;

use App\User;
use App\Jobs\Job;
```

```
use Illuminate\Contracts\Mail\Mailer;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Bus\SelfHandling;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendReminderEmail extends Job implements SelfHandling, Sh
ouldQueue
{
    use InteractsWithQueue, SerializesModels;

    protected $user;

    /**
     * 创建一个新的任务实例
     *
     * @param  User  $user
     * @return void
     */
    public function __construct(User $user)
    {
        $this->user = $user;
    }

    /**
     * 执行任务
     *
     * @param  Mailer  $mailer
     * @return void
     */
    public function handle(Mailer $mailer)
    {
        $mailer->send('emails.reminder', ['user' => $this->use
r], function ($m) {
            //
        });

        $this->user->reminders()->create(...);
    }
}
```

在本例中，注意我们能够直接将 Eloquent 模型传递到对列任务的构造函数中。由于该任务使用了 `SerializesModels` trait，Eloquent 模型将会在任务被执行是优雅地序列化和反序列化。如果你的队列任务在构造函数中接收 Eloquent 模型，只有模型的主键会被序列化到队

列，当任务真正被执行的时候，队列系统会自动从数据库中获取整个模型实例。这对应用而言是完全透明的，从而避免序列化整个 Eloquent 模型实例引起的问题。

`handle` 方法在任务被队列处理的时候被调用，注意我们可以在任务的 `handle` 方法中对依赖进行类型提示。Lumen 服务容器会自动注入这些依赖。

**出错**

如果任务被处理的时候抛出异常，则该任务将会被自动释放回队列以便再次尝试执行。任务会持续被释放知道尝试次数达到应用允许的最大次数。最大尝试次数通过 Artisan 任务 `queue:listen` 或 `queue:work` 上的`--tries` 开关来定义。关于运行队列监听器的更多信息可以在下面看到。

**手动释放任务**

如果你想要手动释放任务，生成的任务类中自带的 `InteractsWithQueue` trait 提供了释放队列任务的 `release` 方法，该方法接收一个参数——同一个任务两次运行之间的等待时间：

```
public function handle(Mailer $mailer){
    if (condition) {
        $this->release(10);
    }
}
```

**检查尝试运行次数**

正如上面提到的，如果在任务处理期间发生异常，任务会自动释放回队列中，你可以通过 `attempts` 方法来检查该任务已经尝试运行次数：

```
public function handle(Mailer $mailer){
    if ($this->attempts() > 3) {
        //
    }
}
```

# 3、推送任务到队列

默认的 Lumen 控制器位于 `app/Http/Controllers/Controller.php` 并使用了 `DispatchesJobs` trait。该 trait 提供了一些允许你方便推送任务到队列的方法，例如 `dispatch` 方法：

```
<?php

namespace App\Http\Controllers;

use App\User;
use Illuminate\Http\Request;
use App\Jobs\SendReminderEmail;
use App\Http\Controllers\Controller;

class UserController extends Controller{
```

```
    /**
     * 发送提醒邮件到指定用户
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function sendReminderEmail(Request $request, $id)
    {
        $user = User::findOrFail($id);

        $this->dispatch(new SendReminderEmail($user));
    }
}
```

**为任务指定队列**

你还可以指定任务被发送到的队列。

通过推送任务到不同队列，你可以对队列任务进行“分类”，甚至优先考虑分配给多个队列的 worker 数目。这并不会如队列配置文件中定义的那样将任务推送到不同队列“连接”，而只是在单个连接中发送给特定队列。要指定该队列，使用任务实例上的 `onQueue` 方法，该方法有 Lumen 自带的基类 `App\Jobs\Job` 提供：

```
<?php

namespace App\Http\Controllers;

use App\User;
use Illuminate\Http\Request;
use App\Jobs\SendReminderEmail;
use App\Http\Controllers\Controller;

class UserController extends Controller{
    /**
     * 发送提醒邮件到指定用户
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function sendReminderEmail(Request $request, $id)
    {
        $user = User::findOrFail($id);
        $job = (new SendReminderEmail($user))->onQueue('emails
');
```

```
        $this->dispatch($job);
    }
}
```

### 3.1 延迟任务

有时候你可能想要延迟队列任务的执行。例如，你可能想要将一个注册 15 分钟后给消费者发送提醒邮件的任务放到队列中，可以通过使用任务类上的 `delay` 方法来实现，该方法由 `Illuminate\Bus\Queueable` trait 提供：

```
<?php

namespace App\Http\Controllers;

use App\User;
use Illuminate\Http\Request;
use App\Jobs\SendReminderEmail;
use App\Http\Controllers\Controller;

class UserController extends Controller{
    /**
     * 发送提醒邮件到指定用户
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function sendReminderEmail(Request $request, $id)
    {
        $user = User::findOrFail($id);
        $job = (new SendReminderEmail($user))->delay(60);
        $this->dispatch($job);
    }
}
```

在本例中，我们指定任务在队列中开始执行前延迟 60 秒。

注意：Amazon SQS 服务最大延迟时间是 15 分钟。

### 3.2 从请求中分发任务

映射 HTTP 请求变量到任务中很常见，Lumen 提供了一些帮助函数让这种实现变得简单，而不用每次请求时手动执行映射。让我么看一下 `DispatchesJobs` trait 上的 `dispatchFrom` 方法。默认情况下，该 trait 包含在 Lumen 控制器基类中：

```
<?php
```

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class CommerceController extends Controller{
    /**
     * 处理指定订单
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function processOrder(Request $request, $id)
    {
        // 处理请求...
        $this->dispatchFrom('App\Jobs\ProcessOrder', $request);
    }
}
```

该方法检查给定任务类的构造函数并从 HTTP 请求（或者其它 `ArrayAccess` 对象）中解析变量来填充任务需要的构造函数参数。所以，如果我们的任务类在构造函数中接收一个 `productId` 变量，该任务将会尝试从 HTTP 请求中获取 `productId` 参数。

你还可以传递一个数组作为 `dispatchFrom` 方法的第三个参数。该数组用于填充所有请求中不存在的构造函数参数：

```
$this->dispatchFrom('App\Jobs\ProcessOrder', $request, [
    'taxPercentage' => 20,
]);
```

# 4、运行队列监听器

**开启任务监听器**

Lumen 包含了一个 Artisan 命令用来运行推送到队列的新任务。你可以使用 `queue:listen` 命令运行监听器：

```
php artisan queue:listen
```

还可以指定监听器使用哪个队列连接：

```
php artisan queue:listen connection
```

注意一旦任务开始后，将会持续运行直到手动停止。你可以使用一个过程监视器如 Supervisor 来确保队列监听器没有停止运行。

**队列优先级**

你可以传递逗号分隔的队列连接列表到 `listen` 任务来设置队列优先级：

```
php artisan queue:listen --queue=high,low
```

在本例中，`high` 队列上的任务总是在从 `low` 队列移动任务之前被处理。

**指定任务超时参数**

你还可以设置每个任务允许运行的最大时间（以秒为单位）：

```
php artisan queue:listen --timeout=60
```

**指定队列睡眠时间**

此外，可以指定轮询新任务之前的等待时间（以秒为单位）：

```
php artisan queue:listen --sleep=5
```

需要注意的是队列只会在队列上没有任务时“睡眠”，如果存在多个有效任务，该队列会持续运行，从不睡眠。

## 4.1 Supervisor 配置

Supervisor 为 Linux 操作系统提供的进程监视器，将会在失败时自动重启 `queue:listen` 或 `queue:work` 命令，要在 Ubuntu 上安装 Supervisor，使用如下命令：

```
sudo apt-get install supervisor
```

Supervisor 配置文件通常存放在`/etc/supervisor/conf.d` 目录，在该目录中，可以创建多个配置文件指示 Supervisor 如何监视进程，例如，让我们创建一个开启并监视 `queue:work` 进程的 `laravel-worker.conf` 文件：

```
[program:laravel-worker]
process_name=%(program_name)s_%(process_num)02d
command=php /home/forge/app.com/artisan queue:work sqs --sleep=
3 --tries=3 --daemon
autostart=true
autorestart=true
user=forge
numprocs=8
redirect_stderr=true
stdout_logfile=/home/forge/app.com/worker.log
```

在本例中，`numprocs` 指令让 Supervisor 运行 8 个 `queue:work` 进程并监视它们，如果失败的话自动重启。配置文件创建好了之后，可以使用如下命令更新 Supervisor 配置并开启进程：

```
sudo supervisorctl reread
sudo supervisorctl update
sudo supervisorctl start laravel-worker:*
```

要了解更多关于 Supervisor 的使用和配置，查看 Supervisor 文档。此外，还可以使用 Lumen Forge 从 web 接口方便地自动配置和管理 Supervisor 配置。

### 4.2 后台队列监听器

Artisan 命令 `queue:work` 包含一个`--daemon`选项来强制队列 worker 持续处理任务而不必重新启动框架。相较于 `queue:listen` 命令该命令对 CPU 的使用有明显降低：

```
php artisan queue:work connection --daemon
php artisan queue:work connection --daemon --sleep=3
php artisan queue:work connection --daemon --sleep=3 --tries=3
```

正如你所看到的，`queue:work` 任务支持大多数 `queue:listen` 中有效的选项。你可以使用 `php artisan help queue:work` 任务来查看所有有效选项。

**后台队列监听器编码考虑**

后台队列 worker 在处理每个任务时不重启框架，因此，你要在任务完成之前释放资源，举个例子，如果你在使用 GD 库操作图片，那么就在完成时使用 `imagedestroy` 释放内存。类似的，数据库连接应该在后台长时间运行完成后断开，你可以使用 `DB::reconnect` 方法确保获取了一个新的连接。

### 4.3 部署后台队列监听器

由于后台队列 worker 是常驻进程，不重启的话不会应用代码中的更改，所以，最简单的部署后台队列 worker 的方式是使用部署脚本重启所有 worker，你可以通过在部署脚本中包含如下命令重启所有 worker：

```
php artisan queue:restart
```

该命令会告诉所有队列 worker 在完成当前任务处理后重启以便没有任务被遗漏。

注意：这个命令依赖于缓存系统重启进度表，默认情况下，APC 在 CLI 任务中无法正常工作，如果你在使用 APC，需要在 APC 配置中添加 `apc.enable_cli=1`。

## 5、处理失败任务

由于事情并不总是按照计划发展，有时候你的队列任务会失败。别担心，它发生在我们大多数人身上！Lumen 包含了一个方便的方式来指定任务最大尝试执行次数，任务执行次数达到最大限制后，会被插入到 `failed_jobs` 表，失败任务的名字可以通过配置文件 `config/queue.php` 来配置。

要创建一个 failed_jobs 表的迁移，可以使用 queue:failed-table 命令：

```
php artisan queue:failed-table
```

运行队列监听器的时候，可以在 `queue:listen` 命令上使用`--tries` 开关来指定任务最大可尝试执行次数：

```
php artisan queue:listen connection-name --tries=3
```

## 5.1 失败任务事件

如果你想要注册一个队列任务失败时被调用的事件，可以使用 `Queue::failing` 方法，该事件通过邮件或 HipChat 通知团队。举个例子，我么可以在 Lumen 自带的 `AppServiceProvider` 中附件一个回调到该事件：

```
<?php

namespace App\Providers;

use Queue;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider{
    /**
     * 启动应用服务
     *
     * @return void
     */
    public function boot()
    {
        Queue::failing(function ($connection, $job, $data) {
            // Notify team of failing job...
        });
    }

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

**任务类的失败方法**

想要更加细粒度的控制，可以在队列任务类上直接定义 `failed` 方法，从而允许你在失败发生时执行指定动作：

```
<?php

namespace App\Jobs;

use App\Jobs\Job;
```

```
use Illuminate\Contracts\Mail\Mailer;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Bus\SelfHandling;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendReminderEmail extends Job implements SelfHandling, Sh
ouldQueue
{
    use InteractsWithQueue, SerializesModels;

    /**
     * 执行任务
     *
     * @param  Mailer  $mailer
     * @return void
     */
    public function handle(Mailer $mailer)
    {
        //
    }

    /**
     * 处理失败任务
     *
     * @return void
     */
    public function failed()
    {
        // Called when the job is failing...
    }
}
```

## 5.2 重试失败任务

要查看已插入到 `failed_jobs` 数据表中的所有失败任务，可以使用 Artisan 命令 `queue:failed`：

```
php artisan queue:failed
```

该命令将会列出任务 ID，连接，对列和失败时间，任务 ID 可用于重试失败任务，例如，要重试一个 ID 为 5 的失败任务，要用到下面的命令：

```
php artisan queue:retry 5
```

如果你要删除一个失败任务，可以使用 `queue:forget` 命令：

```
php artisan queue:forget 5
```

要删除所有失败任务，可以使用 `queue:flush` 命令：

```
php artisan queue:flush
```

# 服务容器

## 1、绑定

几乎所有的服务容器绑定都是在服务提供者中完成。因此本章节的演示例子用到的容器都是在这种上下文环境中，如果一个类没有基于任何接口那么就没有必要将其绑定到容器。容器并不需要被告知如何构建对象，因为它会使用 PHP 的反射服务自动解析出具体的对象。

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

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

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

**绑定一个单例**

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

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

**绑定实例**

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

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

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

### 1.1 绑定接口到实现

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

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

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

```
use App\Contracts\EventPusher;

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

## 1.2 上下文绑定

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

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

你甚至还可以传递一个闭包到 give 方法：

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

## 1.3 标签

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

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

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

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

这些服务被打上标签后，可以通过 `tagged` 方法来轻松解析它们：

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

## 2、解析

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

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

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

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

最后，也是最常用的，你可以简单的通过在类的构造函数中对依赖进行类型提示来从容器中解析对象，包括控制器、事件监听器、队列任务、中间件等都是通过这种方式。在实践中，这是大多数对象从容器中解析的方式。
容器会自动为其解析类注入依赖，比如，你可以在控制器的构造函数中为应用定义的仓库进行类型提示，该仓库会自动解析并注入该类：

```
<?php

namespace App\Http\Controllers;

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

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

    /**
     * 创建一个控制器实例
     *
     * @param  UserRepository  $users
```

```
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }

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

## 3、容器事件

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

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

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

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

# 服务提供者

## 1、简介

服务提供者是所有 Lumen 应用启动的中心，你自己的应用以及所有 Lumen 的核心服务都是通过服务提供者启动。
但是，我们所谓的”启动“指的是什么？通常，这意味着注册事物，包括注册服务容器绑定、时间监听器、中间件甚至路由。服务提供者是应用配置的中心。

如果你打开 Lumen 自带的 `bootstrap/app.php` 文件，将会看到一个`$app->register()`调用，这里就是应用所要加载的所有服务提供者类。
本章里你将会学习如何编写自己的服务提供者并在 Lumen 应用中注册它们。

# 2、编写服务提供者

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

## 2.1 register 方法

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

```
<?php

namespace App\Providers;

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

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

该服务提供者只定义了一个 `register` 方法，并使用该方法在服务容器中定义了一个 `Riak\Contracts\Connection` 的实现。如果你不太理解服务容器是怎么工作的，查看其文档。

### 2.2 boot 方法

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

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

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

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

## 3、注册服务提供者

所有服务提供者都是通过配置文件 `bootstrap/app.php` 中进行注册，该文件包含了一个 `$app->register()`方法调用，你可以将自己自定义的服务提供者放到该方法调用中来注册服务提供者。

# Session

## 1、简介

由于 HTTP 驱动的应用是无状态的，所以我们使用 session 来存储用户请求信息。Lumen 通过干净、统一的 API 处理后端各种有效 session 驱动，目前支持的流行后端驱动包括 Memcached、Redis 和数据库。

### 1.1 开启 Session

要开启 session，你需要取消 `bootstrap/app.php` 文件中`$app->middleware()`方法调用前的注释。

### 1.2 配置

session 驱动由.env 文件中的配置选项 SESSION_DRIVER 来控制。默认情况下，Lumen 使用的 `session` 驱动为 memcached，这对许多应用而言是没有什么问题的。在生产环境中，你可以考虑使用 `memcached` 或者 `redis` 驱动以便获取更快的 session 性能。

`session` 驱动定义请求的 session 数据存放在哪里，Lumen 可以处理多种类型的驱动：

- `file` – session 数据存储在 `storage/framework/sessions` 目录下；
- `cookie` – session 数据存储在经过加密的安全的 cookie 中；
- `database` – session 数据存储在数据库中
- `memcached` / `redis` – session 数据存储在 memcached/redis 中；
- `array` – session 数据存储在简单 PHP 数组中，在多个请求之间是非持久化的。

注意：数组驱动通常用于运行测试以避免 session 数据持久化。

### 1.3 Session 驱动预备知识

**数据库**

当使用 `database`session 驱动时，需要设置表包含 `session` 项，下面是该数据表的表结构声明：

```
Schema::create('sessions', function ($table) {
    $table->string('id')->unique();
    $table->text('payload');
    $table->integer('last_activity');
});
```

Redis

在 Lumen 中使用 Redis `session` 驱动前，需要通过 Composer 安装 `predis/predis` 包以及 illuminate/redis 包。

### 1.4 其它 Session 相关问题

Lumen 框架内部使用 `flash` session 键，所以你不应该通过该名称添加数据项到 session。

# 2、基本使用

**访问 session**

首先，我们来访问 session，我们可以通过 HTTP 请求访问 session 实例，可以在控制器方法中通过类型提示引入请求实例，记住，控制器方法依赖通过 Lumen 服务容器注入：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserController extends Controller{
    /**
     * 显示指定用户的属性
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function showProfile(Request $request, $id)
    {
        $value = $request->session()->get('key');

        //
    }
}
```

从 session 中获取数据的时候，还可以传递默认值作为第二个参数到 `get` 方法，默认值在指定键在 session 中不存在时返回。如果你传递一个闭包作为默认值到 `get` 方法，该闭包会执行并返回执行结果：

```
$value = $request->session()->get('key', 'default');

$value = $request->session()->get('key', function() {
    return 'default';
});
```

如果你想要从 session 中获取所有数据，可以使用 `all` 方法：

```
$data = $request->session()->all();
```

还可以使用全局的 PHP 函数 `session` 来获取和存储 session 中的数据：

```
Route::get('home', function () {
    // 从 session 中获取数据...
    $value = session('key');
```

```
    // 存储数据到 session...
    session(['key' => 'value']);
});
```

**判断 session 中是否存在指定项**

has 方法可用于检查数据项在 session 中是否存在。如果存在的话返回 true：

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

**在 session 中存储数据**

获取到 session 实例后，就可以调用多个方法来与底层数据进行交互，例如，put 方法存储新的数据到 session 中：

```
$request->session()->put('key', 'value');
```

**推送数据到数组 session**

push 方法可用于推送数据到值为数组的 session，例如，如果 user.teams 键包含团队名数组，可以像这样推送新值到该数组：

```
$request->session()->push('user.teams', 'developers');
```

**获取并删除数据**

pull 方法将会从 session 获取并删除数据：

```
$value = $request->session()->pull('key', 'default');
```

**从 session 中删除数据项**

forget 方法从 session 中移除指定数据，如果你想要从 session 中移除所有数据，可以使用 flush 方法：

```
$request->session()->forget('key');
$request->session()->flush();
```

**重新生成 Session ID**

如果你需要重新生成 session ID，可以使用 regenerate 方法：

```
$request->session()->regenerate();
```

## 2.1 一次性数据

有时候你可能想要在 session 中存储只在下个请求中有效的数据，可以通过 flash 方法来实现。使用该方法存储的 session 数据只在随后的 HTTP 请求中有效，然后将会被删除：

```
$request->session()->flash('status', 'Task was successful!');
```

如果你需要在更多请求中保持该一次性数据，可以使用 reflash 方法，该方法将所有一次性数据保留到下一个请求，如果你只是想要保存特定一次性数据，可以使用 keep 方法：

```
$request->session()->reflash();
```

```
$request->session()->keep(['username', 'email']);
```

# 测试

## 1、简介

Lumen 植根于测试，实际上，内置使用 PHPUnit 对测试提供支持是即开即用的，并且 `phpunit.xml` 文件已经为应用设置好了。框架还提供了方便的帮助方法允许你对应用进行富有表现力的测试。
`tests` 目录中提供了一个 `ExampleTest.php` 文件，安装完新的 Lumen 应用后，只需简单在命令行运行 `phpunit` 来运行测试。

### 1.1 测试环境

Lumen 在测试时自动配置 session 和 cache 驱动为数组驱动，这意味着测试时不会持久化存储 session 和 cache。

如果需要的话你可以自由创建其它测试环境配置。`testing` 环境变量可以在 `phpunit.xml` 文件中配置。

### 1.2 定义&运行测试

要创建一个测试用例，只需简单在 `tests` 目录创建一个新的测试文件，测试类应该继承 `TestCase`，然后你可以使用 PHPUnit 定义测试方法。要运行测试，简单从终端执行 `phpunit` 命令即可：

```
<?php

class FooTest extends TestCase{
    public function testSomethingIsTrue()
    {
        $this->assertTrue(true);
    }
}
```

注意：如果你在测试类中定义自己的 `setUp` 方法，确保在其中调用 `parent::setUp`。

## 2、应用测试

Lumen 为生成 HTTP 请求、测试输出、以及填充表单提供了平滑的 API。举个例子，我们看下 `tests` 目录下包含的 `ExampleTest.php` 文件：

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
```

```
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase{
    /**
     * 基本功能测试示例
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->visit('/')
             ->see('Laravel 5')
             ->dontSee('Rails');
    }
}
```

`visit` 方法生成了一个 GET 请求，`see` 方法对我们从应用返回响应中应该看到的给定文本进行断言。`dontSee` 方法对给定文本没有从应用响应中返回进行断言。在 Lumen 中这是最基本的有效应用测试。

## 2.1 与应用交互

当然，除了对响应文本进行断言之外还有做更多测试，让我们看一些点击链接和填充表单的例子：

**点击链接**

在本测试中，我们将为应用生成请求，在返回的响应中“点击”链接，然后对访问 URI 进行断言。例如，假定响应中有一个“关于我们”的链接：

```
<a href="/about-us">About Us</a>
```

现在，让我们编写一个测试点击链接并断言用户访问页面是否正确：

```
public function testBasicExample(){
    $this->visit('/')
         ->click('About Us')
         ->seePageIs('/about-us');
}
```

**处理表单**

Lumen 还为处理表单提供了多个方法。`type`, `select`, `check`, `attach`, 和 `press` 方法允许你与所有表单输入进行交互。例如，我们假设这个表单存在于应用注册页面：

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

    <div>
```

```
        Name: <input type="text" name="name">
    </div>

    <div>
        <input type="checkbox" value="yes" name="terms"> Accept
Terms
    </div>

    <div>
        <input type="submit" value="Register">
    </div>
</form>
```

我们可以编写测试完成表单并检查结果：

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

当然，如果你的表单包含其他输入比如单选按钮或下拉列表，也可以轻松填写这些字段类型。这里是所有表单操作方法列表：

| 方法 | 描述 |
| --- | --- |
| `$this->type($text, $elementName)` | “Type” 文本到给定字段 |
| `$this->select($value, $elementName)` | “Select” 单选框或下拉列表 |
| `$this->check($elementName)` | “Check” 复选框 |
| `$this->attach($pathToFile, $elementName)` | “Attach” 文件到表单 |
| `$this->press($buttonTextOrElementName)` | “Press” 给定文本或 name 的按钮 |

**处理附件**

如果表单包含 `file` 输入类型，可以使用 `attach` 方法添加文件到表单：

```
public function testPhotoCanBeUploaded(){
    $this->visit('/upload')
         ->name('File Name', 'name')
         ->attach($absolutePathToFile, 'photo')
         ->press('Upload')
```

```
        ->see('Upload Successful!');
}
```

## 2.2 测试 JSON API

Lumen 还提供多个帮助函数用于测试 JSON API 及其响应。例如，`get`, `post`, `put`, `patch`, 和 `delete` 方法用于通过多种 HTTP 请求方式发出请求。你还可以轻松传递数据和头到这些方法。作为开始，我们编写测试来生成 POST 请求到`/user`并断言返回的数据是否是 JSON 格式：

```
<?php

class ExampleTest extends TestCase{
    /**
     * 基本功能测试示例
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->post('/user', ['name' => 'Sally'])
             ->seeJson([
                 'created' => true,
             ]);
    }
}
```

`seeJson` 方法将给定数组转化为 JSON，然后验证应用返回的整个 JSON 响应中的 JSON 片段。因此，如果在 JSON 响应中有其他属性，只要给定片段存在的话测试依然会通过。

**精确验证 JSON 匹配**

如果你想要验证给定数组和应用返回的 JSON 能够精确匹配，使用 `seeJsonEquals` 方法：

```
<?php

class ExampleTest extends TestCase{
    /**
     * 基本功能测试示例
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->post('/user', ['name' => 'Sally'])
             ->seeJsonEquals([
                 'created' => true,
             ]);
```

```
    }
}
```

## 2.3 Session/认证

Lumen 提供个多个帮助函数在测试期间处理 session，首先，可以使用 `withSession` 方法设置 session 值到给定数组。这通常在测试请求前获取 session 数据时很有用：

```
<?php

class ExampleTest extends TestCase{
    public function testApplication()
    {
        $this->withSession(['foo' => 'bar'])
             ->visit('/');
    }
}
```

当然，session 的通常用于操作用户状态，例如认证用户。帮助函数 `actingAs` 提供了认证给定用户为当前用户的简单方法，例如，我们使用模型工厂生成和认证用户：

```
<?php

class ExampleTest extends TestCase{
    public function testApplication()
    {
        $user = factory('App\User')->create();

        $this->actingAs($user)
             ->withSession(['foo' => 'bar'])
             ->visit('/')
             ->see('Hello, '.$user->name);
    }
}
```

## 2.4 自定义 HTTP 请求

如果你想要在应用中生成自定义 HTTP 请求并获取完整的 `Illuminate\Http\Response` 对象，可以使用 `call` 方法：

```
public function testApplication(){
    $response = $this->call('GET', '/');
    $this->assertEquals(200, $response->status());
}
```

如果你要生成 `POST`, `PUT`, 或者 `PATCH` 请求可以在请求中传入输入数据数组，在路由或控制器中可以通过 Request 实例访问请求数据：

```
$response = $this->call('POST', '/user', ['name' => 'Taylor']);
```

# 3、处理数据库

Lumen 还提供了多种有用的工具让测试数据库驱动的应用更加简单。首先，你可以使用帮助函数 `seeInDatabase` 来断言数据库中的数据是否和给定数据集合匹配。例如，如果你想要通过 email 值为 `sally@example.com` 的条件去数据表 `users` 查询是否存在该记录 ，我们可以这样做：

```
public function testDatabase(){
    // 调用应用...
    $this->seeInDatabase('users', ['email' => 'sally@foo.com
']);
}
```

当然，`seeInDatabase` 方法和其他类似帮助函数都是为了方便，你还可以使用所有 PHPUnit 内置的断言函数来补充测试。

## 3.1 每次测试后重置数据库

每次测试后重置数据库通常很有用，这样的话上次测试的数据不会影响下一次测试。

**使用迁移**

一种方式是每次测试后回滚数据库并在下次测试前重新迁移。Lumen 提供了一个简单的 `DatabaseMigrations` trait 来自动为你处理。在测试类上简单使用该 trait 如下：

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase{
    use DatabaseMigrations;

    /**
     * 基本功能测试示例
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->visit('/')
             ->see('Lumen');
    }
}
```

**使用事务**

另一种方式是将每一个测试用例包裹到一个数据库事务中，Lumen 提供了方便的 `DatabaseTransactions` trait 自动为你处理：

```
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase{
    use DatabaseTransactions;

    /**
     * 基本功能测试示例
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->visit('/')
             ->see('Lumen');
    }
}
```

## 3.2 模型工厂

测试时，通常需要在执行测试前插入新数据到数据库。在创建测试数据时，Lumen 允许你使用”factories”为每个 Eloquent 模型定义默认的属性值集合，而不用手动为每一列指定值。作为开始，我们看一下 `atabase/factories/ModelFactory.php` 文件，该文件包含了一个工厂定义：

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

在闭包中，作为工厂定义，我们返回该模型上所有属性默认测试值。该闭包接收 PHP 库 Faker 实例，从而允许你方便地为测试生成多种类型的随机数据。

当然，你可以添加更多工厂到 `ModelFactory.php` 文件。

**多个工厂类型**

有时候你可能想要为同一个 Eloquent 模型类生成多个工厂，例如，除了正常用户外可能你想要为“管理员”用户生成一个工厂，你可以使用 `defineAs` 方法定义这些工厂：

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

你可以使用 `raw` 方法获取基本属性而不用重复基本用户工厂中的所有属性，获取这些属性后，只需将你要求的额外值增补进去即可：

```
$factory->defineAs(App\User::class, 'admin', function ($faker)
use ($factory) {
    $user = $factory->raw(App\User::class);
    return array_merge($user, ['admin' => true]);
});
```

**测试中使用工厂**

定义好工厂后，可以在测试或数据库填充文件中通过全局的 `factory` 方法使用它们来生成模型实例，所以，让我们看一些生成模型的例子，首先，我们使用 `make` 方法，该方法创建模型但不将其保存到数据库：

```
public function testDatabase(){
    $user = factory(App\User::class)->make();
    // 用户模型测试...
}
```

如果你想要覆盖模型的一些默认值，可以传递数组值到 `make` 方法。只有指定值被替换，其他数据保持不变：

```
$user = factory(App\User::class)->make([
    'name' => 'Abigail',
]);
```

还可以创建多个模型集合或者创建给定类型的集合：

```
// 创建 3 个 App\User 实例...
$users = factory(App\User::class, 3)->make();
// 创建 1 个 App\User "admin" 实例...
$user = factory(App\User::class, 'admin')->make();
// 创建 3 个 App\User "admin" 实例...
$users = factory(App\User::class, 'admin', 3)->make();
```

**持久化工厂模型**

`create` 方法不仅能创建模型实例，还可以使用 Eloquent 的 `save` 方法将它们保存到数据库：

```
public function testDatabase(){
    $user = factory(App\User::class)->create();
    //用户模型测试...
}
```

你仍然可以通过传递数组到 create 方法覆盖模型上的属性：

```
$user = factory(App\User::class)->create([
    'name' => 'Abigail',
]);
```

**添加关联关系到模型**

你甚至可以持久化多个模型到数据库。在本例中，我们添加一个关联到创建的模型，使用 `create` 方法创建多个模型的时候，会返回一个 Eloquent 集合实例，从而允许你使用集合提供的所有便利方法，例如 `each`：

```
$users = factory(App\User::class, 3)
           ->create()
           ->each(function($u) {
                $u->posts()->save(factory(App\Post::class)->make
());
            });
```

# 4、模拟

## 4.1 模拟事件

如果你在重度使用 Lumen 的事件系统，可能想要在测试时模拟特定事件。例如，如果你在测试用户注册，你可能不想所有 `UserRegistered` 的时间处理器都被触发，因为这可能会发送欢迎邮件，等等。

Lumen 提供可一个方便的 `expectsEvents` 方法来验证期望的事件被触发，但同时阻止该事件的其它处理器运行：

```
<?php

class ExampleTest extends TestCase{
    public function testUserRegistration()
    {
        $this->expectsEvents(App\Events\UserRegistered::class);
        // 测试用户注册代码...
    }
}
```

如果你想要阻止所有事件运行，可以使用 `withoutEvents` 方法：

```
<?php

class ExampleTest extends TestCase{
    public function testUserRegistration()
    {
        $this->withoutEvents();
        // 测试用户注册代码...
    }
}
```

## 4.2 模拟队列任务

有时候，你可能想要在请求时简单测试控制器分发的指定任务，这允许你孤立的测试路由/控制器——将其从任务逻辑中分离出去，当然，接下来你可以在一个独立测试类中测试任务本身。

Lumen 提供了一个方便的 `expectsJobs` 方法来验证期望的任务被分发，但该任务本身不会被测试：

```
<?php

class ExampleTest extends TestCase{
    public function testPurchasePodcast()
    {
        $this->expectsJobs(App\Jobs\PurchasePodcast::class);
        // 测试购买播客代码...
    }
}
```

注意：这个方法只检查通过 `DispatchesJobs` trait 分发方法分发的任务，并不检查直接通过 `Queue::push` 分发的任务。

## 4.3 模拟门面

测试的时候，你可能经常想要模拟 Lumen 门面的调用，例如，看看下面的控制器动作：

```
<?php

namespace App\Http\Controllers;

use Cache;
use Illuminate\Routing\Controller;

class UserController extends Controller{
    /**
     * 显示应用用户列表
     *
```

```
     * @return Response
     */
    public function index()
    {
        $value = Cache::get('key');

        //
    }
}
```

我们可以通过使用 `shouldReceive` 方法模拟 `Cache` 门面的调用，该方法返回一个 Mockery 模拟的实例，由于门面通过 Lumen 服务容器解析和管理，它们比通常的静态类更具有可测试性。例如，我们来模拟 `Cache` 门面的调用：

```
<?php

class FooTest extends TestCase{
    public function testGetIndex()
    {
        Cache::shouldReceive('get')
                    ->once()
                    ->with('key')
                    ->andReturn('value');

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

注意：不要模拟 `Request` 门面，取而代之地，在测试时传递输入到 HTTP 帮助函数如 `call` 和 `post`。

# 验证

## 1、简介

Lumen 提供了多种方法来验证应用输入数据。默认情况下，Lumen 的控制器基类使用 `ValidatesRequests` trait，该 trait 提供了便利的方法通过各种功能强大的验证规则来验证输入的 HTTP 请求。

## 2、快速入门

要学习 Lumen 强大的验证特性，让我们先看一个完整的验证表单并返回错误信息给用户的例子。

## 2.1 定义路由

首先，我们假定在 `app/Http/routes.php` 文件中包含如下路由：

```
// 显示创建博客文章表单...
Route::get('post/create', 'PostController@create');
// 存储新的博客文章...
Route::post('post', 'PostController@store');
```

当然，GET 路由为用户显示了一个创建新的博客文章的表单，POST 路由将新的博客文章存储到数据库。

## 2.2 创建控制器

接下来，让我们看一个处理这些路由的简单控制器示例。我们先将 `store` 方法留空：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PostController extends Controller{
    /**
     * 显示创建新的博客文章的表单
     *
     * @return Response
     */
    public function create()
    {
        return view('post.create');
    }

    /**
     * 存储新的博客文章
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        // 验证并存储博客文章...
    }
}
```

## 2.3 编写验证逻辑

现在我们准备用验证新博客文章输入的逻辑填充 `store` 方法。如果你检查应用的控制器基类（`App\Http\Controllers\Controller`），你会发现该类使用了 `ValidatesRequests` trait，这个 trait 在所有控制器中提供了一个便利的 `validate` 方法。`validate` 方法接收一个 HTTP 请求输入数据和验证规则，如果验证规则通过，代码将会继续往下执行；然而，如果验证失败，将会抛出一个异常，相应的错误响应也会自动发送给用户。在一个传统的 HTTP 请求案例中，将会生成一个重定向响应，如果是 AJAX 请求则会返回一个 JSON 响应。

要更好的理解 `validate` 方法，让我们回到 `store` 方法：

```
/**
 * 存储博客文章
 *
 * @param  Request  $request
 * @return Response
 */
public function store(Request $request){
    $this->validate($request, [
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ]);

    // 验证通过，存储到数据库...
}
```

正如你所看到的，我们只是传递输入的 HTTP 请求和期望的验证规则到 `validate` 方法，在强调一次，如果验证失败，相应的响应会自动生成。如果验证通过，控制器将会继续正常执行。

**嵌套属性注意事项**

如果 HTTP 请求中包含“嵌套”参数，可以使用“.”在验证规则中指定它们：

```
$this->validate($request, [
    'title' => 'required|unique:posts|max:255',
    'author.name' => 'required',
    'author.description' => 'required',
]);
```

## 2.4 显示验证错误信息

那么，如果请求输入参数没有通过给定验证规则怎么办？正如前面所提到的，Lumen 将会自动将用户重定向回上一个位置。此外，所有验证错误信息会自动一次性存放到 session。注意我们并没有在 GET 路由中明确绑定错误信息到视图。这是因为 Lumen 总是从 session 数据中检查错误信息，而且如果有的话会自动将其绑定到视图。所以，值得注意的是每次请求的所有视图中总是存在一个`$errors` 变量，从而允许你在视图中方便而又安

全地使用。`$errors` 变量是的一个 `Illuminate\Support\MessageBag` 实例。想要了解更多关于该对象的信息，查看其文档。
所以，在我们的例子中，验证失败的话用户将会被重定向到控制器的 create 方法，从而允许我们在视图中显示错误信息：

```
<!-- /resources/views/post/create.blade.php -->

<h1>Create Post</h1>

@if (count($errors) > 0)
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

<!-- Create Post Form -->
```

**自定义错误格式**

如果你想要自定义验证失败时被一次性存到 session 中的错误信息的格式，重写控制器基类中的 `formatValidationErrors` 方法，不要忘记在文件顶部导入 `Illuminate\Contracts\Validation\Validator` 类：

```
<?php

namespace App\Http\Controllers;

use Illuminate\Contracts\Validation\Validator;
use Laravel\Lumen\Routing\Controller as BaseController;

abstract class Controller extends BaseController{
    /**
     * {@inheritdoc}
     */
    protected function formatValidationErrors(Validator $valida
tor)
    {
        return $validator->errors()->all();
    }
}
```

### 2.5 AJAX 请求&验证

在这个例子中，我们使用传统的表单来发送数据到应用。然而，很多应用使用 AJAX 请求。在 AJAX 请求中使用 `validate` 方法时，Lumen 不会生成重定向响应。取而代之的，Lumen 生成一个包含验证错误信息的 JSON 响应。该 JSON 响应会带上一个 HTTP 状态码 `422`。

## 3、其它验证方法

### 3.1 手动创建验证器

如果你不想使用 `ValidatesRequests` trait 的 `validate` 方法，可以使用 `Validator` 门面手动创建一个验证器实例，该门面上的 `make` 方法用于生成一个新的验证器实例：

```
<?php

namespace App\Http\Controllers;

use Validator;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PostController extends Controller{
    /**
     * 存储新的博客文章
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        $validator = Validator::make($request->all(), [
            'title' => 'required|unique:posts|max:255',
            'body' => 'required',
        ]);

        if ($validator->fails()) {
            return redirect('post/create')
                        ->withErrors($validator)
                        ->withInput();
        }

        // 存储博客文章...
    }
```

```
}
```

传递给 `make` 方法的第一个参数是需要验证的数据，第二个参数是要应用到数据上的验证规则。

检查请求是够通过验证后，可以使用 `withErrors` 方法将错误数据一次性存放到 session，使用该方法时，`$errors` 变量重定向后自动在视图间共享，从而允许你轻松将其显示给用户，`withErrors` 方法接收一个验证器、或者一个 MessageBag，又或者一个 PHP 数组。

**命名错误包**

如果你在单个页面上有多个表单，可能需要命名 MessageBag，从而允许你为指定表单获取错误信息。只需要传递名称作为第二个参数给 `withErrors` 即可：

```
return redirect('register')
            ->withErrors($validator, 'login');
```

然后你就可以从`$errors` 变量中访问命名的 MessageBag 实例：

```
{{ $errors->login->first('email') }}
```

**验证钩子之后**

验证器允许你在验证完成后添加回调，这种机制允许你轻松执行更多验证，甚至添加更多错误信息到消息集合。使用验证器实例上的 `after` 方法即可：

```
$validator = Validator::make(...);

$validator->after(function($validator) {
    if ($this->somethingElseIsInvalid()) {
        $validator->errors()->add('field', 'Something is wrong w
ith this field!');
    }
});

if ($validator->fails()) {
    //
}
```

# 4、处理错误信息

调用 Validator 实例上的 `errors` 方法之后，将会获取一个 `Illuminate\Support\MessageBag` 实例，该实例中包含了多种处理错误信息的便利方法。

**获取某字段的第一条错误信息**

要获取指定字段的第一条错误信息，可以使用 `first` 方法：

```
$messages = $validator->errors();
echo $messages->first('email');
```

**获取指定字段的所有错误信息**

如果你想要简单获取指定字段的所有错误信息数组，使用 `get` 方法：

```
foreach ($messages->get('email') as $message) {
    //
}
```

**获取所有字段的所有错误信息**

要获取所有字段的所有错误信息，可以使用 `all` 方法：

```
foreach ($messages->all() as $message) {
    //
}
```

**判断消息中是否存在某字段的错误信息**

```
if ($messages->has('email')) {
    //
}
```

**获取指定格式的错误信息**

```
echo $messages->first('email', '<p>:message</p>');
```

**获取指定格式的所有错误信息**

```
foreach ($messages->all('<li>:message</li>') as $message) {
    //
}
```

## 4.1 自定义错误信息

如果需要的话，你可以使用自定义错误信息替代默认的，有多种方法来指定自定义信息。

首先，你可以传递自定义信息作为第三方参数给 `Validator::make` 方法：

```
$messages = [
    'required' => 'The :attribute field is required.',
];

$validator = Validator::make($input, $rules, $messages);
```

在本例中，`:attribute` 占位符将会被验证时实际的字段名替换，你还可以在验证消息中使用其他占位符，例如：

```
$messages = [
    'same'    => 'The :attribute and :other must match.',
    'size'    => 'The :attribute must be exactly :size.',
    'between' => 'The :attribute must be between :min - :max.',
    'in'      => 'The :attribute must be one of the following ty
pes: :values',
];
```

**为给定属性指定自定义信息**

有时候你可能只想为特定字段指定自定义错误信息，可以通过”.”来实现，首先指定属性名，然后是规则：

```
$messages = [
    'email.required' => 'We need to know your e-mail address!',
];
```

**在语言文件中指定自定义消息**

在很多案例中，你可能想要在语言文件中指定属性特定自定义消息而不是将它们直接传递给 `Validator`。要实现这个，添加消息到 `resources/lang/xx/validation.php` 语言文件的 custom 数组：

```
'custom' => [
    'email' => [
        'required' => 'We need to know your e-mail address!',
    ],
],
```

# 5、有效验证规则

下面是有效规则及其函数列表：

- Accepted
- Active URL
- After (Date)
- Alpha
- Alpha Dash
- Alpha Numeric
- Array
- Before (Date)
- Between
- Boolean
- Confirmed
- Date
- Date Format
- Different
- Digits
- Digits Between
- E-Mail
- Exists (Database)
- Image (File)
- In
- Integer
- IP Address
- Max
- MIME Types (File)

- Min
- Not In
- Numeric
- Regular Expression
- Required
- Required If
- Required With
- Required With All
- Required Without
- Required Without All
- Same
- Size
- String
- Timezone
- Unique (Database)
- URL

## accepted

在验证中该字段的值必须是 `yes`、`on`、`1` 或 `true`，这在“同意服务协议”时很有用。

## active_url

该字段必须是一个基于 PHP 函数 `checkdnsrr` 的有效 URL

## after:date

该字段必须是给定日期后的一个值，日期将会通过 PHP 函数 `strtotime` 传递：

```
'start_date' => 'required|date|after:tomorrow'
```

你可以指定另外一个比较字段而不是使用 strtotime 验证传递的日期字符串：

```
'finish_date' => 'required|date|after:start_date'
```

## alpha

该字段必须是字母

## alpha_dash

该字段可以包含字母和数字，以及破折号和下划线

## alpha_num

该字段必须是字母或数字

## array

该字段必须是 PHP 数组

## before:date

验证字段必须是指定日期之前的一个数值，该日期将会传递给 PHP `strtotime` 函数。

## between:min,max

验证字段尺寸在给定的最小值和最大值之间，字符串、数值和文件都可以使用该规则

## boolean

验证字段必须可以被转化为 `boolean`，接收 `true`, `false`, `1`,`0`, `"1"`，和 `"0"`等输入。

## confirmed

验证字段必须有一个匹配字段 `foo_confirmation`，例如，如果验证字段是 `password`，必须输入一个与之匹配的 `password_confirmation` 字段

## date

验证字段必须是一个基于 PHP `strtotime` 函数的有效日期

## date_format:format

验证字段必须匹配指定格式，该格式将使用 PHP 函数 `date_parse_from_format` 进行验证。你应该在验证字段时使用 `date` 或 `date_format`

## different:field

验证字段必须是一个和指定字段不同的值

## digits:value

验证字段必须是数字且长度为 `value` 指定的值

## digits_between:min,max

验证字段数值长度必须介于最小值和最大值之间

## email

验证字段必须是格式化的电子邮件地址

## exists:table.column

验证字段必须存在于指定数据表

基本使用：

```
'state' => 'exists:states'
```

指定自定义列名：

```
'state' => 'exists:states,abbreviation'
```

还可以添加更多查询条件到 `where` 查询子句：

```
'email' => 'exists:staff,email,account_id,1'
```

传递 NULL 作为 `where` 子句的值将会判断数据库值是否为 NULL：

```
'email' => 'exists:staff,email,deleted_at,NULL'
```

## image

验证文件必须是图片（jpeg、png、bmp、gif 或者 svg）

## in:foo,bar…

验证字段值必须在给定的列表中

## integer

验证字段必须是整型

## ip

验证字段必须是 IP 地址

## max:value

验证字段必须小于等于最大值，和字符串、数值、文件字段的 size 规则一起使用

## mimes:foo,bar,…

验证文件的 MIMIE 类型必须是该规则列出的扩展类型中的一个

MIMIE 规则的基本使用：

```
'photo' => 'mimes:jpeg,bmp,png'
```

## min:value

验证字段的最小值，和字符串、数值、文件字段的 size 规则一起使用

## not_in:foo,bar,…

验证字段值不在给定列表中

### numeric

验证字段必须是数值

### regex:pattern

验证字段必须匹配给定正则表达式

注意：使用 `regex` 模式时，规则必须放在数组中，而不能使用管道分隔符，尤其是正则表达式中使用管道符号时。

### required

验证字段时必须的

### required_if:anotherfield,value,…

验证字段在另一个字段等于指定值 value 时是必须的

### required_with:foo,bar,…

验证字段只有在任一其它指定字段存在的话才是必须的

### required_with_all:foo,bar,…

验证字段只有在所有指定字段存在的情况下才是必须的

### required_without:foo,bar,…

验证字段只有当任一指定字段不存在的情况下才是必须的

### required_without_all:foo,bar,…

验证字段只有当所有指定字段不存在的情况下才是必须的

### same:field

给定字段和验证字段必须匹配

### size:value

验证字段必须有和给定值相 value 匹配的尺寸，对字符串而言，`value` 是相应的字符数目；对数值而言，`value` 是给定整型值；对文件而言，`value` 是相应的文件字节数

### string

验证字段必须是字符串

### timezone

验证字符必须是基于 PHP 函数 `timezone_identifiers_list` 的有效时区标识

### unique:table,column,except,idColumn

验证字段在给定数据表上必须是唯一的，如果不指定 `column` 选项，字段名将作为默认 `column`。

**指定自定义列名：**

```
'email' => 'unique:users,email_address'
```

**自定义数据库连接**

有时候，你可能需要自定义验证器生成的数据库连接，正如上面所看到的，设置 `unique:users` 作为验证规则将会使用默认数据库连接来查询数据库。要覆盖默认连接，在数据表名后使用”.“指定连接：

```
'email' => 'unique:connection.users,email_address'
```

**强制一个唯一规则来忽略给定 ID：**

有时候，你可能希望在唯一检查时忽略给定 ID，例如，考虑一个包含用户名、邮箱地址和位置的”更新属性“界面，当然，你将会验证邮箱地址是唯一的，然而，如果用户只改变用户名字段而并没有改变邮箱字段，你不想要因为用户已经拥有该邮箱地址而抛出验证错误，你只想要在用户提供的邮箱已经被别人使用的情况下才抛出验证错误，要告诉唯一规则忽略用户 ID，可以传递 ID 作为第三个参数：

```
'email' => 'unique:users,email_address,'.$user->id
```

**添加额外的 `where` 子句：**

还可以指定更多条件给 `where` 子句：

```
'email' => 'unique:users,email_address,NULL,id,account_id,1'
```

### url

验证字段必须是基于 PHP 函数 `filter_var` 过滤的的有效 URL

## 6、添加条件规则

在某些场景下，你可能想要只有某个字段存在的情况下运行验证检查，要快速完成这个，添加 `sometimes` 规则到规则列表：

```
$v = Validator::make($data, [
    'email' => 'sometimes|required|email',
]);
```

在上例中，email 字段只有存在于`$data` 数组时才会被验证。

**复杂条件验证**

有时候你可能想要基于更复杂的条件逻辑添加验证规则。例如，你可能想要只有在另一个字段值大于 `100` 时才要求一个给定字段是必须的，或者，你可能需要只有当另一个字段存在时两个字段才都有给定值。添加这个验证规则并不是一件头疼的事。首先，创建一个永远不会改变的静态规则到 Validator 实例：

```
$v = Validator::make($data, [
    'email' => 'required|email',
    'games' => 'required|numeric',
]);
```

让我们假定我们的 web 应用服务于游戏收集者。如果一个游戏收集者注册了我们的应用并拥有超过 `100` 个游戏，我们想要他们解释为什么他们会有这么多游戏，例如，也许他们在运营一个游戏二手店，又或者他们只是喜欢收集。要添加这种条件，我们可以使用 Validator 实例上的 `sometimes` 方法：

```
$v->sometimes('reason', 'required|max:500', function($input) {
    return $input->games >= 100;
});
```

传递给 `sometimes` 方法的第一个参数是我们需要有条件验证的名称字段，第二个参数是我们想要添加的规则，如果作为第三个参数的闭包返回 `true`，规则被添加。该方法让构建复杂条件验证变得简单，你甚至可以一次为多个字段添加条件验证：

```
$v->sometimes(['reason', 'cost'], 'required', function($input)
{
    return $input->games >= 100;
});
```

注意：传递给闭包的`$input` 参数是 `Illuminate\Support\Fluent` 的一个实例，可用于访问输入和文件。

# 7、自定义验证规则

Lumen 提供了多种有用的验证规则；然而，你可能还是想要指定一些自己的验证规则。注册验证规则的一种方法是使用 `Validator` 门面的 extend 方法。让我们在服务提供者中使用这种方法来注册一个自定义的验证规则：

```
<?php

namespace App\Providers;

use Validator;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider{
    /**
     * 启动应用服务
```

```
     *
     * @return void
     */
    public function boot()
    {
        Validator::extend('foo', function($attribute, $value,
$parameters) {
            return $value == 'foo';
        });
    }

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

自定义验证器闭包接收三个参数：要验证的属性名称，属性值和传递给规则的参数数组。

你还可以传递类和方法到 `extend` 方法而不是闭包：

```
Validator::extend('foo', 'FooValidator@validate');
```

**定义错误信息**

你还需要为自定义规则定义错误信息。你可以使用内联自定义消息数组或者在验证语言文件中添加条目来实现这一目的。消息应该被放到数组的第一维，而不是在只用于存放属性指定错误信息的 custom 数组内：

```
"foo" => "Your input was invalid!",
"accepted" => "The :attribute must be accepted.",
// 验证错误信息其它部分...
```

当创建一个自定义验证规则时，你可能有时候需要为错误信息定义自定义占位符，可以通过创建自定义验证器然后调用 `Validator` 门面上的 `replacer` 方法来实现。可以在服务提供者的 `boot` 方法中编写代码：

```
/**
 * 启动应用服务
 *
 * @return void
 */
public function boot(){
```

```
    Validator::extend(...);
    Validator::replacer('foo', function($message, $attribute,
$rule, $parameters) {
        return str_replace(...);
    });
}
```