Laravel5.2 中文手册

前言

欢迎阅读 Laravel5.2文档。这份文档既可以作为入门指南，也可以作为 Laravel 功能特色的参考手册。如果你迫不及待想要开始 Laravel 之旅的话，可以直接跳到你想看的章节，不过，我们还是强烈建议你按顺序阅读这份文档，这样能让你循序渐进的打好基础，而且，每一章节都是后后续章节的基础。

使用 Laravel 将是一种享受!

Laravel 是一套功能强大的 PHP 开发框架，并且着重于灵活性和语法的表现力。对于初学者，Laravel 像其他流行、轻量级框架一样易学、易用；对于经验丰富的同学，Laravel 能让你体验代码模块化的好处。Laravel 的灵活性能让你随心所欲的更新、重构你用应用；Laravel 富于表现力的语法能让你或你的团队的代码简洁、容易阅读。

Laravel 优于其它 PHP 框架

Laravel 在很多方面优于其它框架，以下列出的就是其中的一部分：

- **Bundle** 是 Laravel 的扩展包组织形式或称呼。Laravel 的扩展包仓库已经相当成熟了，可以很容易的帮你把扩展包（bundle）安装到你的应用中。你可以选择下载一个扩展包（bundle）然后拷贝到 bundles 目录，或者通过命令行工具“Artisan”自动安装。
- 在 Laravel 中已经具有了一套高级的 PHP ActiveRecord 实现 -- **Eloquent ORM**。它能方便的将“约束（constraints）”应用到关系的双方，这样你就具有了对数据的完全控制，而且享受到 ActiveRecord 的所有便利。Eloquent 原生支持 Fluent 中查询构造器（query-builder）的所有方法。
- **应用逻辑（Application Logic）**可以在控制器（controllers）中实现，也可以直接集成到路由（route）声明中，并且语法和 Sinatra 框架类似。Laravel 的设计理念是：给开发者以最大的灵活性，既能创建非常小的网站也能构建大型的企业应用。
- **反向路由（Reverse Routing）**赋予你通过路由（routes）名称创建链接（URI)的能力。只需使用路由名称（route name），Laravel 就会自动帮你创建正确的 URI。这样你就可以随时改变你的路由（routes），Laravel 会帮你自动更新所有相关的链接。
- **Restful 控制器（Restful Controllers）**是一项区分 GET 和 POST 请求逻辑的可选方式。比如在一个用户登陆逻辑中，你声明了一个 get_login()的动作（action）来处理获取登陆页面的服务；同时也声明了一个 post_login()动作（action）来校验表单 POST 过来的数据，并且在验证之后，做出重新

转向（redirect）到登陆页面还是转向控制台的决定。

- **自动加载类（Class Auto-loading）**简化了类（class）的加载工作，以后就可以不用去维护自动加载配置表和非必须的组件加载工作了。当你想加载任何库（library）或模型（model）时，立即使用就行了，Laravel 会自动帮你加载需要的文件。
- **视图组装器（View Composers）**本质上就是一段代码，这段代码在视图（View）加载时会自动执行。最好的例子就是博客中的侧边随机文章推荐，“视图组装器”中包含了加载随机文章推荐的逻辑，这样，你只需要加载内容区域的视图（view）就行了，其它的事情 Laravel 会帮你自动完成。
- **反向控制容器（IoC container）**提供了生成新对象、随时实例化对象、访问单例（singleton）对象的便捷方式。反向控制（IoC）意味着你几乎不需要特意去加载外部的库（libraries），就可以在代码中的任意位置访问这些对象，并且不需要忍受繁杂、冗余的代码结构。
- **迁移（Migrations）**就像是版本控制（version control）工具，不过，它管理的是数据库范式，并且直接集成在了 Laravel 中。你可以使用“Artisan”命令行工具生成、执行“迁移”指令。当你的小组成员改变了数据库范式的时候，你就可以轻松的通过版本控制工具更新当前工程，然后执行“迁移"指令即可，好了，你的数据库已经是最新的了！
- **单元测试（Unit-Testing）**是 Laravel 中很重要的部分。Laravel 自身就包含数以百计的测试用例，以保障任何一处的修改不会影响其它部分的功能，这就是为什么在业内 Laravel 被认为是最稳版本的原因之一。Laravel 也提供了方便的功能，让你自己的代码容易的进行单元测试。通过 Artisan 命令行工具就可以运行所有的测试用例。
- **自动分页（Automatic Pagination）**功能避免了在你的业务逻辑中混入大量无关分页配置代码。方便的是不需要记住当前页，只要从数据库中获取总的条目数量，然后使用 limit/offset 获取选定的数据，最后调用‘paginate’方法，让 Laravel 将各页链接输出到指定的视图（View)中即可，Laravel 会替你自动完成所有工作。Laravel 的自动分页系统被设计为容易实现、易于修改。虽然 Laravel 可以自动处理这些工作，但是不要忘了调用相应方法和手动配置分页系统哦！

上面提到的只是 Laravel 优于其它框架的几点。在这份文档中包含了 Laravel 所有的特性和更多优点。

安装与设置

目录

- 要求
- 安装
- 服务器配置
- 基本设置
- 环境设置
- 友好的链接

安装要求

- Apache, nginx, 或者其他 web 服务器。
- Laravel 框架应用了很多 PHP 5.3版才具备的强大的新特性，所以你必须安装 PHP5.3或者以上版本。
- Laravel 使用 FileInfo 库来检测 mime 类型。PHP 5.3版已经默认包含了 FileInfo 库。Windows 用户需要在 php.ini 中启用该模块。关于 FileInfo 库的更多信息请阅读：installation / configuration details on PHP.net。
- Laravel 使用 Mcrypt 库 来加密和生成哈希。PHP 5.3已经预装了 Mcrypt 库。如果你在 phpinfo() 中没有找到 Mcrypt 已经启用的信息，请检查你的服务器环境是否安装完全，或者查看 PHP 手册中 Mcrypt 库有关信息。

安装

1. 下载 Laravel
2. 解压 Laravel 压缩包，然后上传文件到你的 web 服务器。
3. 在 config/application.php 中设置 application key，你可以设置为任意的32位字符串。
4. 确保 `storage/views` 目录具有写入权限。
5. 现在你可以尝试在浏览器中运行框架。

如果不出意外，你应该看到了 Laravel 漂亮的初始页面。一切准备就绪，我们可以继续 Laravel 学习之旅!

选装程序

如果你想充分了解和学习 Laravel 框架的应用，推荐你安装以下程序：

- SQLite, MySQL, PostgreSQL, 或者 SQL Server PDO driver.
- Memcached 或者 APC.

安装遇到问题？

如果你在安装过程中遇到了问题，可以检查以下情况：

- 请确保 **public** 文件夹是服务器的根目录，如果不是，你可以尝试访问 Laravel 的 public 文件夹，如 http:localhost/public/。
- 如果你启用了 mod_rewrite 拓展，请把 **application/config/application.php** 文件中的 **index** 参数设置为空。
- 请确保服务器的 storage 目录及其子目录具有写入权限。

服务器配置

就像大部分的 web 开发框架一样，Laravel 的设计也考虑了代码和存储安全的问题，Laravel 框架只把允许公众访问的文件放在 web 服务器的根目录(DocumentRoot)，这样可以有效的防止因为服务器的设置错误而

泄露重要的代码和信息资料。稳定压倒一切!!!

下面的案例中，我们假定把 Laravel 安装到 **/Users/JonSnow/Sites/MySite** 目录。

这对 MySite 的 Apache 虚拟主机的基本配置如下：

```
<VirtualHost *:80>

    DocumentRoot /Users/JonSnow/Sites/MySite/public

    ServerName mysite.dev

</VirtualHost>
```

注意，虽然安装目录是 **/Users/JonSnow/Sites/MySite** ，但是 DocumentRoot 配置项必须指向 **/Users/JonSnow/Sites/MySite/public** 目录。

将 DocumentRoot 指向 public 目录是一条通用的最佳实践方法，但是某些主机是不允许改变 DocumentRoot 配置的，没关系，人民的力量是无穷无尽的，Laravel 论坛里收集了一系列有用的方法。

基本设置

Laravel 框架的所有配置文件都存放在应用程序的 config 文件夹中，建议你把所有的配置文件都看一下，对应用程序的设置可以有一个基本的了解。你应该注意一下 **application/config/application.php** 文件，它包含了应用程序的基本设置。

在开发或者启用网站之前，你应该首先修改 **application/config/application.php** 中的 **application key**。Laravel 会用它来加密或者生成哈希。你可以手动设置一个32位的随机字符串，也可以使用 Laravel 提供的 Artisan 命令行工具来生成一个符合标准的字符串。有关 Aartisan 工具的信息可以访问 Artisan 命令列表。

注意： 如果你启用了 mod_rewrite，你应该把 index 参数设置为空。

环境设置

通常情况下，应用程序的开发环境和正式生产环境的设置是不同的，Laravel 使用的 URL 机制可以让你轻而易举的解决这个问题。打开 Laravel 框架的安装目录下的 paths.php 文件，你可以看到下面的数组：

```
$environments = array(
```

```
    'local' => array('http://localhost*', '*.dev'),

);
```

这个数组表示任何以"localhost"开头或者以"*.dev"结尾的请求，都被视为"local"环境。

然后，建立 **application/config/local** 文件夹，那么 local 文件夹下面的任何设置都会覆盖 **application/config** 中的基本设置。举个例子，你希望在新建的 **local** 目录中创建一个 **application.php** 文件：

```
return array(

    'url' => 'http://localhost/laravel/public',

);
```

在这个例子中，local 中的 **URL** 设置会覆盖 **application/config/application.php** 文件中的 **URL** 设置。需要注意的是，你只需要指定那些你想要覆盖的设置。

Laravel 的环境设置就是这么简单，你可以用它来创建你需要的环境。

友好的链接

通常情况下，你不想"index.php"出现在网站的链接中，那么你可以用 rewrite 重定向来去掉"index.php"。如果你使用的是 Apache 服务器，请启用 mod_rewrite 模块，然后在你的 public 目录创建一个**.htaccess** 文件,内容如下：

```
<IfModule mod_rewrite.c>
```

```
    RewriteEngine on



    RewriteCond %{REQUEST_FILENAME} !-f

    RewriteCond %{REQUEST_FILENAME} !-d



    RewriteRule ^(.*)$ index.php/$1 [L]

</IfModule>
```

如果上面的.htaccess 文件没有效果，可以试试下面的写法：

```
Options +FollowSymLinks

RewriteEngine on



RewriteCond %{REQUEST_FILENAME} !-f

RewriteCond %{REQUEST_FILENAME} !-d



RewriteRule . index.php [L]
```

在设置了重定向之后，你还应该把 **application/config/application.php** 中的 **index** 参数设置为空。

注意： 不同服务器的 rewrite 方法有所不同，请根据具体情况配置。

路由

目录

- 基础
- 通配符（Wildcards）
- 404事件（The 404 Event）
- 过滤器（Filters）
- 模式过滤器（Pattern Filters）
- 全局过滤器（Global Filters）
- 路由组（Route Groups）
- 命名路由（Named Routes）
- HTTPS 路由（HTTPS Routes）
- 扩展包路由（Bundle Routes）
- 控制器路由（Controller Routing）
- 命令行路由测试（CLI Route Testing）

基础

Laravel 充分利用 PHP 5.3的特性，使路由变得简单并富于表达性。这使得从构建 API 到完整的 web 应用都变得尽可能容易。路由的实现代码在 **application/routes.php** 文件。

和其它框架不同，应用逻辑在 Laravel 中可以通过两种方式集成。虽然在控制器（controllers）中实现业务逻辑是普遍的做法，但是在 Laravel 中也可以直接在路由中嵌入应用逻辑。这种方式**尤其**适用于只有几个页面的小型网站，这样就免去了创建一大堆控制器（controllers），还要为每个控制器创建一些不相干的方法（methods），而最后只是一部分方法（methods）通过手动设置路由的方式被暴露出来。

在下面的代码示例中，第一个参数（parameter）是你“注册”的路由（route），第二个参数是这个路由将要触发的函数（function），函数中包含了应用逻辑。定义路由时不需要开头的斜线（front-slash），唯一的例外是默认路由（default route）**只包含**一个斜线（front-slash）。

注意： 路由的权重在于其被注册的先后顺序。 因此，任何通配（catch-all）的路由应该在 **routes.php** 文件的底部注册

注册一个响应 "GET /" 的路由：

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

{

    return "Hello World!";
```

```
});
```

注册一个能同时响应（GET、POST、PUT、DELETE）HTTP 请求方法（HTTP verbs）的路由：

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

{

    return "Hello World!";

});
```

注册响应其它 HTTP 请求方法（HTTP verbs）的路由：

```
Route::post('user', function()

{

    //

});



Route::put('user/(:num)', function($id)

{

    //

});
```

```
Route::delete('user/(:num)', function($id)

{

    //

});
```

注册一个能响应多个 **HTTP** 请求方法（**HTTP verbs**）的路径（**URI**）：

```
Router::register(array('GET', 'POST'), $uri, $callback);
```

通配符（Wildcards）

强制路径（**URI**）中的某部分为数字：

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

{

    //

});
```

允许路径（**URI**）中的某部分是字母、数字串：

```
Route::get('post/(:any)', function($title)

{
```

```
    //

});
```

允许路径（URI）中的某部分是可选的：

```
Route::get('page/(:any?)', function($page = 'index')

{

    //

});
```

404事件（The 404 Event）

如果一个请求（request）不能匹配任何一个路由，404事件将被触发。在 **application/routes.php** 文件中可以找到默认的事件处理代码。

默认的404事件处理代码

```
Event::listen('404', function()

{

    return Response::error('404');

});
```

你可以按照你自己的需求定制这部分代码！

延伸阅读：

- *事件（Events）*

过滤器（Filters）

过滤器（filters）可以在路由之前或之后触发。如果在路由之前触发的过滤器（filters）有返回值，那么这个返回值将被认为是对此次请求（request）的回应（response），路由将停止执行。这一特性便于实现身份验证之类的功能。在 **application/routes.php** 文件中定义了所有过滤器（filters）。

注册一个过滤器（filter）：

```
Route::filter('filter', function()

{

    return Redirect::to('home');

});
```

绑定一个过滤器（filter）到路由：

```
Route::get('blocked', array('before' => 'filter', function()

{

    return View::make('blocked');

}));
```

给路由绑定一个之后（after）执行的过滤器（filter）：

```
Route::get('download', array('after' => 'log', function()
```

```
{

    //

}));
```

绑定多个过滤器（filters）到路由：

```
Route::get('create', array('before' => 'auth|csrf', function()

{

    //

}));
```

给过滤器（filters）传递参数：

```
Route::get('panel', array('before' => 'role:admin', function()

{

    //

}));
```

模式过滤器（Pattern Filters）

有时你可能需要针对所有包含部分路径（URI）的请求（request）绑定一个过滤器（filter），例如，你想对以“admin”开头的路径（URI）绑定一个叫”auth“的过滤器（filter），以下代码就是具体实现：w

D 定义一个基于路径模式（URI pattern）的过滤器（filter）：

```
Route::filter('pattern: admin/*', 'auth');
```

你也可以在为某个给定的路径（URI）绑定过滤器（filters）时直接提供一个带有名称（name）和回调函数（callback）的数组，这样，过滤器（filters）也就完成了注册。

Defining a filter and URI pattern based filter in one:

```
Route::filter('pattern: admin/*', array('name' => 'auth', function()

{

    //

}));
```

全局过滤器（Global Filters）

Laravel 默认有两个过滤器（filters），他们分别在每次请求（request）之前（before）和之后（after）执行。你可以在 **application/routes.php** 文件中找到这两个过滤器（filters）。这两个过滤器可以方便你启动通用扩展包（bundles）或者添加全局资源（assets）。

注意： 之后（after） 过滤器接收到的参数是对应当前请求（request）的 **回应（Response）** 对象。

路由组（Route Groups）

路由组方便你为一组路由绑定一些属性（attributes），从而保持代码的整洁。

```
Route::group(array('before' => 'auth'), function()

{
```

```
    Route::get('panel', function()

    {

        //

    });



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

    {

        //

    });

});
```

命名路由（Named Routes）

总是会有修改路由的时候，这就会让写死的路径（URI）产生错误。给路由赋予一个名称（name）可以方便通过这个名称（name）动态生成路径（URI），即便以后路由变化了，路径（URI）仍然和你新的路由保持一致。

定义一个命名路由：

```
Route::get('/', array('as' => 'home', function()

{

    return "Hello World";

}));
```

```

```

通过命名路由生成 URL：

```
$url = URL::to_route('home');
```

重定向到命名路由：

```
return Redirect::to_route('home');
```

对于一个命名路由，可以方便反查当前请求（request）是否是由这个命名路由在处理。

反查处理当前请求（request）的路由是否具有给定的名称（name）：

```
if (Request::route()->is('home'))

{

    // 名称为"home"的路由正在处理当前请求（request）!

}
```

HTTPS 路由（HTTPS Routes）

定义路由时，可以通过使用"https"参数指定所生成的 URL（或重定向时）采用 HTTPS 协议。

定义一个 HTTPS 路由：

```
Route::get('login', array('https' => true, function()
```

```
{

    return View::make('login');

}));
```

使用”secure“函数完成同样的事情：

```
Route::secure('GET', 'login', function()

{

    return View::make('login');

});
```

扩展包路由（Bundle Routes）

扩展包（bundle）是 Laravel 的模块化扩展系统。可以通过配置扩展包，方便的处理请求（request）。这里是扩展包的详细介绍，此处略过。 通过此段介绍，你会认识到扩展包不仅可以通过路由（route）暴露功能，还可以在扩展包中注册路由。

打开 **application/bundles.php** 文件，添加以下代码：

注册扩展包，处理相应的路由：

```
return array(



    'admin' => array('handles' => 'admin'),
```

```
);
```

注意到代码中的 **handles** 参数了吗？ 这告诉 Laravel 加载 Admin 扩展包并处理任何以“admin”开头的路径（URI）。

现在准备为你的扩展包注册几个路由吧，在你的扩展包的根目录创建 **routes.php** 文件，并添加以下代码：

给扩展包添加一个根路由（root route）：

```
Route::get('(:bundle)', function()

{

    return 'Welcome to the Admin bundle!';

});
```

我们来解析一下这段代码。注意到 **(:bundle)** 了吗？ 它将被替换为前面注册扩展包时的 **handles** 参数的值。这使你的代码保持 D.R.Y. - 不重复，还能让使用你的代码的开发者随意修改扩展包的根（root）URI 而不破快扩展包中定义的路由！

当然，你可以在扩展包中的所有路由上使用 **(:bundle)** 占位符，而不仅仅是跟路由（root route）。

注册扩展包路由：

```
Route::get('(:bundle)/panel', function()

{

    return "I handle requests to admin/panel!";

});
```

控制器路由（Controller Routing）

Controllers provide another way to manage your application logic. If you're unfamiliar with controllers you may want to read about controllers and return to this section.

It is important to be aware that all routes in Laravel must be explicitly defined, including routes to controllers. This means that controller methods that have not been exposed through route registration **cannot** be accessed. It's possible to automatically expose all methods within a controller using controller route registration. Controller route registrations are typically defined in **application/routes.php**.

Most likely, you just want to register all of the controllers in your application's "controllers" directory. You can do it in one simple statement. Here's how:

Register all controllers for the application:

```
Route::controller(Controller::detect());
```

The **Controller::detect** method simply returns an array of all of the controllers defined for the application.

If you wish to automatically detect the controllers in a bundle, just pass the bundle name to the method. If no bundle is specified, the application folder's controller directory will be searched.

Register all controllers for the "admin" bundle:

```
Route::controller(Controller::detect('admin'));
```

Registering the "home" controller with the Router:

```
Route::controller('home');
```

Registering several controllers with the router:

```
Route::controller(array('dashboard.panel',  'admin'));
```

```

```

Once a controller is registered, you may access its methods using a simple URI convention:

```
http://localhost/controller/method/arguments
```

This convention is similar to that employed by CodeIgniter and other popular frameworks, where the first segment is the controller name, the second is the method, and the remaining segments are passed to the method as arguments. If no method segment is present, the "index" method will be used.

This routing convention may not be desirable for every situation, so you may also explicitly route URIs to controller actions using a simple, intuitive syntax.

Registering a route that points to a controller action:

```
Route::get('welcome', 'home@index');
```

Registering a filtered route that points to a controller action:

```
Route::get('welcome', array('after' => 'log', 'uses' => 'home@index'));
```

Registering a named route that points to a controller action:

```
Route::get('welcome', array('as' => 'home.welcome', 'uses' => 'home@index'));
```

命令行路由测试（CLI Route Testing）

可以用 Laravel 自带的“Artisan”命令行工具测试你的路由，只需指定请求方法（request method）和需要测

试的 URI，所有返回的响应（response）都会通过 var_dump 函数输出到命令行上。

通过 Artisan 命令行工具调用路由：

```
php artisan route:call get api/user/1
```

控制器

目录

- 基础
- 控制器路由
- 插件控制器
- 行动过滤器
- 嵌套控制器
- 控制器布局
- REST 风格控制器
- 依赖注入
- 控制器工厂

基础

控制器是负责处理用户输入和管理模块、库与视图之间交互的类。通常情况下，控制器会向模块请求数据，然后把数据传递给视图，最后返回给用户。

在程序开发中控制器通常被用来实现应用逻辑。Laravel 框架还允许开发者在路由中声明应用逻辑，这部分会在路由文档中详细说明。但是我们鼓励新手仍然从控制器开始。在处理应用逻辑方面，控制器和路由没有什么不同。

控制器类都应该存放在 **application/controllers** 目录中，并且都继承于 Base_Controller 类。Laravel 框架默认自带了一个 Home_Controller 类。

创建一个简单的控制器:

```
class Admin_Controller extends Base_Controller
```

```
{



    public function action_index()

    {

        //

    }



}
```

Actions 是允许被访问的控制器方法，它们都应该以"action_"为前缀，除此以外的其他方法都是禁止访问的。

注： Base_Controller 类继承于 Laravel 框架的 Controller 类。

控制器路由

需要我们注意到是，在 Laragel 框架中所有的路由（包括控制器路由）都必须明确定义。

这意味着没有在路由中注册的控制器方法都是不可见的。使用控制器路由注册之后，控制器方法会自动接受访问。控制器路由的注册信息通常定义在 **application/routes.php** 文件中。

访问 the routing page 获取更多关于控制器路由的信息。.

插件控制器

插件包是 Laravel 框架的模块化管理系统。插件包可以非常容易的配置应用的处理请求。这部分我们将在[插件包]文档中作进一步了解。

创建插件包控制器的方法和创建应用控制器一样，只需要在控制器名前加上插件包名作为前缀。比如你的插件包名字叫"admin"，那么插件包控制器可以这样写：

创建插件控制器:

```
class Admin_Home_Controller extends Base_Controller

{



    public function action_index()

    {

        return "Hello Admin!";

    }



}
```

但是我们怎么用路由来注册插件控制器呢？其实也非常简单：

在路由中注册插件控制器:

```
Route::controller('admin::home');
```

现在我们就可以在浏览器中访问"admin"插件包的 home 控制器了。

注： 在 Laravel 框架中，我们使用双冒号来表示插件包，插件包更多信息可以阅读插件包文档。

行动过滤器

行动过滤器可以运行在控制方法之前,也可以在控制器方法之后.在 Laravel 框架中你不仅可以为控制器分配过滤器，同时还可以决定何种 HTTP 请求会触发过滤器。

你可以在控制器构造器中为控制器分配前置或者后置过滤器。

给所有请求附加过滤器:

```
$this->filter('before', 'auth');
```

在这个例子中，'auth'过滤器会运行在所有控制器方法之前。我们可以在 **application/routes.php** 文件中找到'auth'过滤器。它是用来验证用户是否登录，如果没有就会重定向到'login'。

给少数控制器附加过滤器:

```
$this->filter('before', 'auth')->only(array('index', 'list'));
```

在这个例子中 auth 过滤器会在 action_index()和 action_list()方法前运行，用户必须登录才能访问这些页面。但是该控制器中的其他方法不会触发身份验证。

给多数控制器附加过滤器:

```
$this->filter('before', 'auth')->except(array('add', 'posts'));
```

在前一个例子当中，过滤器只会运行在指定的控制器方法之前。而在这个例子中，我们声明的是不需要过滤的控制器方法。

为 POST 请求附加过滤器:

```
$this->filter('before', 'csrf')->on('post');
```

在这个例子中我给 POST 请求附加了一个 csrf 过滤器。'csrf'过滤器主要是用来过滤来自其他系统的 posts（比如传说中的 spam 机器人）。Larvel 框架自带了这个过滤器，你可以在 **application/routes.php** 文件中找到它。

进阶阅读:

- *路由过滤器*

嵌套控制器

控制器可以存放在 **application/controllers** 目录的任意层次的子目录中。

创建一个控制器 **controllers/admin/panel.php**，代码如下：

```
class Admin_Panel_Controller extends Base_Controller

{



    public function action_index()

    {

        //

    }



}
```

在路由中用'.'号来注册嵌套控制器：

```
Route::controller('admin.panel');
```

注： 当使用嵌套控制器的时候，控制器的注册顺序总是按目录的层次从深到浅。

访问控制器的'index'方法:

```
http://localhost/admin/panel
```

控制器布局

控制器布局的完整文档请阅读模板文档.

REST 风格控制器

除了使用"action_"前缀之外，我们还可以使用 HTTP 请求类型来作为控制器方法的前缀。

为控制添加 REST 风格属性：

```
class Home_Controller extends Base_Controller

{

    public $restful = true;

}
```

建立 REST 风格的控制器方法:

```
class Home_Controller extends Base_Controller

{

    public $restful = true;
```

```
    public function get_index()
    {
        //
    }

    public function post_index()
    {
        //
    }

}
```

在我们建立 CRUD 方法的时候这种风格非常的友好。

依赖注入

如果你正在编写可测试的代码，你可能会想在你的控制器构造使用依赖注入。这非常简单，只需要在 IoC 容器中注册你的控制器即可。在注册控制器时，需要使用 **controller** 前缀，因此在 **application/start.php** 文件中，我们可以像下面这样注册控制器：

```
IoC::register('controller: user', function()
{
    return new User_Controller;
});
```

```

```

当控制器接收请求时，Laravel 框架会自动检测控制器在容器中是否注册，如果已注册，那么将会生成一个控制器实例。

注： 在使用控制器的依赖注入前，你应该阅读 IoC 容器文档。

控制器工厂

如果你想更好的控制控制器实例，那么你就需要是使用 Laravel 提供的控制器工厂。

为控制器实例注册一个事件：

```
Event::listen(Controller::factory, function($controller)

{

    return new $controller;

});
```

这个事件会接收需要实例化的类名，然后返回给你一个控制器实例。

模型与类库

目录

- 模型
- 类库
- 自动加载
- 最佳实践

模型

模型是应用程序的核心部分，应用逻辑（控制器/路由）和视图都是用户与模型进行交互的媒介。模型中最典型的逻辑是商业逻辑。

下面是模型中常见的功能：

- 数据库交互
- 文件 I/O
- 服务器交互

例如，可能你正在写一个博客程序，那么你就需要一个"post"模型。如果允许用户评论的话，你还需要一个"comment"模型和"User"模型。

类库

类库是框架中能够实现某种功能但是不属于某个具体应用的类的集合。例如 PDF 生成类可以转换 HTML 文件，它的功能相对复杂，并且可以独立于应用程序之外，我们就可以把它放到"library"（类库）里。

创建一个库类和创建普通的类一样容易，只需要把它放到 libraries 文件夹中。举个例子，下面我们将创建一个简单的库类，它可以打印我们传入的信息。我们只需要在 libraries 目录中建立一个 **printer.php** 文件，内容如下：

```
<?php



class Printer {



    public static function write($text) {

        echo $text;

    }

}
```

现在你就可以在应用程序的任何地方调用 Printer::write()方法了。

自动加载

Laravel 框架的自动加载器让我们能够非常容易的使用模型和类库。关于自动加载器可以阅读：Auto-Loading](/docs/loading)。

最佳实践

我们经常听到这样一句话："控制器应该保持简洁！"但是在实践中我们怎么才能做到呢？问题的关键在于我们怎么理解"模型"这个词。多数时候模型只被用来负责和数据库交互，这会导致了控制器过度臃肿。让我们来尝试下不同的方法。

如果我们不用"modles"目录又该怎么做呢？让我们为它取个更容易理解的名字，比如我们的卫星导航网站叫"Trackler"，那我们就在应用目录里创建一个"trackler"文件夹。

现在我们把功能分为三个部分："entities"，"services"和"repositories"。然后在"trackler"目录中建立这三个文件夹：

Entities

我们把 entities 作为应用的数据容器，他们负责存储属性。在例子中我们有一个"location"，它具有经度和纬度两个属性：

```
<?php namespace Trackler\Entities;



class Location {



    public $latitude;

    public $longitude;



    public function __construct($latitude, $longitude)

    {

        $this->latitude = $latitude;
```

```
        $this->longitude = $longitude;

    }


}
```

Services

Services 包含了应用程序的*流程*。继续用 Trackler 举例，应用中有一个表单，用户可以输入自己的 GPS 坐标。但是我们需要验证用户输入坐标的格式，那么我们就在"services"目录中建立一个"validators"文件夹，同时创建一个验证类：

```
<?php namespace Trackler\Services\Validators;


use Trackler\Entities\Location;


class Location_Validator {


    public static function validate(Location $location)

    {

        // Validate the location instance...

    }


}
```

Repositories

Repositories 是应用的数据访问层，它主要负责查询和储存应用的 *entities*。继续举例，我们需要一个坐标库来储存坐标，我们可以用任何方式储存：

```
<?php namespace Trackler\Repositories;

use Trackler\Entities\Location;

class Location_Repository {

    public function save(Location $location, $user_id)
    {
        // Store the location for the given user ID...
    }

}
```

现在我们完成了应用程序三个功能的分离，数据库操作被隔离起来，我们就可以在 services 和 controllers 里使用 respositories，而不用担心切换数据储存方式给应用程序带来影响了。

视图与回应(RESPONSE)

目录

- 基础
- 绑定数据到视图
- 嵌套视图
- 命名视图
- 视图合成
- 重定向
- Redirecting With Flash Data
- 下载
- 错误

基础

Views contain the HTML that is sent to the person using your application. By separating your view from the business logic of your application, your code will be cleaner and easier to maintain.

All views are stored within the **application/views** directory and use the PHP file extension. The **View** class provides a simple way to retrieve your views and return them to the client. Let's look at an example!

创建视图：

```
<html>

    I'm stored in views/home/index.php!

</html>
```

从路由中返回视图实例：

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

{
```

```
    return View::make('home.index');

});
```

从控制器中返回视图实例：

```
public function action_index()

{

    return View::make('home.index');

});
```

检查是否存在某个视图文件：

```
$exists = View::exists('home.index');
```

Sometimes you will need a little more control over the response sent to the browser. For example, you may need to set a custom header on the response, or change the HTTP status code. Here's how:

返回一个自定义的 **Response** 实例：

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

{

    $headers = array('foo' => 'bar');



    return Response::make('Hello World!', 200, $headers);

});
```

```

```

Returning a custom response containing a view, with binding data:

```
return Response::view('home', array('foo' => 'bar'));
```

返回携带 JSON 数据的 Response 实例：

```
return Response::json(array('name' => 'Batman'));
```

Eloquent

以 JSON 格式返回带有 Eloquent 模型的 Response 实例：

```
return Response::eloquent(User::find(1));
```

给视图绑定数据

Typically, a route or controller will request data from a model that the view needs to display. So, we need a way to pass the data to the view. There are several ways to accomplish this, so just pick the way that you like best!

绑定数据到视图：

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

{

    return View::make('home')->with('name', 'James');

});
```

```

```

Accessing the bound data within a view:

```
<html>

    Hello, <?php echo $name; ?>.

</html>
```

Chaining the binding of data to a view:

```
View::make('home')

    ->with('name', 'James')

    ->with('votes', 25);
```

Passing an array of data to bind data:

```
View::make('home', array('name' => 'James'));
```

Using magic methods to bind data:

```
$view->name  = 'James';

$view->email = 'example@example.com';
```

Using the ArrayAccess interface methods to bind data:

```
$view['name']  = 'James';

$view['email'] = 'example@example.com';
```

嵌套视图

Often you will want to nest views within views. Nested views are sometimes called "partials", and help you keep views small and modular.

Binding a nested view using the "nest" method:

```
View::make('home')->nest('footer', 'partials.footer');
```

向嵌套视图传递数据：

```
$view = View::make('home');

$view->nest('content', 'orders', array('orders' => $orders));
```

Sometimes you may wish to directly include a view from within another view. You can use the **render** helper function:

调用 "render" 函数输出视图：

```
<div class="content">

    <?php echo render('user.profile'); ?>
```

```
</div>
```

It is also very common to have a partial view that is responsible for display an instance of data in a list. For example, you may create a partial view responsible for displaying the details about a single order. Then, for example, you may loop through an array of orders, rendering the partial view for each order. This is made simpler using the **render_each** helper:

Rendering a partial view for each item in an array:

```
<div class="orders">

    <?php echo render_each('partials.order', $orders, 'order');

</div>
```

The first argument is the name of the partial view, the second is the array of data, and the third is the variable name that should be used when each array item is passed to the partial view.

命名视图

Named views can help to make your code more expressive and organized. Using them is simple:

注册一个命名视图：

```
View::name('layouts.default', 'layout');
```

获取某个命名视图的实例：

```
return View::of('layout');
```

绑定数据到某个命名视图的实例：

```
return View::of('layout', array('orders' => $orders));
```

View Composers

Each time a view is created, its "composer" event will be fired. You can listen for this event and use it to bind assets and common data to the view each time it is created. A common use-case for this functionality is a side-navigation partial that shows a list of random blog posts. You can nest your partial view by loading it in your layout view. Then, define a composer for that partial. The composer can then query the posts table and gather all of the necessary data to render your view. No more random logic strewn about! Composers are typically defined in **application/routes.php**. Here's an example:

Register a view composer for the "home" view:

```
View::composer('home', function($view)

{

    $view->nest('footer', 'partials.footer');

});
```

Now each time the "home" view is created, an instance of the View will be passed to the registered Closure, allowing you to prepare the view however you wish.

Register a composer that handles multiple views:

```
View::composer(array('home', 'profile'), function($view)

{

    //

});
```

```

```

Note: A view can have more than one composer. Go wild!

重定向

It's important to note that both routes and controllers require responses to be returned with the 'return' directive. Instead of calling "Redirect::to()"" where you'd like to redirect the user. You'd instead use "return Redirect::to()". This distinction is important as it's different than most other PHP frameworks and it could be easy to accidentally overlook the importance of this practice.

重定向到另一个 URI：

```
return  Redirect::to('user/profile');
```

Redirecting with a specific status:

```
return  Redirect::to('user/profile',  301);
```

重定向到一个 HTTPS URI：

```
return  Redirect::to_secure('user/profile');
```

重定向到网站首页：

```
return  Redirect::home();
```

重定向到前一个页面：

```
return  Redirect::back();
```

```

```

重定向到一个命名路由：

```
return  Redirect::to_route('profile');
```

重定向到一个控制器动作：

```
return  Redirect::to_action('home@index');
```

Sometimes you may need to redirect to a named route, but also need to specify the values that should be used instead of the route's URI wildcards. It's easy to replace the wildcards with proper values:

Redirecting to a named route with wildcard values:

```
return  Redirect::to_route('profile',  array($username));
```

Redirecting to an action with wildcard values:

```
return  Redirect::to_action('user@profile',  array($username));
```

Redirecting With Flash Data

After a user creates an account or signs into your application, it is common to display a welcome or status message. But, how can you set the status message so it is available for the next request? Use the with() method to send flash data along with the redirect response.

```
return  Redirect::to('profile')->with('status',  'Welcome  Back!');
```

```

```

You can access your message from the view with the Session get method:

```
$status = Session::get('status');
```

Further Reading:

- *Sessions*

下载

Sending a file download response:

```
return Response::download('file/path.jpg');
```

Sending a file download and assigning a file name:

```
return Response::download('file/path.jpg', 'photo.jpg');
```

错误

To generating proper error responses simply specify the response code that you wish to return. The corresponding view stored in **views/error** will automatically be returned.

返回一个带有**404**错误码的 **Response** 实例：

```
return Response::error('404');
```

```

```

返回一个带有**500**错误码的 **Response** 实例：

```
return  Response::error('500');
```

管理静态资源文件

目录

- 注册静态资源文件
- Dumping Assets
- 静态资源文件间的依赖关系
- 静态资源文件容器
- 扩展包中的静态资源文件

注册静态资源文件

The **Asset** class provides a simple way to manage the CSS and JavaScript used by your application. To register an asset just call the **add** method on the **Asset** class:

注册一个静态资源文件

```
Asset::add('jquery',  'js/jquery.js');
```

The **add** method accepts three parameters. The first is the name of the asset, the second is the path to the asset relative to the **public** directory, and the third is a list of asset dependencies (more on that later). Notice that we did not tell the method if we were registering JavaScript or CSS. The **add** method will use the file extension to determine the type of file we are registering.

输出静态资源文件地址

When you are ready to place the links to the registered assets on your view, you may use the **styles** or **scripts** methods:

向视图中输出静态资源文件地址：

```
<head>

    <?php echo Asset::styles(); ?>

    <?php echo Asset::scripts(); ?>

</head>
```

静态资源文件间的依赖关系

Sometimes you may need to specify that an asset has dependencies. This means that the asset requires other assets to be declared in your view before it can be declared. Managing asset dependencies couldn't be easier in Laravel. Remember the "names" you gave to your assets? You can pass them as the third parameter to the **add** method to declare dependencies:

Registering a bundle that has dependencies:注册某个扩展包

```
Asset::add('jquery-ui', 'js/jquery-ui.js', 'jquery');
```

In this example, we are registering the **jquery-ui** asset, as well as specifying that it is dependent on the **jquery** asset. Now, when you place the asset links on your views, the jQuery asset will always be declared before the jQuery UI asset. Need to declare more than one dependency? No problem:

Registering an asset that has multiple dependencies:

```
Asset::add('jquery-ui', 'js/jquery-ui.js', array('first', 'second'));
```

静态资源文件容器

To increase response time, it is common to place JavaScript at the bottom of HTML documents. But, what if you also need to place some assets in the head of your document? No problem. The asset class provides a simple way to manage asset **containers**. Simply call the **container** method on the Asset class and mention the container name. Once you have a container instance, you are free to add any assets you wish to the container using the same syntax you are used to:

获取一个静态资源文件容器的实例：

```
Asset::container('footer')->add('example',  'js/example.js');
```

从某个容器中输出所有静态资源文件地址：

```
echo  Asset::container('footer')->scripts();
```

扩展包中的静态资源文件

Before learning how to conveniently add and dump bundle assets, you may wish to read the documentation on creating and publishing bundle assets.

When registering assets, the paths are typically relative to the **public** directory. However, this is inconvenient when dealing with bundle assets, since they live in the **public/bundles** directory. But, remember, Laravel is here to make your life easier. So, it is simple to specify the bundle which the Asset container is managing.

Specifying the bundle the asset container is managing:

```
Asset::container('foo')->bundle('admin');
```

Now, when you add an asset, you can use paths relative to the bundle's public directory. Laravel will automatically generate the correct full paths.

模版

目录

- 基础
- 片段
- Blade 模版引擎
- Blade 控制结构
- Blade 布局

基础

也许你的应用中所有页面都是用同样的布局方式，如果手工的为每一个控制器 action 方法都创建一次页面布局将会是非常痛苦的，如果能够为控制器指定一个公用布局的话将会让开发变得更轻松。下面就来介绍如何实现吧：

为控制器指定一个 "layout" 属性：

```
class Base_Controller extends Controller {

    public $layout = 'layouts.common';

}
```

在控制器 action 方法中访问 "layout" 属性：

```
public function action_profile()
{
    $this->layout->nest('content', 'user.profile');
}
```

注： 控制器中使用了公共布局之后，action 方法不许要返回任何东西了。

Sections

View sections provide a simple way to inject content into layouts from nested views. For example, perhaps you want to inject a nested view's needed JavaScript into the header of your layout. Let's dig in:

Creating a section within a view:

```
<?php Section::start('scripts'); ?>
    <script src="jquery.js"></script>
<?php Section::stop(); ?>
```

Rendering the contents of a section:

```
<head>
    <?php echo Section::yield('scripts'); ?>
</head>
```

Using Blade short-cuts to work with sections:

```
@section('scripts')
    <script src="jquery.js"></script>
@endsection

<head>
    @yield('scripts')
</head>
```

Blade 模版引擎

Blade makes writing your views pure bliss. To create a blade view, simply name your view file with a ".blade.php" extension. Blade allows you to use beautiful, unobtrusive syntax for writing PHP control structures and echoing data. Here's an example:

输出一个变量：

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

输出函数的返回值：

```
{{ Asset::styles() }}
```

输出视图（view）

你可以在一个视图中使用 **@include** 指令输出另一个视图。被输出的视图会自动继承当前视

图的所有数据。

```
<h1>Profile</hi>
@include('user.profile')
```

同样的，你还可以使用 **@render** 指令输出一个视图，和 **@include** 指令不同的是，**@render** 指令输出的视图 **不会** 继承当前视图的数据。

```
@render('admin.list')
```

Blade 注释：

```
{{-- 只是一行注释 --}}
```

```
{{--
    这是一个
    多行
    注释。
--}}
```

注： 和 HTML 注释不同，Blade 注释不会出现在生成的 HTML 源码中。

Blade 控制结构

For 循环：

```
@for ($i = 0; $i <= count($comments); $i++)
    The comment body is {{ $comments[$i] }}
@endfor
```

Foreach 循环：

```
@foreach ($comments as $comment)
    The comment body is {{ $comment->body }}.
@endforeach
```

While 循环：

```
@while ($something)
    I am still looping!
@endwhile
```

If 表达式：

```
@if ( $message == true )
    I'm displaying the message!
@endif
```

If Else 表达式：

```
@if (count($comments) > 0)
    I have comments!
@else
    I have no comments!
@endif
```

Else If 表达式：

```
@if ( $message == 'success' )
    It was a success!
@elseif ( $message == 'error' )
    An error occurred.
@else
    Did it work?
@endif
```

构建 HTML

目录

- 实体符号
- 脚本(Script)与样式表(CSS)
- 链接
- 到命名路由的链接
- 到控制器动作的链接
- Mail-To 链接
- 图像
- 列表
- 自定义宏

实体符号

When displaying user input in your Views, it is important to convert all characters which have significance in HTML to their "entity" representation.

For example, the < symbol should be converted to its entity representation. Converting HTML characters to their entity representation helps protect your application from cross-site scripting:

Converting a string to its entity representation:

```
echo  HTML::entities('<script>alert(\'hi\');</script>');
```

Using the "e" global helper:

```
echo  e('<script>alert(\'hi\');</script>');
```

脚本(Script)与样式表(CSS)

生成一个引用 **JavaScript** 文件的 **script** 标签：

```
echo  HTML::script('js/scrollTo.js');
```

生成一个引用 **CSS** 文件的 **link** 标签：

```
echo  HTML::style('css/common.css');
```

Generating a reference to a CSS file using a given media type:

```
echo HTML::style('css/common.css', 'print');
```

扩展阅读：

- *管理静态资源*

链接

Generating a link from a URI:

```
echo HTML::link('user/profile', 'User Profile');
```

Generating a link that should use HTTPS:

```
echo HTML::secure_link('user/profile', 'User Profile');
```

Generating a link and specifying extra HTML attributes:

```
echo HTML::link('user/profile', 'User Profile', array('id' => 'profile_link'));
```

到命名路由的链接

Generating a link to a named route:

```
echo HTML::link_to_route('profile');
```

```

```

Generating a link to a named route with wildcard values:

```
$url = HTML::link_to_route('profile', 'User Profile', array($username));
```

Further Reading:

- *Named Routes*

到控制器动作的链接

生成到控制器动作的链接：

```
echo HTML::link_to_action('home@index');
```

Generating a link to a controller action with wildcard values:

```
echo HTML::link_to_action('user@profile', 'User Profile', array($username));
```

Mail-To 链接

The "mailto" method on the HTML class obfuscates the given e-mail address so it is not sniffed by bots.

生成 **Mail-To** 链接：

```
echo HTML::mailto('example@gmail.com', 'E-Mail Me!');
```

Creating a mail-to link using the e-mail address as the link text:

```
echo HTML::mailto('example@gmail.com');
```

图像

生成 **img** 标签：

```
echo HTML::image('img/smile.jpg', $alt_text);
```

生成带有额外 **HTML** 属性的 **img** 标签：

```
echo HTML::image('img/smile.jpg', $alt_text, array('id' => 'smile'));
```

列表

从数组生成列表：

```
echo HTML::ol(array('Get Peanut Butter', 'Get Chocolate', 'Feast'));

echo HTML::ul(array('Ubuntu', 'Snow Leopard', 'Windows'));

echo HTML::dl(array('Ubuntu' => 'An operating system by Canonical', 'Windows' => 'An operating system by Microsoft'));
```

自定义宏

It's easy to define your own custom HTML class helpers called "macros". Here's how it works. First, simply register the macro with a given name and a Closure:

注册一个 HTML 宏：

```
HTML::macro('my_element', function()

{

    return '<article type="awesome">';

});
```

现在你就可以用这个宏的名称调用它了：

调用自定义的宏：

```
echo HTML::my_element();
```

输入与 COOKIE

目录

- 输入
- JSON 输入

- 文件
- Old Input
- Redirecting With Old Input
- Cookies
- 合并与替换

Input

Input 类会处理应用程序中来自 GET、POST、PUT 或者 DELETE 类型的请求。下面是一些用 Input 类来访问 Input 数据的例子：

获取 input 数组中指定的值：

```
$email = Input::get('email');
```

Note: "get"方法适用于所有类型(GET,POST,PUT 和 DELETE)的请求，而不仅仅是指 GET 请求。

获取 input 数组中的所有数据：

```
$input = Input::get();
```

获取所有 input 数据，包括$_FILES 数组：

```
$input = Input::all();
```

默认情况下，如果获取的 input 数据不存在时会返回 *null* 值。不过你可以通过给函数传递第二个参数来替代 *null* 值：

当请求的数据不存在时返回一个默认值：

```
$name = Input::get('name', 'Fred');
```

```

```

把匿名函数作为默认返回值：

```
$name = Input::get('name', function() {return 'Fred';});
```

检查 input 数组中是否包含指定条目：

```
if (Input::has('name')) ...
```

Note: 当检查的数据不存在的时候"has"方法会返回 *false*。

JSON Input

当使用像 Backbone.js 这样的 JavaScript MVC 框架时，你可能需要获取应用程序发送的 JSON 数据。Laravel 框架提供了'Input::json'方法使你的工作更加轻松：

获取来自应用程序的 JSON 数据：

```
$data = Input::json();
```

文件类

获取$_FILES 数组：

```
$files = Input::file();
```

获取$_FILES 数组中的指定数据：

```
$picture = Input::file('picture');
```

获取$_FILES 数组中特定条目的 size:

```
$size = Input::file('picture.size');
```

Old Input

You'll commonly need to re-populate forms after invalid form submissions. Laravel's Input class was designed with this problem in mind. Here's an example of how you can easily retrieve the input from the previous request. First, you need to flash the input data to the session:

Flashing input to the session:

```
Input::flash();
```

Flashing selected input to the session:

```
Input::flash('only', array('username', 'email'));

Input::flash('except', array('password', 'credit_card'));
```

Retrieving a flashed input item from the previous request:

```
$name = Input::old('name');
```

Note: You must specify a session driver before using the "old" method.
Further Reading:

- *Sessions*

Redirecting With Old Input

Now that you know how to flash input to the session. Here's a shortcut that you can use when redirecting that prevents you from having to micro-manage your old input in that way:

Flashing input from a Redirect instance:

```
return Redirect::to('login')->with_input();
```

Flashing selected input from a Redirect instance:

```
return Redirect::to('login')->with_input('only', array('username'));

return Redirect::to('login')->with_input('except', array('password'));
```

Cookies

Laravel provides a nice wrapper around the $_COOKIE array. However, there are a few things you should be aware of before using it. First, all Laravel cookies contain a "signature hash". This allows the framework to verify that the cookie has not been modified on the client. Secondly, when setting cookies, the cookies

are not immediately sent to the browser, but are pooled until the end of the request and then sent together. This means that you will not be able to both set a cookie and retrieve the value that you set in the same request.

Retrieving a cookie value:

```
$name = Cookie::get('name');
```

Returning a default value if the requested cookie doesn't exist:

```
$name = Cookie::get('name', 'Fred');
```

Setting a cookie that lasts for 60 minutes:

```
Cookie::put('name', 'Fred', 60);
```

Creating a "permanent" cookie that lasts five years:

```
Cookie::forever('name', 'Fred');
```

Deleting a cookie:

```
Cookie::forget('name');
```

Merging & Replacing

Sometimes you may wish to merge or replace the current input. Here's how:

Merging new data into the current input:

```
Input::merge(array('name' => 'Spock'));
```

Replacing the entire input array with new data:

```
Input::merge(array('doctor' => 'Bones', 'captain' => 'Kirk'));
```

For Else 表达式：

```
@forelse ($posts as $post)
    {{ $post->body }}
@empty
    There are not posts in the array!
@endforelse
```

Unless 表达式：

```
@unless(Auth::check())
    Login
@endunless

// Equivalent to...

<?php if ( ! Auth::check()): ?>
    Login
<?php endif; ?>
```

Blade 布局

Not only does Blade provide clean, elegant syntax for common PHP control structures, it also gives you a beautiful method of using layouts for your views. For example, perhaps your application uses a "master" view to provide a common look and feel for your application. It may look something like this:

```
<html>
    <ul class="navigation">
        @section('navigation')
            <li>Nav Item 1</li>
            <li>Nav Item 2</li>
        @yield_section
    </ul>

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

Notice the "content" section being yielded. We need to fill this section with some text, so let's make another view that uses this layout:

```
@layout('master')

@section('content')
    Welcome to the profile page!
@endsection
```

Great! Now, we can simply return the "profile" view from our route:

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

The profile view will automatically use the "master" template thanks to Blade's **@layout** expression.

Important: The **@layout** call must always be on the very first line of the file, with no leading whitespaces or newline breaks.

Appending with @parent

Sometimes you may want to only append to a section of a layout rather than overwrite it. For example, consider the navigation list in our "master" layout. Let's assume we just want to append a new list item. Here's how to do it:

```
@layout('master')
```

```
@section('navigation')
    @parent
    <li>Nav Item 3</li>
@endsection

@section('content')
    Welcome to the profile page!
@endsection
```

@parent will be replaced with the contents of the layout's *navigation* section, providing you with a beautiful and powerful method of performing layout extension and inheritance.

输入与 COOKIE

目录

- 输入
- JSON 输入
- 文件
- Old Input
- Redirecting With Old Input
- Cookies
- 合并与替换

Input

Input 类会处理应用程序中来自 GET、POST、PUT 或者 DELETE 类型的请求。下面是一些用 Input 类来访问 Input 数据的例子：

获取 input 数组中指定的值：

```
$email = Input::get('email');
```

Note: "get"方法适用于所有类型(GET,POST,PUT 和 DELETE)的请求，而不仅仅是指 GET 请求。

获取 input 数组中的所有数据：

```
$input = Input::get();
```

获取所有 input 数据，包括$_FILES 数组：

```
$input = Input::all();
```

默认情况下，如果获取的 input 数据不存在时会返回 *null* 值。不过你可以通过给函数传递第二个参数来替代 *null* 值：

当请求的数据不存在时返回一个默认值：

```
$name = Input::get('name', 'Fred');
```

把匿名函数作为默认返回值：

```
$name = Input::get('name', function() {return 'Fred';});
```

检查 input 数组中是否包含指定条目：

```
if (Input::has('name')) ...
```

Note: 当检查的数据不存在的时候"has"方法会返回 *false*。

JSON Input

当使用像 Backbone.js 这样的 JavaScript MVC 框架时，你可能需要获取应用程序发送的 JSON 数据。Laravel 框架提供了'Input::json'方法使你的工作更加轻松：

获取来自应用程序的 JSON 数据：

```
$data = Input::json();
```

文件类

获取$_FILES 数组：

```
$files = Input::file();
```

获取$_FILES 数组中的指定数据：

```
$picture = Input::file('picture');
```

获取$_FILES 数组中特定条目的 size:

```
$size = Input::file('picture.size');
```

Old Input

You'll commonly need to re-populate forms after invalid form submissions. Laravel's Input class was designed with this problem in mind. Here's an example of how you can easily retrieve the input from the previous request. First, you need to flash the input data to the session:

Flashing input to the session:

```
Input::flash();
```

Flashing selected input to the session:

```
Input::flash('only', array('username', 'email'));


Input::flash('except', array('password', 'credit_card'));
```

Retrieving a flashed input item from the previous request:

```
$name = Input::old('name');
```

Note: You must specify a session driver before using the "old" method.
Further Reading:

- *Sessions*

Redirecting With Old Input

Now that you know how to flash input to the session. Here's a shortcut that you can use when redirecting that prevents you from having to micro-manage your old input in that way:

Flashing input from a Redirect instance:

```
return Redirect::to('login')->with_input();
```

Flashing selected input from a Redirect instance:

```
return Redirect::to('login')->with_input('only', array('username'));



return Redirect::to('login')->with_input('except', array('password'));
```

Cookies

Laravel provides a nice wrapper around the $_COOKIE array. However, there are a few things you should be aware of before using it. First, all Laravel cookies contain a "signature hash". This allows the framework to verify that the cookie has not been modified on the client. Secondly, when setting cookies, the cookies are not immediately sent to the browser, but are pooled until the end of the request and then sent together. This means that you will not be able to both set a cookie and retrieve the value that you set in the same request.

Retrieving a cookie value:

```
$name = Cookie::get('name');
```

Returning a default value if the requested cookie doesn't exist:

```
$name = Cookie::get('name', 'Fred');
```

Setting a cookie that lasts for 60 minutes:

```
Cookie::put('name', 'Fred', 60);
```

Creating a "permanent" cookie that lasts five years:

```
Cookie::forever('name',  'Fred');
```

Deleting a cookie:

```
Cookie::forget('name');
```

Merging & Replacing

Sometimes you may wish to merge or replace the current input. Here's how:

Merging new data into the current input:

```
Input::merge(array('name'  =>  'Spock'));
```

Replacing the entire input array with new data:

```
Input::merge(array('doctor'  =>  'Bones',  'captain'  =>  'Kirk'));
```

插件包

目录

- 基础

- 创建插件包
- 注册插件包
- 插件包与类加载
- 启动插件包
- 插件包路由
- 使用插件包
- 插件包的静态文件
- 安装插件包
- 升级插件包

基础

插件包是 Laravel 3.0 改进中最核心的部分。插件包让我们能更加方便的把代码组织到一个模块中。每个插件包都可以包含自己的视图、设置、路由、数据库迁移、任务等。你可以把插件包用到任何地方，它可以是一个对象关系数据库映射，也可以是一个强大的身份验证系统。插件包的代码模块化是驱动 Laravel 所有设计决策的重要方面。你甚至可以把整个 application 目录看作是一个 Laravel 框架默认加载和使用的插件包。

创建插件包

在创建一个插件包之前，你必须在 **bundles** 目录为它新建一个文件夹。比如我们要建立一个"admin"插件包，作为我们应用的管理后台。在 **application/start.php** 文件中提供的基本配置可以帮助我们限定应用程序如何运行。同样，我们要在新建的插件包文件夹中创建一个 start.php 文件，每次加载插件包的时候它都会运行。让我们来创建它：

为插件包创建一个 start.php 文件:

```
<?php



Autoloader::namespaces(array(

    'Admin'  =>  Bundle::path('admin').'models',

));
```

```

```

在这个 start 文件中，我们让自动加载类自动加载在 admin 命名空间中定义的 models 目录下的类。你可以在 start 文件中做任何事情，但是通常情况下，我们用它来自动加载需要的文件。**事实上，你并不一定要为应用包建立一个启动文件。**

然后，我们来看怎么为应用程序注册一个插件包！

注册插件包

先我们建立了一个 admin 插件包，我们需要在 Laravel 中注册它。打开 **application/bundles.php** 文件。我们应用程序使用的所有插件包都在这里注册的。让我们把 admin 插件包添加进去：

注册一个简单的插件包:

```
return  array('admin'),
```

默认情况下，Laravel 会假设 Admin 插件包是在插件目录的根目录下，但是我们也可以自定义插件包路径。

注册一个自定义路径的插件包:

```
return  array(

    'admin'  =>  array('location'  =>  'userscape/admin'),

);
```

现在 Laravel 会在 **bundles/userscape/admin** 目录中寻找 admin 插件包。

插件包与类加载

通常情况下，插件包的 **start.php** 文件只包含了自动加载注册信息。因此有时候你希望跳过 **start.php** 文件，直接在注册数组中声明自己的插件包映射信息。可以用下面的方法：

在插件注册数组中定义自动加载映射：

```
return array(



    'admin' => array(

        'autoloads' => array(

            'map' => array(

                'Admin' => '(:bundle)/admin.php',

            ),

            'namespaces' => array(

                'Admin' => '(:bundle)/lib',

            ),

            'directories' => array(

                '(:bundle)/models',

            ),

        ),

    ),



);
```

```

```

请注意，这里的每一项设置都对应 Laravelauto-loader 中的一个函数。事实上，每一项设置的参数都会自动传递给 auto-loader 中对应的函数。

你还要注意下**(:bundle)**占位符。它会自动替换为插件包的路径。

启动插件包

现在我们已经创建并注册了插件包，但是我们还不能使用它，我们需要先启动它：

启用一个插件包:

```
Bundle::start('admin');
```

这行代码告诉Laravel执行插件包的**start.php**文件，并在auto-loader中注册。如果插件包存在**routes.php**文件，start 方法也会自动加载。

Note: 插件包只会启动一次，重复的请求都会被 start 方法忽略。
如果你在应用程序中使用了一个插件包，但是又不想每次使用前都手动启动它。那么你可以在**application/bundles.php** 文件中，把它设置为自动启动：

把插件包设置为自动启动:

```
return array(

    'admin' => array('auto' => true),

);
```

如果插件包被设置为自动启动，那你不必要再去手动启用一个插件包，你只需要直接使用它，剩余的工作都交给 Laravel 去完成。如果你尝试使用插件包的视图文件，那么插件包的配置文件、语言文件、路由或者

过滤器都会自动启用。

每当一个插件包启动成功，都将触发一个事件。你可以用下面的方法来监听已经启动的插件包：

监听插件包的启动事件:

```
Event::listen('laravel.started: admin', function()

{

    // The "admin" bundle has started...

});
```

同样还可以将一个插件包设置为不可启用，那么这个插件包就不能被启用了。

把插件包设置为不可启用:

```
Bundle::disable('admin');
```

插件包路由

更多关于插件包路由和控制器的信息可以阅读：bundle routing 和 bundle controllers。

使用插件包

前面已经说过，插件包可以包含自己的视图、配置、语言等文件。在 Laravel 框架中可以通过双冒号来使用它们。让我们看下面的例子：

获取插件包视图:

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

获取插件包配置:

```
return  Config::get('bundle::file.option');
```

获取插件包语言:

```
return  Lang::line('bundle::file.line');
```

有些时候你想要获取插件包更多的"meta"信息，比如插件包是否存在、插件包路径或者插件包的配置数组。那么可以用下面的方法：

确认插件包是否存在:

```
Bundle::exists('admin');
```

获取插件包的安装路径:

```
$location  =  Bundle::path('admin');
```

获取插件包的配置:

```
$config  =  Bundle::get('admin');
```

获取所有已安装的插件包名:

```
$names  =  Bundle::names();
```

插件包静态文件

如果你的插件包包含视图，并且希望它们能像 Javascript 和 images 文件一样在 **public** 目录被调用。方法很简单，只需要在你的插件包目录中新建一个 **public** 文件夹，然后把静态文件放到里面即可。

但是现在我们怎么把它们放到应用程序的 **public** 文件夹中呢？Laravel 框架的"Artisan"命令行工具提供了一个简单的命令可以把插件包 assets 文件夹中的文件复制到 public 目录中：

Publish bundle assets into the public directory:

```
php artisan bundle:publish
```

这条命令会在 public 目录中为 bundles 创建一个 **public/bundles** 目录。比如你的插件包叫"admin"，那么就会建立一个 **public/bundles/admin** 文件夹，里面包含了所有 admin 插件包 public 目录里的文件。

更多与插件包 assets 相关的信息可以阅读：asset management。

安装插件包

很多时候我们都需要手动安装插件包，不过"Artisan"CLI 工具提供了一种更好的方法来安装和升级插件包。Laravel 可以通过解压 ZIP 压缩包来安装插件包：

通过 Artisan 安装插件包:

```
php artisan bundle:install eloquent
```

命令执行完插件包就安装成功了，现在你可以注册它，发布它的 assets 文件了。

可以访问官方的 Laravel bundle directory 获得优秀的插件包。

升级插件包

当你升级插件包时，Laravel 会自动移除旧的插件包，然后重新安装。

通过 Artisan 升级插件包:

```
php artisan bundle:upgrade eloquent
```

Note: 在升级插件包之后,你需要重新发布插件包的 assets 文件。
重要提醒: 当插件包在升级重新安装成功之后，你必须留意插件包升级前的所有变动，你可能需要再次修改插件包的配置，修改插件包的目录等。可以用插件包的启动事件来设置它们，把类似下面的代码加入到 **application/start.php** 文件中：

Listening for a bundle's start event:

```
Event::listen('laravel.started: admin', function()

{

    Config::set('admin::file.option', true);

});
```

自动加载

目录

- 基础
- 注册目录
- 注册映射
- 注册命名空间

基础

自动加载允许你通过即用即加载的方式来加载需要的类文件，而不用每次都写繁琐的 *require* 和 *include* 语句。因此，每一次请求的执行过程都只加载必须的类，也不不要关心类的加载问题，只要需要的时候直接使用即可。

models 和 **libraries** 目录是默认注册到自动加载器中的，参见 **application/start.php** 文件。自动加载器使用类名到文件名的加载方式，所有文件名必须由小写字符构成。例如，models 目录中的 "User" 类应当对应到 "user.php" 文件。如果类是放在子目录中的话，只需让命名空间符合目录结构即可。例如 "Entities\User" 类对应到 models 目录中的 "entities/user.php" 文件。

注册目录

如上所述，models 和 libraries 目录都被默认注册到自动加载类之中了。然而，你还可以注册任意的目录，让这些目录也享受到同样的类（class）到目录的加载方式：

通过 Autoloader 类注册目录：

```
Autoloader::directories(array(

    path('app').'entities',

    path('app').'repositories',

));
```

注册映射

你可以手动将一个类映射为某个相关的代码文件，这种方式是最高效的加载类的方式：

通过 Autoloader 类注册一个类到文件的映射：

```
Autoloader::map(array(

    'User'      =>  path('app').'models/user.php',

    'Contact'  =>  path('app').'models/contact.php',
```

```
));
```

注册命名空间

很多第三方工具库都遵循 PSR-0规范。PSR-0规定类名应当和文件名一致；命名空间应当和目录结构一致。如果你使用的工具库遵循 PSR-0规范，只需注册它的根命名空间和目录即可：

注册命名空间：

```
Autoloader::namespaces(array(

    'Doctrine'  =>  path('libraries').'Doctrine',

));
```

在 PHP 内建对命名空间的支持之前，很多项目都使用下划线分隔类名的方式来标识其目录结构。如果你使用的是这种遗留的工具库，也可以很方便将其注册到自动加载类中。例如，以 SwiftMailer 为例，你可能会注意这个工具库中所有的类名都已“Swift”开头，因此，我们将“Swift”作为这一项目的根目录进行注册。

注册用下划线分隔符标识目录结构的工具库：

```
Autoloader::underscored(array(

    'Swift'  =>  path('libraries').'SwiftMailer',

));
```

错误与日志

目录

- 基本配置

- 日志
- Logger 类

基本配置

所有与日志有关的配置项都在 **application/config/errors.php** 文件中，下面让我们逐个介绍。

忽略错误

ignore 配置项包含一个错误级别数组，这些错误级别是要被 Laravel 所忽略的。这样，当脚本遇到这些错误的时候就不会停止执行。然而，当日志系统被启用的时候，这些错误是会被记录的。

错误详情

detail 配置项指定了是否让框架输出错误信息和详细的调用栈。当开发的时候，这些功能将会非常有用，建议设置为 **true** 。然而，在生产环境中应当设置为 **false** 。一旦设置为 **false** ，**application/views/error/500.php** 文件将会作为默认的视图文件，用于输出一些普通的错误消息。

日志

在 **application/config/errors.php** 文件中设置 **log** 配置项为 "true" 即可启用 Laravel 的日志功能。请用之后，为 **logger** 配置项所设定的闭包函数将会在每次错误发生时执行。这就为如何记录日志提供了灵活的方式。你甚至可以在错误发生时给开发组内的每个人发送邮件！

默认情况下，日志信息会被存储在 **storage/logs** 目录，并且是按日期创建日志文件存储的。这样就不会让所有日志信息混在一起了。

Logger 类

某些时候你可能需要使用 Laravel 的 **Log** 类来输出一些调试信息，或者只是输出一些提示性信息，下面就是如何实现：

向日志中输出一条日志消息：

```
Log::write('info', 'This is just an informational message!');
```

通过魔术方法指定日志消息的类型：

```
Log::info('This is just an informational message!');
```

错误与日志

目录

•基本配置
•日志
•Logger 类

基本配置

所有与日志有关的配置项都在 application/config/errors.php 文件中，下面让我们逐个介绍。

忽略错误

ignore 配置项包含一个错误级别数组，这些错误级别是要被 Laravel 所忽略的。这样，当脚本遇到这些错误的时候就不会停止执行。然而，当日志系统被启用的时候，这些错误是会被记录的。

错误详情

detail 配置项指定了是否让框架输出错误信息和详细的调用栈。当开发的时候，这些功能将会非常有用，建议设置为 true 。然而，在生产环境中应当设置为 false 。一旦设置为 false，application/views/error/500.php 文件将会作为默认的视图文件，用于输出一些普通的错误消息。

日志

在 application/config/errors.php 文件中设置 log 配置项为 "true" 即可启用 Laravel 的日志功能。请用之后，为 logger 配置项所设定的闭包函数将会在每次错误发生时执行。这就为如何记录日志提供了灵活的方式。你甚至可以在错误发生时给开发组内的每个人发送邮件！

默认情况下，日志信息会被存储在 storage/logs 目录，并且是按日期创建日志文件存储的。这样就不会让所有日志信息混在一起了。

Logger 类

某些时候你可能需要使用 Laravel 的 Log 类来输出一些调试信息，或者只是输出一些提示性信息，下面就是如何实现：

向日志中输出一条日志消息：

```
Log::write('info', 'This is just an informational message!');
```

通过魔术方法指定日志消息的类型:

```
Log::info('This is just an informational message!');
```

检测请求

目录

- URI 操作
- 其它辅助函数

URI 操作

获取当前请求的 URI：

```
echo  URI::current();
```

获取 URI 中的某一片段：

```
echo  URI::segment(1);
```

=如果要获取的片段不存在，则返回默认值：

```
echo  URI::segment(10,  'Foo');
```

获取完整的请求路径（URI），包括参数：

```
echo  URI::full();
```

如果你需要检测当前 URI 是否是某个特定的字符串或者以某个字符串开始，那么下面就是演示如何利用 is()

方法达成目的：

检测 URI 是否是"home"：

```
if (URI::is('home'))

{

    // The current URI is "home"!

}
```

检测当前 URI 是否以"docs/"开头：

```
if URI::is('docs/*'))

{

    // The current URI begins with "docs/"!

}
```

其它辅助函数

获取当前请求所用的方法(GET、POST、DELETE 等)：

```
echo Request::method();
```

访问 $_SERVER 全局数组：

```
echo Request::server('http_referer');
```

```

```

获取发起请求端的 IP 地址：

```
echo Request::ip();
```

检测当前请求时候是 HTTPS：

```
if (Request::secure())

{

    // This request is over HTTPS!

}
```

检测当前请求是否是 AJAX 请求：

```
if (Request::ajax())

{

    // This request is using AJAX!

}
```

检测当前请求是否是通过 Artisan 命令行发起的：

```
if (Request::cli())

{
```

```
    // This request came from the CLI!

}
```

生成 URL

目录

- 基础
- 路由 URL
- 控制器动作 URL
- 静态资源文件 URL
- URL 辅助函数

基础

获取 web 应用的根路径(base URL)：

```
$url = URL::base();
```

生成相对于根路径的 URL：

```
$url = URL::to('user/profile');
```

生成基于 HTTPS 协议的 URL：

```
$url = URL::to_secure('user/login');
```

获取当前请求的 URL：

```
$url = URL::current();
```

获取当前请求的完整 URL，包括请求参数：

```
$url = URL::full();
```

路由 URL

生成某个命名路由的 URL：

```
$url = URL::to_route('profile');
```

你可能需要生成命名路由的 URL，同时使用实际值替换其中的通配符，这很简单：

生成某个命名路由的 URL，并用实际值替换其中的通配符：

```
$url = URL::to_route('profile', array($username));
```

扩展阅读：

- 命名路由

控制器动作 URL

生成某个控制器动作的 URL：

```
$url = URL::to_action('user@profile');
```

生成某个控制器动作的 URL，并用实际值替换其中的通配符：

```
$url = URL::to_action('user@profile', array($username));
```

静态资源文件 URL

针对静态资源文件的 URL 不会包含 "application.index" 配置项所设置的值。即访问静态资源文件时是不会通过统一入口"index.php"文件的，而是直接访问实际的文件。

生成某个静态资源文件的 URL：

```
$url = URL::to_asset('js/jquery.js');
```

URL 辅助函数

下面这组全局函数也是用来生成 URL 的，和上面的 URL 类中的方法是一一对应的，但是更短小，书写更方便。

生成相对于根路径的 URL：

```
$url = url('user/profile');
```

生成静态资源文件的 URL

```
$url = asset('js/jquery.js');
```

生成某个命名路由的 URL：

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

生成某个命名路由的 URL，并用实际值替换其中的通配符：

```
$url = route('profile', array($username));
```

生成某个控制器动作的 URL：

```
$url = action('user@profile');
```

生成某个控制器动作的 URL，并用实际值替换其中的通配符：

```
$url = action('user@profile', array($username));
```

事件

目录

- 基础
- 触发事件
- 监听事件
- 事件队列
- Laravel 默认事件

基础

事件为建立松耦合应用提供了极大的灵活性，它允许扩展包参与到应用的核心执行流程中而无需修改程序代码。

触发事件

只要把你要触发的事件名告诉 **Event** 类就可以触发一个事件：

触发一个事件：

```
$responses = Event::fire('loaded');
```

注意，我们将 **fire** 方法的执行结果赋予了一个变量。这个方法将返回一个数组，数组中包含所有监听这一事件的监听器执行后的返回数据。

如果只是要获取第一条返回数据的话，可以这样：

触发一个事件并获取第一条返回值：

```
$response = Event::first('loaded');
```

注意： **first** 方法会触发所有监听某一事件的监听器，但是最后只返回第一条数据：
Event::until 方法在获取到第一条非 null 的返回数据之前将按顺序执行所有监听某一事件的监听器。

触发一个事件，并直到有一个事件监听器返回非 null 数据为止：

```
$response = Event::until('loaded');
```

监听事件

光有事件，没人监听也是白搭。下面演示为事件注册一个事件处理器：

注册一个事件处理器：

```
Event::listen('loaded', function()

{

    // I'm executed on the "loaded" event!

});
```

传递个这个方法的闭包函数会在 "loaded" 事件每次 触发时执行。

事件队列

有时候你会希望将需要触发的事件 放到“队列”里存储起来，而不要立即触发。这就需要使用 `queue` 和 `flush` 方法了。首先，向事件队列中注册一个带有唯一标识符的事件：

向事件队列注册一个事件

```
Event::queue('foo', $user->id, array($user));
```

.这个方法接受3个参数。第一个参数是事件队列的名称，第二个参数是事件的唯一标识符，第三个参数是用来传递给事件清理器的数组，这一数组携带了有用的数据。

接下来，我们为 `foo` 事件注册一个事件处理器：

注册一个事件处理器：

```
Event::flusher('foo', function($key, $user)

{

    //

});
```

注意，flusher 方法接受两个参数。第一个事存放在事件队列中的某个事件的唯一标识符，在上面的例子中就是用户 ID；第二个参数（其后还可以有更多参数）是这个事件所携带的数据。

最后，我们调用 `flush` 方法将队列中的所有事件交由事件处理器处理掉：

```
Event::flush('foo');
```

Laravel 默认事件

Laravel 框架会在执行过程中触发以下几个事件：

启动扩展包之后触发的事件：

```
Event::listen('laravel.started: bundle', function() {});
```

执行数据库查询之后触发的事件：

```
Event::listen('laravel.query', function($sql, $bindings, $time) {});
```

将数据发送给浏览器之前触发的事件：

```
Event::listen('laravel.done', function($response) {});
```

Log 类在记录日志消息之后触发的事件：

```
Event::listen('laravel.log', function($type, $message) {});
```

校验

目录

- 基础
- 校验规则
- 获取错误消息
- Validation Walkthrough
- 自定义错误消息
- 自定义校验规则

基础

Almost every interactive web application needs to validate data. For instance, a registration form probably requires the password to be confirmed. Maybe the e-mail address must be unique. Validating data can be a cumbersome process. Thankfully, it isn't in Laravel. The Validator class provides an awesome array of validation helpers to make validating your data a breeze. Let's walk through an example:

获取需要校验的数据（数组形式）：

```
$input = Input::all();
```

定义校验规则：

```
$rules = array(

    'name'    => 'required|max:50',

    'email' => 'required|email|unique:users',

);
```

创建 Validator 实例并校验数据：

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



if ($validation->fails())

{

    return $validation->errors;

}
```

With the *errors* property, you can access a simple message collector class that makes working with your error messages a piece of cake. Of course, default error messages have been setup for all validation rules. The default messages live at **language/en/validation.php**.

Now you are familiar with the basic usage of the Validator class. You're ready to dig in and learn about the rules you can use to validate your data!

校验规则

- Required
- Alpha, Alpha Numeric, & Alpha Dash
- Size
- Numeric
- Inclusion & Exclusion
- Confirmation
- Acceptance
- Same & Different
- Regular Expression Match
- Uniqueness & Existence
- Dates
- E-Mail Addresses
- URLs
- Uploads

Required

Validate that an attribute is present and is not an empty string:

```
'name' => 'required'
```

Validate that an attribute is present, when another attribute is present:

```
'last_name' => 'required_with:first_name'
```

Alpha, Alpha Numeric, & Alpha Dash

Validate that an attribute consists solely of letters:

```
'name' => 'alpha'
```

Validate that an attribute consists of letters and numbers:

```
'username' => 'alpha_num'
```

Validate that an attribute only contains letters, numbers, dashes, or underscores:

```
'username' => 'alpha_dash'
```

```

```

Size

Validate that an attribute is a given length, or, if an attribute is numeric, is a given value:

```
'name' => 'size:10'
```

Validate that an attribute size is within a given range:

```
'payment' => 'between:10,50'
```

Note: All minimum and maximum checks are inclusive.

Validate that an attribute is at least a given size:

```
'payment' => 'min:10'
```

Validate that an attribute is no greater than a given size:

```
'payment' => 'max:50'
```

Numeric

Validate that an attribute is numeric:

```
'payment' => 'numeric'
```

Validate that an attribute is an integer:

```
'payment' => 'integer'
```

Inclusion & Exclusion

Validate that an attribute is contained in a list of values:

```
'size' => 'in:small,medium,large'
```

Validate that an attribute is not contained in a list of values:

```
'language' => 'not_in:cobol,assembler'
```

Confirmation

The *confirmed* rule validates that, for a given attribute, a matching *attribute_confirmation* attribute exists.

Validate that an attribute is confirmed:

```
'password' => 'confirmed'
```

Given this example, the Validator will make sure that the *password* attribute matches the

password_confirmation attribute in the array being validated.

Acceptance

The *accepted* rule validates that an attribute is equal to *yes* or *1*. This rule is helpful for validating checkbox form fields such as "terms of service".

Validate that an attribute is accepted:

```
'terms' => 'accepted'
```

Same & Different

Validate that an attribute matches another attribute:

```
'token1' => 'same:token2'
```

Validate that two attributes have different values:

```
'password' => 'different:old_password',
```

Regular Expression Match

The *match* rule validates that an attribute matches a given regular expression.

Validate that an attribute matches a regular expression:

```
'username' => 'match:/[a-z]+/';
```

```

```

Uniqueness & Existence

Validate that an attribute is unique on a given database table:

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

In the example above, the *email* attribute will be checked for uniqueness on the *users* table. Need to verify uniqueness on a column name other than the attribute name? No problem:

Specify a custom column name for the unique rule:

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

Many times, when updating a record, you want to use the unique rule, but exclude the row being updated. For example, when updating a user's profile, you may allow them to change their e-mail address. But, when the *unique* rule runs, you want it to skip the given user since they may not have changed their address, thus causing the *unique* rule to fail. It's easy:

Forcing the unique rule to ignore a given ID:

```
'email' => 'unique:users,email_address,10'
```

Validate that an attribute exists on a given database table:

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

Specify a custom column name for the exists rule:

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

Dates

Validate that a date attribute is before a given date:

```
'birthdate' => 'before:1986-05-28';
```

Validate that a date attribute is after a given date:

```
'birthdate' => 'after:1986-05-28';
```

Note: The **before** and **after** validation rules use the **strtotime** PHP function to convert your date to something the rule can understand.

E-Mail Addresses

Validate that an attribute is an e-mail address:

```
'address' => 'email'
```

Note: This rule uses the PHP built-in *filter_var* method.

URLs

Validate that an attribute is a URL:

```
'link' => 'url'
```

Validate that an attribute is an active URL:

```
'link' => 'active_url'
```

Note: The *active_url* rule uses *checkdnsr* to verify the URL is active.

Uploads

The *mimes* rule validates that an uploaded file has a given MIME type. This rule uses the PHP Fileinfo extension to read the contents of the file and determine the actual MIME type. Any extension defined in the *config/mimes.php* file may be passed to this rule as a parameter:

Validate that a file is one of the given types:

```
'picture' => 'mimes:jpg,gif'
```

Note: When validating files, be sure to use Input::file() or Input::all() to gather the input.

Validate that a file is an image:

```
'picture' => 'image'
```

Validate that a file is no more than a given size in kilobytes:

```
'picture' => 'image|max:100'
```

获取错误消息

Laravel makes working with your error messages a cinch using a simple error collector class. After calling the *passes* or *fails* method on a Validator instance, you may access the errors via the *errors* property. The error collector has several simple functions for retrieving your messages:

Determine if an attribute has an error message:

```
if ($validation->errors->has('email'))

{

    // The e-mail attribute has errors...

}
```

Retrieve the first error message for an attribute:

```
echo $validation->errors->first('email');
```

Sometimes you may need to format the error message by wrapping it in HTML. No problem. Along with the :message place-holder, pass the format as the second parameter to the method.

Format an error message:

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

Get all of the error messages for a given attribute:

```
$messages = $validation->errors->get('email');
```

Format all of the error messages for an attribute:

```
$messages = $validation->errors->get('email', '<p>:message</p>');
```

Get all of the error messages for all attributes:

```
$messages = $validation->errors->all();
```

Format all of the error messages for all attributes:

```
$messages = $validation->errors->all('<p>:message</p>');
```

Validation Walkthrough

Once you have performed your validation, you need an easy way to get the errors back to the view. Laravel makes it amazingly simple. Let's walk through a typical scenario. We'll define two routes:

```
Route::get('register', function()

{

    return View::make('user.register');

});
```

```
Route::post('register', function()
{
    $rules = array(...);

    $validation = Validator::make(Input::all(), $rules);

    if ($validation->fails())
    {
        return Redirect::to('register')->with_errors($validation);
    }
});
```

Great! So, we have two simple registration routes. One to handle displaying the form, and one to handle the posting of the form. In the POST route, we run some validation over the input. If the validation fails, we redirect back to the registration form and flash the validation errors to the session so they will be available for us to display.

But, notice we are not explicitly binding the errors to the view in our GET route. However, an errors variable ($errors) will still be available in the view. Laravel intelligently determines if errors exist in the session, and if they do, binds them to the view for you. If no errors exist in the session, an empty message container will still be bound to the view. In your views, this allows you to always assume you have a message container available via the errors variable. We love making your life easier.

For example, if email address validation failed, we can look for 'email' within the $errors session var.

```
$errors->has('email')
```

```

```

Using Blade, we can then conditionally add error messages to our view.

```
{{ $errors->has('email') ? 'Invalid Email Address' : 'Condition is false. Can be left blank' }}
```

This will also work great when we need to conditionally add classes when using something like Twitter Bootstrap.
For example, if the email address failed validation, we may want to add the "error" class from Bootstrap to our *div class="control-group"* statement.

```
<div class="control-group {{ $errors->has('email') ? 'error' : '' }}">
```

When the validation fails, our rendered view will have the appended *error* class.

```
<div class="control-group error">
```

自定义错误消息

Want to use an error message other than the default? Maybe you even want to use a custom error message for a given attribute and rule. Either way, the Validator class makes it easy.

Create an array of custom messages for the Validator:

```
$messages = array(

    'required' => 'The :attribute field is required.',

);
```

```
$validation = Validator::make(Input::get(), $rules, $messages);
```

Great! Now our custom message will be used anytime a required validation check fails. But, what is this **:attribute** stuff in our message? To make your life easier, the Validator class will replace the **:attribute** place-holder with the actual name of the attribute! It will even remove underscores from the attribute name.

You may also use the **:other**, **:size**, **:min**, **:max**, and **:values** place-holders when constructing your error messages:

Other validation message place-holders:

```
$messages = array(

    'same'    => 'The :attribute and :other must match.',

    'size'    => 'The :attribute must be exactly :size.',

    'between' => 'The :attribute must be between :min - :max.',

    'in'      => 'The :attribute must be one of the following types: :values',

);
```

So, what if you need to specify a custom required message, but only for the email attribute? No problem. Just specify the message using an **attribute_rule** naming convention:

Specifying a custom error message for a given attribute:

```
$messages = array(

    'email_required' => 'We need to know your e-mail address!',

);
```

In the example above, the custom required message will be used for the email attribute, while the default message will be used for all other attributes.

However, if you are using many custom error messages, specifying inline may become cumbersome and messy. For that reason, you can specify your custom messages in the **custom** array within the validation language file:

Adding custom error messages to the validation language file:

```
'custom' => array(

    'email_required' => 'We need to know your e-mail address!',

)
```

自定义校验规则

Laravel provides a number of powerful validation rules. However, it's very likely that you'll need to eventually create some of your own. There are two simple methods for creating validation rules. Both are solid so use whichever you think best fits your project.

Registering a custom validation rule:

```
Validator::register('awesome', function($attribute, $value, $parameters)

{

    return $value == 'awesome';

});
```

In this example we're registering a new validation rule with the validator. The rule receives three arguments. The first is the name of the attribute being validated, the second is the value of the attribute being validated, and the third is an array of parameters that were specified for the rule.

Here is how your custom validation rule looks when called:

```
$rules = array(

    'username' => 'required|awesome',

);
```

Of course, you will need to define an error message for your new rule. You can do this either in an ad-hoc messages array:

```
$messages = array(

    'awesome' => 'The attribute value must be awesome!',

);



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

Or by adding an entry for your rule in the **language/en/validation.php** file:

```
'awesome' => 'The attribute value must be awesome!',
```

As mentioned above, you may even specify and receive a list of parameters in your custom rule:

```
// When building your rules array...



$rules = array(

    'username' => 'required|awesome:yes',
```

```
);



// In your custom rule...



Validator::register('awesome', function($attribute, $value, $parameters)

{

    return $value == $parameters[0];

});
```

In this case, the parameters argument of your validation rule would receive an array containing one element: "yes".

Another method for creating and storing custom validation rules is to extend the Validator class itself. By extending the class you create a new version of the validator that has all of the pre-existing functionality combined with your own custom additions. You can even choose to replace some of the default methods if you'd like. Let's look at an example:

First, create a class that extends **Laravel\Validator** and place it in your **application/libraries** directory:

Defining a custom validator class:

```
<?php



class Validator extends Laravel\Validator {}
```

Next, remove the Validator alias from **config/application.php**. This is necessary so that you don't end up with 2 classes named "Validator" which will certainly conflict with one another.

Next, let's take our "awesome" rule and define it in our new class:

Adding a custom validation rule:

```
<?php



class Validator extends Laravel\Validator {



    public function validate_awesome($attribute, $value, $parameters)

    {

        return $value == 'awesome';

    }



}
```

Notice that the method is named using the **validate_rule** naming convention. The rule is named "awesome" so the method must be named "validate_awesome". This is one way in which registering your custom rules and extending the Validator class are different. Validator classes simply need to return true or false. That's it!

Keep in mind that you'll still need to create a custom message for any validation rules that you create. The method for doing so is the same no matter how you define your rule!

文件操作类

目录

- 读取文件
- 写入文件
- 上传文件

- 文件拓展名
- 检测文件类型
- 获取 MIME 类型
- 复制目录
- 删除目录

读取文件

读取指定文件内容:

```
$contents = File::get('path/to/file');
```

写入文件

把内容写入文件：

```
File::put('path/to/file', 'file contents');
```

追加内容:

```
File::append('path/to/file', 'appended file content');
```

上传文件

Moving a $_FILE to a permanent location:

```
Input::upload('picture', 'path/to/pictures', 'filename.ext');
```

```

```

Note: You can easily validate file uploads using the Validator class.

文件拓展名

获取拓展名:

```
File::extension('picture.png');
```

检测文件类型

确保是指定类型的文件:

```
if (File::is('jpg', 'path/to/file.jpg'))

{

    //

}
```

文件操作类的 **is** 方法并不是仅仅简单的检查文件拓展名，它会使用 PHP 的 Fileinfo 拓展来读取文件内容，以确保取得正确的 MIME 类型。

Note: 你可以在 **application/config/mimes.php** 中为 **is** 方法定义添加任何默认允许的拓展名。
Note: 使用函数前需安装 Fileinfo 拓展，更多相关信息可以查看：PHP Fileinfo page。

获取 MIME 类型

通过拓展名获取 MIME 类型：

```
echo  File::mime('gif');
```

Note: 这个函数只会简单的返回在 **application/cofig/mimes.php** 中被定义过的 MIME 类型。

复制目录

通过递归复制一个目录到指定目录：

```
File::cpdir($directory,  $destination);
```

移除目录

递归删除目录：

```
File::rmdir($directory);
```

字符串操作

目录

- 大小写
- 限定单词或字符个数
- 生成随机字符串
- 单数 和 复数
- 标语

大小写

Str 类包含3个用来操作字符串大小写的实用方法： **upper**、**lower** 和 **title**。PHP 中也有很多类似的函数，例如 strtoupper、strtolower、 ucwords 。但是 **Str** 类中的方法更智能，这是因为 **Str** 类中的方法可以自动处理 UTF-8 字符串 ，前提是 PHP 中安装并且启用了 mbstring 多字节字符串扩展模块。

```
echo Str::lower('I am a string.');



echo Str::upper('I am a string.');



echo Str::title('I am a string.');
```

限定单词或字符个数

限制字符串中字符的个数：

```
echo Str::limit($string, 10);

echo Str::limit_exact($string, 10);
```

限制字符串中单词的个数（针对英文）：

```
echo Str::words($string, 10);
```

生成随机字符串

生成只包含字母和数字的随机字符串：

```
echo  Str::random(32);
```

生成只包含字母的随机字符串：

```
echo  Str::random(32,  'alpha');
```

单数 和 复数

String 类包含用来对单词进行单复数相互转换的方法：

获取某个单词的复数形式：

```
echo  Str::plural('user');
```

获取某个单词的单数形式：

```
echo  Str::singular('users');
```

如果给定的数值大于1，则返回给定单词的复数形式：

```
echo  Str::plural('comment',  count($comments));
```

标语

生成一个标语字符串，并且符合 URL 标准，即可以作为 URL 的一部分：

```
return Str::slug('My First Blog Post!');
```

生成一个标语字符串，并且符合 URL 标准，单词之间的空格用指定的连接符替换：

```
return Str::slug('My First Blog Post!', '_');
```

本地化

目录

- 基础
- 获取一条翻译文本
- 占位符与替换

基础

本地化就是将你的应用翻译成不同语言的过程。**Lang** 类提供了简化的机制用以组织和获取多语言应用所需的文本。

所有的翻译文件都存储在 **application/language** 目录。在 **application/language** 目录中，你需要为每门不同的语言创建一个目录，例如，你的应用有 English 和 Spanish 语言版本，你就需要在 **language** 目录下创建 **en** 和 **sp** 两个目录。

为每门语言所创建的目录下可以存放多个翻译文件，每个翻译文件只是返回一个存储字符串的数组，数组中包含了这门语言所对应的翻译文本。实际上，翻译文件的结构和配置文件是相同的。例如，在 **application/language/en** 目录下，你可以创建一个名为 **marketing.php** 的文件，文件内容如下：

创建一个翻译文件：

```
return array(
```

```
    'welcome' => 'Welcome to our website!',



);
```

接下来，你需要在 **application/language/sp** 目录下创建一个相同名称的 **marketing.php** 文件，这个文件包含如下的内容：

```
return array(



    'welcome' => 'Bienvenido a nuestro sitio web!',



);
```

很好，现在你就学会了如何设置翻译文件和语言目录了。

获取一条翻译文本

获取一条翻译文本：

```
echo Lang::line('marketing.welcome')->get();
```

使用 "__" 快捷函数获取一条翻译文本：

```
echo __('marketing.welcome');
```

```
```

注意到点号分隔 "marketing" 与 "welcome" 了吗？点号之前的部分对应到语言文件，点号之后的部分对应到文件中的某个字符串。

需要获取非默认语言的翻译文本怎么办？小菜一碟，只要在 **get** 方法中指定语言名称即可：

获取某门语言的一条翻译文本：

```
echo Lang::line('marketing.welcome')->get('sp');
```

占位符与替换

Now, let's work on our welcome message. "Welcome to our website!" is a pretty generic message. It would be helpful to be able to specify the name of the person we are welcoming. But, creating a language line for each user of our application would be time-consuming and ridiculous. Thankfully, you don't have to. You can specify "place-holders" within your language lines. Place-holders are preceded by a colon:现在让我们拿欢迎信息练练手吧。"Welcome to our website!" 是经常出现的一条信息，如果这条信息里面能出现被欢迎人的名字就更好了。但是，为每个用户创建一条文本的话就太费时间了，而且也太冗余，幸好有更方便的办法。你可以在这条消息中插入一个“替换符”。注意，替换符前面必须加一个冒号：

创建一条包含占位符的翻译条目：

```
'welcome' => 'Welcome to our website, :name!'
```

获取一条翻译条目，并传递参数用以替换占位符：

```
echo Lang::line('marketing.welcome', array('name' => 'Taylor'))->get();
```

用"__"函数获取一条翻译条目，并传递参数用以替换占位符：

```
echo __('marketing.welcome', array('name' => 'Taylor'));
```

IOC 容器

- 定义
- 注册对象
- 提取对象

定义

简单来说，IoC 容器就是创建对象的一种方式。你可以用它来定义复杂对象的创建，将来使用的时候，就可以仅用一行代码完成对象的创建了。你还可以在类（class）、控制器（controller）中用它来实现依赖“注入”功能。

IoC 容器能协助你把应用整理的灵活、容易测试。之所以这么说，是因为你可以随时用新的实现同样接口的类替换掉以前注册的类，还可以将你要测试的代码与外部的依赖代码相隔离，可以参考 stubs and mocks。

注册对象

在 IoC 容器中注册一个提取器（resolver）：

```
IoC::register('mailer', function()

{

    $transport = Swift_MailTransport::newInstance();



    return Swift_Mailer::newInstance($transport);

});
```

```

```

很好！现在我们已经在容器中为 SwiftMailer 注册了一个提取器（resolver）。但是，如果我们不想让容器每次都创建一个新的 SwiftMailer 实例该怎么办呢？就是说，我们只想让容器每次都返回头一次初始化之后的实例。好吧，那就告诉容器我们注册的是一个单例（singleton）：

在容器中注册一个单例（singleton）：

```
IoC::singleton('mailer', function()

{

    //

});
```

你还可以将一个已经存在的对象作为单例（singleton）注册到容器中。

在容器中注册一个已经存在的实例：

```
IoC::instance('mailer', $instance);
```

提取对象

容器中已经注册了 SwiftMailer 对象了，现在我们可以使用 IoC 类的 **resolve** 方法提取它：

```
$mailer = IoC::resolve('mailer');
```

注： 你还可以参考 在容器中注册控制器（controller）。

ARTISAN 指令集

目录

- 帮助
- 应用设置
- Sessions
- 迁移
- 扩展包
- 任务
- 单元测试
- 路由
- Application Keys
- 命令行选项

帮助

描述	指令
查看 artisan 所支持的所有指令。	`php artisan help:commands`

应用设置 (更多信息)

描述	指令
生成 application key。 只有 **config/application.php** 文件中的 application key 配置项为空时，才会自动生成 application key。	`php artisan key:generate`

基于数据的 Session (更多信息)

描述	指令
创建 Session 数据表。	`php artisan session:table`

Migrations (更多信息)

描述	指令
创建迁移数据表。	`php artisan migrate:install`
创建一个迁移。	`php artisan migrate:make create_users_table`
为某个扩展包创建一个迁移。	`php artisan migrate:make bundle::tablename`
执行所有的迁移。	`php artisan migrate`
执行 application 目录下的所有迁移。	`php artisan migrate application`
执行某个扩展包下的所有迁移。	`php artisan migrate bundle`
回滚最后一次执行的迁移。	`php artisan migrate:rollback`
回滚前面执行的所有迁移。	`php artisan migrate:reset`

扩展包 (更多信息)

描述	指令
安装一个扩展包	`php artisan bundle:install eloquent`
升级一个扩展包	`php artisan bundle:upgrade eloquent`
升级所有扩展包	`php artisan bundle:upgrade`
发布某个扩展包内的所有资源文件	`php artisan bundle:publish bundle_name`
发布所有扩展包的资源文件	`php artisan bundle:publish`

注意： 安装扩展包之后，你需要手动注册扩展包

任务 (更多信息)

描述	指令
执行某个任务	`php artisan notify`
执行某个带参数的任务	`php artisan notify taylor`
执行某个任务中的一个指定的方法	`php artisan notify:urgent`
R 执行某个扩展包内容的某个任务	`php artisan admin::generate`
执行某个扩展包内的某个任务的指定的方法	`php artisan admin::generate:list`

单元测试 (更多信息)

描述	指令
执行测试指令	`php artisan test`
执行扩展包内的测试	`php artisan test bundle-name`

路由 (更多信息)

描述	指令
执行一个路由	`php artisan route:call get api/user/1`

注意： 你可以用 post、put、delete 等代替 get 方法。

Application Keys

描述	指令
生成 application key	`php artisan key:generate`

注意： 你可以通过参数的形式指定 application key 的长度。

命令行选项

描述	指令
设置 Laravel 环境	`php artisan foo --env=local`
设置默认数据库连接	