Laravel 中文官方文档 4.2

版本:4.2

来源:www.golaravel.com

整理:飞龙

日期:2015.4.10

序言

简介

从哪里开始

学习一门框架既令人生畏，又令人兴奋。为了提供平稳的学习体验，我们尝试为Laravel创建了清晰简明的文档。下面列出了推荐的阅读顺序：

- 安装 和 配置
- 路由
- 请求与输入
- 视图与响应
- 控制器

通过阅读这些文档，你应该能够掌握Laravel框架对于基本请求/响应的处理。下一步你可能希望阅读 配置数据库, 查询构造器, 以及 Eloquent ORM。或者你想要了解 身份验证和安全 以便在你的应用中实现用户登录功能。

Laravel哲学

Laravel是一套web应用开发框架，它具有富于表达性且简洁的语法。我们相信，开发过程应该是愉悦、创造性的体验。Laravel努力剔除开发过程中的痛苦，因此我们提供了验证（authentication）、路由（routing）、session和缓存（caching）等开发过程中经常用到的工具或功能。

Laravel的目标是给开发者创造一个愉快的开发过程，并且不牺牲应用的功能性。快乐的开发者才能创造最棒的代码！为了这个目的，我们博取众框架之长处集中到Laravel中，这些框架甚至是基于Ruby on Rails、ASP.NET MVC、和Sinatra等开发语言或工具的。

Laravel是易于理解并且强大的，它提供了强大的工具用以开发大型、健壮的应用。杰出的IoC、数据库迁移工具和紧密集成的单元测试支持，这些工具赋予你构建任何应用的能力。

译者：王赛（Bootstrap中文网）

Laravel 快速入门

安装

通过 Laravel 安装器

首先，通过 Composer 下载 Laravel 安装器。

```
composer global require "laravel/installer=~1.1"
```

请确保把 `~/.composer/vendor/bin` 路径添加到 PATH 环境变量里，这样`laravel` 可执行文件才能被命令行找到，以后您就可以在命令行下直接使用 `laravel` 命令.

安装成功后，可以使用命令 `laravel new` 在您指定的目录下创建一份全新安装的 Laravel。例如，`laravel new blog` 将会在当前目录下创建一个叫 `blog` 的目录，此目录里面存放着全新安装的 Laravel 以及其依赖的工具包。这种安装方法比通过 Composer 安装要快许多。

通过 Composer

Laravel 框架使用 composer 来执行安装及管理依赖。如果还没有安装它的话，请先从 安装 Composer 开始吧。

安装之后，您可以通过终端执行下列命令来安装 Laravel：

```
composer create-project laravel/laravel your-project-name --prefer-dist
```

这个命令会下载并安装一份全新的 Laravel 存放在指定的 `your-project-name` 的目录中。

手动安装

手动安装 Laravel，可以直接从 Github 上的 Laravel Respoitory 下载一份打包的代码，解压缩，然后在解压后的根目录里，执行 `composer install` 即可，这个命令会把框架所需要的依赖下载完整。

权限配置

在安装 Laravel 之后，确保 Web 服务器有写入 `app/storage` 目录的权限。详情请见安装文档说明。

运行 **Laravel**

一般而言，您需要一个 Web 服务器（比如: Apache 或是 Nginx）来运行您的 Laravel 应用。如果您是使用 PHP 5.4 以上版本，为了方便，可以使用 PHP 内建的开发服务器，只需要通过使用 Laravel 的 Artisan 命令 来执行 `serve` 即可：

```
php artisan serve
```

目录结构

安装完框架后，我们来了解熟悉一下项目的目录结构。`app` 目录里面包含了 `views（视图）`，`controllers（控制器）`，还有 `models（模型）` 等目录。您的应用程序大多数的代码都会在这个目录中，需要注意的还有存放相关配置文件的目录 `app/config` 。

本地开发环境

过去要在本机上配置一个本地的 PHP 开发环境是非常让人头痛的事情。除了要安装正确的 PHP 版本、对应的扩展包，还有一些所需的组件，都是一个大工程。很多情况下因为这些繁琐的配置问题，导致我们直接放弃了尝试要做的事，非常的可惜。

为了解决这状况，使用 Laravel Homestead 吧。Homestead 是由 Laravel 和 Vagrant 所设计的虚拟机。而 Homestead Vagrant 里封装了建立一个完整 PHP 应用所需的所有软件。因此可以在瞬间创建一个虚拟化的、独立不受干扰的开发环境。特别适用于团队开发环境的统一。下面列出包装在 Homestead 里的软件：

- Nginx
- PHP 5.6
- MySQL
- Redis
- Memcached
- Beanstalk

Homestead 依赖于 VirtualBox 和 Vagrant ， 所以您需要先安装他们。两个软件都有各平台的图形化安装界面。请参阅 Homestead 说明 进行了解。

Laravel 的社区特别推荐使用 Homestead 来开发应用，主要有以下优势：

1. 安装部署简单, 快速使用, 并支持各种主流系统;
2. 统一开发环境, 避免了团队开发时, 各种 `运行系统`, `软件发行版本`, `软件配置` 的不一致所带来不必要的复杂性;
3. 能使开发环境最大程度上跟线上生产环境一致;

路由

首先, 我们先创建第一个路由。在 Laravel 中, 最简单的路由是通过匿名函数来实现的。打开 `app/routes.php` 文件, 在文件的最下方添加如下代码:

```
Route::get('users', function()
{
    return 'Users!';
});
```

现在, 在浏览器中输入 `/users`, 您应该会看到页面出现 `Users!`。简单吧, 您已经建立了第一个路由。

路由也可以指向一个控制器类和动作方法。例如:

```
Route::get('users', 'UserController@getIndex');
```

以上代码声明 `/users` 路由的请求应该使用 `UserController` 类的 `getIndex` 方法。查看更多关于控制器路由的信息, 请查阅控制器。

建立视图

接下来, 我们要创建视图来显示我们的用户数据。视图以 HTML 代码存放在 `app/views` 的目录中, 后缀名一般为 `.blade.php`。我们先来创建两个视图文件: `layout.blade.php` 和 `users.blade.php`。首先, 我们建立`layout.blade.php` 文件:

```
<html>
    <body>
        <h1>Laravel Quickstart</h1>

        @yield('content')
    </body>
</html>
```

接下来, 我们建立 `users.blade.php` 文件:

```
@extends('layout')

@section('content')
    Users!
@stop
```

这里有些语法或许让您感到陌生。因为我们使用的是 Laravel 的模板系统：Blade。Blade 非常快，仅需要通过少量的正则表示式来把模板文件编译成 PHP 代码。Blade 提供了强大的功能，例如模板的继承，还有一些常用的 PHP 控制结构语法，如 `if` 和 `for`。更多信息请查阅 Blade。

现在，我们已经有了自己的视图，让我们回到 `/users` 路由。我们改用视图来替代直接输出的 `Users!`：

```
Route::get('users', function()
{
    return View::make('users');
});
```

太棒了！现在您已经成功地建立了一个继承自 layout 的简单视图。接下来，我们来学习一下数据库层。

建立迁移文件

我们使用 Laravel 的迁移 (migration) 系统来建立数据库表以保存我们的数据。迁移记录着数据库的改变历史，可以通过版本控制的方式来更好的让团队开发保持一致性。

首先，我们要配置数据库连接。您可以在 `app/config/database.php` 文件中配置所有的数据库连接信息。Laravel 默认使用 MySQL，所以您必须将相关的数据库连接信息填入其中。您也可以更改 `driver` 选项为 `sqlite`，如此 Laravel 就会使用放置在 `app/database` 里的 SQLite 数据库。

接下来，我们来创建迁移文件，我们使用 Artisan CLI来创建。在项目的根目录下，通过终端执行下列命令：

```
php artisan migrate:make create_users_table
```

然后，在 `app/database/migrations` 目录下就会产生对应的迁移文件。文件中会有一个包含了两个方法 `up` 和 `down` 的类。在 `up` 方法中，您必须写上您要对您的数据库表做哪些更动，而在 `down` 的方法里，您只要写上对应的回滚代码。

我们定义一个迁移文件如下：

```
public function up()
{
    Schema::create('users', function($table)
    {
        $table->increments('id');
        $table->string('email')->unique();
        $table->string('name');
        $table->timestamps();
    });
}
```

```
public function down()
{
    Schema::drop('users');
}
```

然后继续在项目的根目录下，通过终端执行 `migrate` 命令来执行迁移动作。

```
php artisan migrate
```

如果您想回滚迁移，您可以执行以下命令

```
php artisan migrate:rollback
```

现在我们已经建好了数据库表了，开始放些数据进去吧。

Eloquent ORM

Laravel 提供了很棒的 ORM：Eloquent。如果您曾经使用过 Ruby on Rails 框架，那您将会觉得 Eloquent 很熟悉，因为它遵循着 ActiveRecord ORM 风格的数据库互动模式。

首先，我们先来定义一个模型 (model)。一个 Eloquent 模型可以用来查询关联的数据库表，以及表内的某一行。模型通常存放在 `app/models` 目录中。让我们先来在这目录里创建一个 `User.php` 的模型文件：

```
class User extends Eloquent {}
```

注意，我们并未告诉 Eloquent 使用哪个表。Eloquent 有多种惯例，一种就是使用模型的复数形态作为该模型的数据库表名称，非常方便，这是 Laravel 的开发规范。

使用您喜欢的数据库管理工具，插入几条数据到 `users` 数据库表，我们将使用 Eloquent 来取得这些数据并且传递到视图中去。

现在我们修改我们的 `/users` 路由如下：

```
Route::get('users', function()
{
    $users = User::all();

    return View::make('users')->with('users', $users);
});
```

在以上代码中，首先，`User` 模型里的 `all` 方法会取得 `users` 表里的所有记录。接下来，我们通过 `with` 方法将这些记录传递到视图中。`with` 方法接受一个键和与其对应的值，这样这对键值就可以在视图中使用了。

显示数据

现在，通过 `view` 下的 `with` 方法，视图中已经可以获取到 `users` 变量了，我们可以显示出来，如下：

```
@extends('layout')

@section('content')
    @foreach($users as $user)
        <p>{{ $user->name }}</p>
    @endforeach
@stop
```

您会发现没有看到任何 `echo` 语句。当使用 Blade 时，您可以使用两个大括号来输出数据。接下来，通过访问 `/users` 路由，就能看到用户数据了。

这仅仅只是开始。在这个教程中，您已经了解了 Laravel 基础部分，但是还有更多令人兴奋的东西等着您学习。继续阅读文件且更深入的了解 Eloquent 和 Blade 的强大特性。或许，您也更有兴趣去了解 队列 和 单元测试。

部署应用程序

Laravel 的其中一个目标就是让 PHP 应用程序开发从下载到部署都非常的轻松，而 Laravel Forge 提供了一个简单的方式去部署您的 Laravel 应用到服务器上。Forge 可以配置并供应在 DigitalOcean、Linode、Rackspace 和 Amazon EC2 上的机器群。如同 Homestead 一样，所有必须的最新版软件都已安装在内：Nginx、PHP 5.5、MySQL、Postgres、Redis、Memcached 等等。Forge 的 “快速部署” 可以让您在每次发布更新至 GitHub 或是 Bitbucket 时自动部署应用。

更重要的是，Forge 能帮助您配置 queue workers、SSL、Cron jobs、子域名等等。更多的信息请参阅 Forge 网站。

版本说明

- [Laravel 4.2](#laravel-4.2)
- [Laravel 4.1](#laravel-4.1)

Laravel 4.2

通过在4.2版本的安装目录下运行命令“php artisan changes”来获得此版本完整的变更列表, 也可以查看Github上的变更文件(https://github.com/laravel/framework/blob/4.2/src/Illuminate/Foundation/changes.json)。这些更改记录只包含当前版本主要的功能改进和变更。

> 注意: 在4.2版本的发布周期中, 很多小的bug修复和功能改进被合并进了Laravel 4.1的各个发布版本中。所以, 也一定要检查Laravel 4.1的变更列表!

最低要求**PHP 5.4**

Laravel 4.2 要求 PHP 5.4 或更高版本. 这个PHP版本的升级需求使得我们能够使用PHP的新特性, 例如, 为Laravel Cashier(/docs/billing)等工具提供的更具表现力的接口。与PHP 5.3相比, PHP 5.4 在速度和执行效率上也有显著的提高。

Laravel Forge

Laravel Forge是一个新的基于web的应用程序, 它提供一种简单的方式来创建和管理你自己云端的PHP服务, 这些服务包括Linode、DigitalOcean、Rackspace和Amazon EC2。Forge支持Nginx配置自动化, 访问SSH key, Cron job的自动化, 通过NewRelic或Papertrail进行服务监控, “Push To Deploy”, 配置Laravel队列工人, 此外, Forge提供最简单且最经济的方式来运行你的所有Laravel应用程序。

默认情况下, Laravel 4.2安装目录下的文件“app/config/database.php”是用来配置Forge的, 使得新的应用能更方便的部署到这个平台上来。

在Forge的官方网站(https://forge.laravel.com)上, 可以找到更多的有关Laravel Forge的信息。

Laravel Homestead

Laravel Homestead是一个官方的Vagrant环境, 它被用来开发强健的Laravel和PHP应用程序。在箱子被打包派发之前, 绝大部分的箱子供应需求会被处理, 它允许箱子极其快速的启动。Homestead包括Nginx 1.6、PHP 5.5.12、MySQL、Postgres、Redis、Memcached、Beanstalk、Node、Gulp、Grunt和Bower。Homestaed包含一个简单的配置文件“Homestead.yaml”, 它能在单个

箱子里管理多个Laravel应用程序。

默认情况下，Laravel 4.2安装目录包含一个配置文件“app/config/local/database.php”，它是用来配置使用箱子以外的Homestead数据库的，这使得Laravel初始化安装目录和配置更加方便。

官方文档已经包括Homestead文档（/docs/homestead）。

Laravel Cashier

Laravel Cashier是一个简单的、富于表现力的库，它用来管理条形码的订阅计费。随着Laravel 4.2的引入，尽管安装组件本身仍然是可选的，但是我们在Laravel主文档里包含了Cashier文档。这个版本的Cashier修复了很多bug，支持多币种，并兼容最新的条形码API。

守护进程队列工人（**Daemon Queue Workers**）

Artisan的“queue:work”命令现在支持“--daemon”选项，它用来启动一个“守护进程模式”的工人，这个模式使得工人在不需要重启框架的情况下继续工作。这会显著的减少CPU的使用率，其代价只是使得应用程序的部署过程稍显复杂。

在队列文档（/docs/queues#daemon-queue-workers）里能找到更多的队列工人相关的信息。

邮件**API**驱动

Laravel 4.2为“邮件”功能引入了新的Mailgun和Mandrill接口驱动程序。对很多应用程序来说，比起SMTP方式，此接口提供了一种更快和更可靠发送电子邮件的方法。新的驱动程序采用Guzzle 4 HTTP库。

软删除特性（**Soft Deleting Traits**）

一种为“软删除”和其它“全局范围”而生的更干净的架构通过PHP 5.4的特性被引入进来。这种新架构考虑到了更简单的构建全局特性等功能，也可考虑到了一个更干净的框架本身的关注点分离问题。

在文档（/docs/eloquent#soft-deleting）里能找到更多的关于“软删除特性”的信息。

方便的认证和提醒特性（**Convenient Auth & Remindable Traits**）

现在，Laravel 4.2的安装目录默认使用简单的特性来包含认证和密码提醒用户接口所需要的属性。这提供了一个更干净的默认的箱子之外的“用户”模型文件。

简单的分页

一个新方法“simplePaginate”被加入到查询和Eloquent构建器里，当在分页页面里使用简单的“上一个”和“下一个”链接时，使用此方法查询就会更高效。

迁移确认（**Migration Confirmation**）

现在，在生产中的破坏性操作将被要求确认。使用“--force”选项的迁移命令可以强制执行而不会有任何提示。

Laravel 4.1

全部变更列表

通过在4.1版本的安装目录下运行命令“php artisan changes”来获得此版本完整的变更列表，也可以查看Github上的变更文件（https://github.com/laravel/framework/blob/4.1/src/Illuminate/Foundation/changes.json）。这些更改记录只包含当前版本主要的功能改进和变更。

新**SSH**组件

一个全新的“SSH”组件被引入到此版本中。其特性允许你很容易的SSH到远程服务器并运行命令。想了解更多信息，请查阅SSH组件文档（/docs/ssh）。

新的“php artisan tail”命令使用了新SSH组件。更多信息，请查阅“tail”命令文档（http://laravel.com/docs/ssh#tailing-remote-logs）。

Boris In Tinker

命令“php artisan tinker”使用的是Boris REPL（https://github.com/d11wtq/boris），首先你的系统得支持Boris REPL。要使用这个特性必须安装PHP扩展“readline”和“pcntl”。如果你没有这两个扩展，将使用4.0版本的shell.

Eloquent改进点

一种新的“hasManyThrough”关系已经被加入到Eloquent里。要学习怎样使用，请查阅Eloquent文档（/docs/eloquent#has-many-through）。

一个新方法“whereHas”也已经被引入，此方法允许基于关系约束的索引模型，相关文档（/docs/eloquent#querying-relations）。

数据库读写连接

在数据库层有可供使用的自动处理单独读写的连接，包括队列构建器和Eloquent。更多相关信息，请查阅文档（/docs/database#read-write-connections）。

队列优先权

现在支持通过传递给命令“queue:listen”一组以逗号分隔的列表来设定队列的优先级。

失败的队列工作处理

现在，在命令“queue:listen”中使用“--tries”选项时，队列功能会自动处理失败的工作。在队列文档（/docs/queues#failed-jobs）里可以找到处理失败工作的更多信息。

缓存标记

缓存章节（sections）已经被标记（tags）取代。缓存标记允许你把多个标记赋值给一个缓存项，也允许把所有的缓存项赋值给一个标记。在缓存文档（/docs/cache#cache-tags）里可以找到使用缓存标记相关的更多信息。

灵活的密码提醒

密码提醒引擎已经被改成当验证密码的时候为开发者提供很大的灵活性，引擎会闪存状态信息到会话里。有关使用增强的密码提醒的更多信息，请查阅文档（/docs/security#password-reminders-and-reset）。

改善的路由引擎

Laravel 4.1以完全重写的路由层为特色。虽然接口是相同的，但是路由注册比4.0版本要快100%。整个引擎被大大的简化了，而且依赖于Symfony的路由被最小化到路由表达式的编译里了。

改善的会话引擎

在这个版本中，我们也引入了一个全新的会话引擎。类似于路由的改进，新的会话层更简洁更快。我们不再使用Symfony（PHP的）会话处理功能，而是使用一种更简单更容易维护的定制的解决方案。

Doctrine DBAL

如果你在迁移（migrations）中用到了“renameColumn”方法，你需要在“composer.json”文件中添加“doctrine/dbal”依赖。这个包不再默认包含在Laravel中。

升级指南

从**4.1**升级到**4.2**

PHP 5.4+

Laravel 4.2需要PHP 5.4.0或更高版本。

加密的默认设置

在配置文件“app/config/app.php”里添加一个新的选项“cipher”。此选项的值应该是“MCRYPT_RIJNDAEL_256”。

```
'cipher' => MCRYPT_RIJNDAEL_256
```

这个设置可以用来控制Laravel加密功能使用默认密钥。

> **Note:** In Laravel 4.2, the default cipher is `MCRYPT_RIJNDAEL_128` (AES), which is considered to be the most secure cipher. Changing the cipher back to `MCRYPT_RIJNDAEL_256` is required to decrypt cookies/values that were encrypted in Laravel <= 4.1

软删除模型现在的使用特性

如果你正在使用软删除模型，“softDeletes”属性已经被删除了。现在你应该使用“SoftDeletingTrait”属性，像下面这样：

```
use Illuminate\Database\Eloquent\SoftDeletingTrait;

class User extends Eloquent {
    use SoftDeletingTrait;
}
```

你还应该手动添加“deleted_at”列到“dates”属性上：

```
class User extends Eloquent {
    use SoftDeletingTrait;

    protected $dates = ['deleted_at'];
}
```

所有软删除操作的API保持不变。

> **Note:** The `SoftDeletingTrait` can not be applied on a base model. It must be used on an actual model class.

重命名的视图/分页环境

如果你直接的引用了“Illuminate\View\Environment”类或“Illuminate\Pagination\Environment”类，升级你的代码，用“Illuminate\View\Factory”和“Illuminate\Pagination\Factory”来替代。原来的两个类已经被重命名,以更好地反映其功能。

分页呈现时的额外参数

如果你扩展了“Illuminate\Pagination\Presenter”类，它的抽象方法“getPageLinkWrapper”的签名已经被改变了，添加了“ref”参数：

```
abstract public function getPageLinkWrapper($url, $page, $rel = null);
```

Iron.Io Queue Encryption

If you are using the Iron.io queue driver, you will need to add a new `encrypt` option to your queue configuration file:

```
'encrypt' => true
```

从小于等于**4.1.x**升级到**4.1.29**

Laravel 4.1.29改善了所有的数据库驱动程序的列引用。当模型“不”使用“fillable”属性时，它保护您的应用程序免受某个mass assignment漏洞。如果你在模型上使用了“fillable”属性来防止mass assignemnt漏洞，你的应用程序就是不易受攻击的。然而，如果你正在使用“guarded”，同时传递了一个用户控制的数组给“update”或“save”类型的函数，你应该立即升级到“4.1.29”，因为你的应用程序可能面临“mass assignment”漏洞的风险。

要升级到Laravel 4.1.29，简单的执行“composer update”命令即可。这个版本中没有引入重大的改变。

从小于等于**4.1.25**升级到**4.1.26**

Laravel 4.1.26为cookies引入了安全方面的改进。在此更新之前，如果一个cookie被另一个恶意用户劫持，这个cookie将长期有效，即使此账户真正的所有者进行了重置密码、退出登录等操作.

这项改变需要在你的数据表“users”（或等价的表）添加一个新的列“remember_token”。在这项改变之后，每次用户登录你的应用程序时都会被分配给一个新的token。当此用户从应用程序退出时，token也将被更新。这项改变的意义是：当cookie被劫持，简单的退出应用程序也会使cookie失效。

升级路线

首先，添加一个新的可为空的列“remember_token”到你的“users”表里，其类型为VARCHAR(100)，或TEXT，或等价的类型。

接下来，如果你使用了Eloquent认证驱动，用下面三个方法来更新你的“User”类：

```
public function getRememberToken()
{
    return $this->remember_token;
}

public function setRememberToken($value)
{
    $this->remember_token = $value;
}

public function getRememberTokenName()
{
    return 'remember_token';
}
```

> 注意：这项改变将使所有存在的session失效，所以，所有的用户将被迫在你的应用程序上重新认证。

包维护者

两个新方法被加入到了“Illuminate\Auth\UserProviderInterface”接口里。默认的驱动里可以找到实现的示例：

```
public function retrieveByToken($identifier, $token);

public function updateRememberToken(UserInterface $user, $token);
```

“Illuminate\Auth\UserInterface”接口也添加了“升级路线”里描述的三个新方法。

从4.0升级到4.1

升级你的**Composer**依赖

要升级你的应用程序到Laravel 4.1，把“composer.json”文件里的“laravel/framework”的版本改

成“4.1.*”。

替换文件

用仓库里的最新副本替换你的“public/index.php”文件（https://github.com/laravel/laravel/blob/master/public/index.php）。

用仓库里的最新副本替换你的“artisan”文件（https://github.com/laravel/laravel/blob/master/artisan）。

添加配置文件和选项

更新你的配置文件“app/config/app.php”里的“aliases”和“providers”数组。在文档（https://github.com/laravel/laravel/blob/master/app/config/app.php）里能找到这两个数组更新过的值。一定要把你自己的定制和包服务加回到providers/aliases数组里。

从仓库里添加新的文件“app/config/remote.php”（https://github.com/laravel/laravel/blob/master/app/config/remote.php

在你的文件“app/config/session.php”里添加新的配置项“expire_on_close”，默认值应该是“false”。

在你的文件“app/config/queue.php”里添加新的配置章节“failed”。下面是此章节的默认值：

```
'failed' => array(
    'database' => 'mysql', 'table' => 'failed_jobs',
),
```

（可选）在你的文件“app/config/view.php”里把配置项“pagination”更新为“pagination::slider-3”。

控制器更新

如果“app/controllers/BaseController.php”文件的顶部有使用声明，把“use Illuminate\Routing\Controllers\Controller;”改为“use Illuminate\Routing\Controller;”。

密码提醒更新

为了更加灵活，密码提醒被彻底修改了。通过运行Artisan命令“php artisan auth:reminders-controller”，你可以检查新的存根控制器。你也可以浏览更新文档（/docs/security#password-reminders-and-reset），并按照文档来更新你的应用程序。

更新你的语言文件“app/lang/en/reminders.php”，使其与文件（https://github.com/laravel/laravel/blob/master/app/lang/en/reminders.php）相匹配。

环境检测更新

由于安全的原因，URL域不再被用来检测你的应用程序环境。这些值是容易欺骗的，并允许攻击者修改请求环境。你应该把你的环境检测转换成机器主机名（在命令行界面执行“hostname”命令，适用于Mac, Linux, 和Windows）。

更简单的日志文件

Laravel现在生成一个单独的日志文件“app/storage/logs/laravel.log”。然而，你还可以在文件“app/start/global.php”里配置此行为。

删除重定向末尾斜杠

在你的文件“bootstrap/start.php”里，删除此句调用“$app->redirectIfTrailingSlash()”，这个方法不再需要了，因为它的功能被框架里的“.htaccess”文件负责了。

接下来，把你的Apache的“.htaccess”文件替换成新的（https://github.com/laravel/laravel/blob/master/public/.htaccess），这个文件是用来处理尾部斜杠的。

当前路由访问

现在通过“Route::current()”来访问当前路由，替换掉原来的“Route::getCurrentRoute()”。

Composer更新

一旦你完成了上面的更改，你就能运行“composer update”命令来更新你的核心应用程序文件了！如果你收到类加载错误，试着运行启用“--no-scripts”选项的更新命令，就像这样：“composer update --no-scripts”。

通配符事件监听器

通配符事件监听器不再附加事件到你的处理函数的参数里。如果你需要找到被触发的事件，你应该使用“Event::firing()”。

Contribution Guide

Introduction

Laravel is an open-source project and anyone may contribute to Laravel for its improvement. We welcome contributors, regardless of skill level, gender, race, religion, or nationality. Having a diverse, vibrant community is one of the core values of the framework!

To encourage active collaboration, Laravel currently only accepts pull requests, not bug reports. "Bug reports" may be sent in the form of a pull request containing a failing unit test. Alternatively, a demonstration of the bug within a sandbox Laravel application may be sent as a pull request to the main Laravel repository. A failing unit test or sandbox application provides the development team "proof" that the bug exists, and, after the development team addresses the bug, serves as a reliable indicator that the bug remains fixed.

The Laravel source code is managed on Github, and there are repositories for each of the Laravel projects:

- Laravel Framework
- Laravel Application
- Laravel Documentation
- Laravel Cashier
- Laravel Envoy
- Laravel Homestead
- Laravel Homestead Build Scripts
- Laravel Website
- Laravel Art

Core Development Discussion

Discussion regarding bugs, new features, and implementation of existing features takes place in the `#laravel-dev` IRC channel (Freenode). Taylor Otwell, the maintainer of Laravel, is typically present in the channel on weekdays from 8am-5pm (UTC-06:00 or America/Chicago), and sporadically present in the channel at other times.

The `#laravel-dev` IRC channel is open to all. All are welcome to join the channel either to participate or simply observe the discussions!

New Features

Before sending pull requests for new features, please contact Taylor Otwell via the `#laravel-dev` IRC channel (Freenode). If the feature is found to be a good fit for the framework, you are free to make a pull request. If the feature is rejected, don't give up! You are still free to turn your feature into a package which can be released to the world via Packagist.

When adding new features, don't forget to add unit tests! Unit tests help ensure the stability and reliability of the framework as new features are added.

Bugs

Via Unit Test

Pull requests for bugs may be sent without prior discussion with the Laravel development team. When submitting a bug fix, try to include a unit test that ensures the bug never appears again!

If you believe you have found a bug in the framework, but are unsure how to fix it, please send a pull request containing a failing unit test. A failing unit test provides the development team "proof" that the bug exists, and, after the development team addresses the bug, serves as a reliable indicator that the bug remains fixed.

If are unsure how to write a failing unit test for a bug, review the other unit tests included with the framework. If you're still lost, you may ask for help in the `#laravel` IRC channel (Freenode).

Via Laravel Liferaft

If you aren't able to write a unit test for your issue, Laravel Liferaft allows you to create a demo application that recreates the issue. Liferaft can even automate the forking and sending of pull requests to the Laravel repository. Once your Liferaft application is submitted, a Laravel

maintainer can run your application on Homestead and review your issue.

Creating Liferaft Applications

Laravel Liferaft provides a fresh, innovative way to contribute to Laravel. First, you will need to install the Liferaft CLI tool via Composer:

Installing Liferaft

```
composer global require "laravel/liferaft=~1.0"
```

Make sure to place the `~/.composer/vendor/bin` directory in your PATH so the `liferaft` executable is found when you run the `liferaft` command in your terminal.

Authenticating With GitHub

Before getting started with Liferaft, you need to register a GitHub personal access token. You can generate a personal access token from your GitHub settings panel. The default scopes selected by GitHub will be sufficient; however, if you wish, you may grant the `delete_repo` scope so Liferaft can delete your old sandbox applications.

```
liferaft auth my-github-token
```

Create A New Liferaft Application

To create a new Liferaft application, just use the `new` command:

```
liferaft new my-bug-fix
```

This command will do several things. First, it will fork the Laravel GitHub repository to your GitHub account. Next, it will clone the forked repository to your machine and install the Composer dependencies. Once the repository has been installed, you can begin recreating your issue within the Liferaft application!

Recreating Your Issue

After creating a Liferaft application, simply recreate your issue. You are free to define routes, create Eloquent models, and even create database migrations! The only requirement is that your application is able to run on a fresh Laravel Homestead virtual machine. This allows Laravel maintainers to easily run your application on their own machines.

Once you have recreated your issue within the Liferaft application, you're ready to send it back to the Laravel repository for review!

Send Your Application For Review

Once you have recreated your issue, it's almost time to send it for review! However, you should first complete the `liferaft.md` file that was generated in your Liferaft application. The first line of this file will be the title of your pull request. The remaining content will be included in the pull request body. Of course, GitHub Flavored Markdown is supported.

After completing the `liferaft.md` file, push all of your changes to your GitHub repository. Next, just run the Liferaft `throw` command from your application's directory:

```
liferaft throw
```

This command will create a pull request against the Laravel GitHub repository. A Laravel maintainer can easily grab your application and run it in their own Homestead environment!

Grabbing Liferaft Applications

Intrested in contributing to Laravel? Liferaft makes it painless to install Liferaft applications and view them on your own Homestead environment.

First, for convenience, clone the laravel/laravel into a `liferaft` directory on your machine:

```
git clone https://github.com/laravel/laravel.git liferaft
```

Next, check out the `develop` branch so you will be able to install Liferaft applications that target both stable and upcoming Laravel releases:

```
git checkout -b develop origin/develop
```

Next, you can run the Liferaft `grab` command from your repository directory. For example, if you want to install the Liferaft application associated with pull request #3000, you should run the following command:

```
liferaft grab 3000
```

The `grab` command will create a new branch on your Liferaft directory, and pull in the changes for the specified pull request. Once the Liferaft application is installed, simply serve the directory through your Homestead virtual machine! Once you debug the issue, don't forget to

send a pull request to the laravel/framework repository with the proper fix!

Have an extra hour and want to solve a random issue? Just run `grab` without a pull request ID:

```
liferaft grab
```

Which Branch?

> **Note:** This section primarly applies to those sending pull requests to the laravel/framework repository, not Liferaft applications.

All bug fixes should be sent to the latest stable branch. Bug fixes should **never** be sent to the `master` branch unless they fix features that exist only in the upcoming release.

Minor features that are **fully backwards compatible** with the current Laravel release may be sent to the latest stable branch.

Major new features should always be sent to the `master` branch, which contains the upcoming Laravel release.

If you are unsure if your feature qualifies as a major or minor, please ask Taylor Otwell in the `#laravel-dev` IRC channel (Freenode).

Security Vulnerabilities

If you discover a security vulnerability within Laravel, please send an e-mail to Taylor Otwell at taylorotwell@gmail.com. All security vulnerabilities will be promptly addressed.

Coding Style

Laravel follows the PSR-0 and PSR-1 coding standards. In addition to these standards, the following coding standards should be followed:

- The class namespace declaration must be on the same line as `<?php`.
- A class' opening `{` must be on the same line as the class name.
- Functions and control structures must use Allman style braces.
- Indent with tabs, align with spaces.

开始学习

安装

安装 **Composer**

Laravel 框架使用 Composer（PHP包管理工具，参考 Composer 中文文档）来管理代码依赖性。首先，你需要下载 Composer 的 PHAR 打包文件（`composer.phar`），下载完成后把它放在项目目录下或者放到 `usr/local/bin` 目录下以便在系统中全局调用。在Windows操作系统中，你可以使用 Composer 的Windows安装工具。

安装 **Laravel**

通过 **Laravel** 安装器安装

首先，通过 Composer 下载 Laravel 安装器。

```
composer global require "laravel/installer=~1.1"
```

请确保把 `~/.composer/vendor/bin` 路径添加到 PATH 环境变量里，这样`laravel` 可执行文件才能被命令行找到，以后您就可以在命令行下直接使用 `laravel` 命令.

安装成功后，可以使用命令 `laravel new` 在您指定的目录下创建一份全新安装的 Laravel。例如，`laravel new blog` 将会在当前目录下创建一个叫 `blog` 的目录，此目录里面存放着全新安装的 Laravel 以及其依赖的工具包。这种安装方法比通过 Composer 安装要快许多。

通过 **Composer** 的 **create-project** 命令安装 **Laravel**

还可以通过在命令行执行 Composer 的 `create-project` 命令来安装Laravel：

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

通过下载 **Laravel** 包安装

Composer 安装完成后，下载最新版Laravel框架，把它解压缩到你服务器上的一个目录中。然后在 Laravel 应用的根目录下运行命令行命令 `php composer.phar install`（或者 `composer install`）来安装所有的框架依赖包。在此过程中，为了成功完成安装，你需要在服务器上安装好 Git。

当 Laravel 框架安装好后，你可以使用命令行命令 `php composer.phar update` 来更新框架。

对服务器环境的要求

Laravel 框架对系统环境有如下要求：

- PHP >= 5.4
- MCrypt PHP 扩展

从 PHP 5.5 版本开始，针对某些操作系统的安装包需要你自己手工安装 PHP 的 JSON 扩展模块。如果你使用的是 Ubuntu，可以通过，`apt-get install php5-json` 命令直接安装。

配置

Laravel 框架几乎无需配置就可立即使用。你可以自由地快速开始开发。然而，你也许希望先查看下 `app/config/app.php` 配置文件和相关的文档说明。它包含了一些你也许要修改的配置选项，如 `时区` 和 `地区` 等。

一旦 Laravel 安装成功，你还应该配置本地开发环境，这样你就能在本地机器上开发时收集所有详细的错误信息了。默认情况下，详细错误信息报告在生产环境的配置文件中是关闭的。

> 注意：在生产环境的配置文件中，绝对不要把 `app.debug` 设置为 `true` ，切记，切记！

权限设置

Laravel框架有一个目录需要额外设置权限：需要为 `app/storage` 目录下的文件设置写权限。

路径设置

一些框架目录路径是可以设置的。如果需要改变这些目录的位置，可以查看 `bootstrap/paths.php` 文件中的设置。

Pretty URLs

Apache 服务器

Laravel框架通过设置 `public/.htaccess` 文件去除链接中的`index.php`。如果你你的服务器使用的是 Apache，请确保开启 `mod_rewrite` 模块。

如果框架附带的 `.htaccess` 文件在你的Apache环境中不起作用，请尝试下面这个版本：

```
Options +FollowSymLinks
RewriteEngine On
```

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

Nginx 服务器

如果是 Nginx 服务器，将下列指令放到网址的配置文件中，就能让网址更优雅了：

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

配置

引言

Laravel 框架的所有配置文件都存储于 `app/config` 目录。每个文件中的每个选项都做了详细的记录，以便随时翻阅文件，熟悉提供给你的选项。

有时你需要在程序执行阶段访问配置的值。你可以使用 `Config` 类：

访问一个配置的值

```
Config::get('app.timezone');
```

你也可以指定一个默认值，如果配置选项不存在它将被返回：

```
$timezone = Config::get('app.timezone', 'UTC');
```

设置一个配置的值

注意“点”语法风格可以用于访问不同文件里的值，你也可以在程序执行阶段设置配置的值：

```
Config::set('database.default', 'sqlite');
```

在程序执行阶段设置的配置的值仅作用于当前请求，并不会延续到后续请求中。

环境配置

基于应用程序的运行环境拥有不同配置值通常是非常有益的。例如，相对于生产服务器你希望在你本地开发设备上使用一个不同的缓存驱动。使用基于环境的配置可以很容易的做到这点。

在 `config` 目录里简单的创建一个与你的环境同名的文件夹，比如 `local`。接着，创建你希望在这个环境中指定选项被覆盖的配置文件。例如，在 local 环境下覆盖缓存驱动，你将要在 `app/config/local` 里创建一个 `cache.php` 文件并包含以下内容：

```
<?php

return array(
```

```
    'driver' => 'file',

);
```

> 注意：不要使用 'testing' 作为环境名称。这是为单元测试预留的。

注意，你不必指定基础配置文件中的 每一个 选项，仅需指定你希望覆盖的选项。环境配置文件将 "叠加" 在基础配置文件之上。

接着，我们需要告知框架如何判定自己运行于哪个环境中。默认的环境始终是 `production`。然而，你可以在安装程序的根目录下 `bootstrap/start.php` 文件中设置其他环境。在这个文件中你将找到一个 `$app->detectEnvironment` 的调用。向这个方法传入的数组用于确定当前环境。必要时你可以增加其他环境和机器名。

```
<?php

$env = $app->detectEnvironment(array(

    'local' => array('your-machine-name'),

));
```

在这个例子中，'local' 是环境名称而 'your-machine-name' 是你本地服务器的主机名。在 Linux 和 Mac 上，你可以使用 `hostname` 终端命令来确定你的主机名。

如果你需要更灵活的环境检测，你可以通过向 `detectEnvironment` 方法传入一个 `匿名函数`，这允许你按照自己的方式执行环境检测：

```
$env = $app->detectEnvironment(function()
{
    return $_SERVER['MY_LARAVEL_ENV'];
});
```

访问当前的应用环境

你可以通过 `environment` 方法访问当前的应用环境：

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

你也可以通过向 `environment` 方法传递参数来检测环境是否与给定的值匹配：

```
if (App::environment('local'))
{
    // 当前为 local 运行环境
}

if (App::environment('local', 'staging'))
{
```

```
    // 当前为 local 或 staging 运行环境
}
```

提供者配置

当使用环境配置，你可能想要 "追加" 环境 服务提供者 到你的基础 `app` 配置文件中。然而，如果你尝试这么做，你需要注意这个环境 `app` 提供者将会完全覆盖你的基础 `app` 配置文件中的值。要强制追加提供者，需要在你的环境 `app` 配置文件中使用 `append_config` 辅助函数：

```
'providers' => append_config(array(
    'LocalOnlyServiceProvider',
))
```

敏感信息保护配置

对于 "真实" 的应用程序，保持你所有的敏感配置信息位于配置文件之外，这是明智的。诸如数据库密码，第三方 API 密钥，加密密钥等尽可能的放置于配置文件之外。所以，要放在哪里呢？谢天谢地，Laravel 提供了一个非常简单的方案来保护这些配置项，使用 "点" 风格的文件。

首先，设置你的应用程序 识别你的机器是在 `local` 环境下。接着，在你项目的根目录创建一个 `.env.local.php` 文件，这通常与包含 `composer.json` 文件的目录相同。这个 `.env.local.php` 必须返回一个键值对数组，就像一个典型的 Laravel 配置文件：

```
<?php

return array(

    'TEST_STRIPE_KEY' => 'super-secret-sauce',

);
```

这个文件中所有返回的键值对，将会自动通过 `$_ENV` 和 `$_SERVER` PHP "超全局变量" 变为可用。现在你可以在你的配置文件中引用这些全局变量：

```
'key' => $_ENV['TEST_STRIPE_KEY']
```

确保在你的 `.gitignore` 文件中增加了对 `.env.local.php` 文件的忽略规则。这将允许你团队的其他开发者创建他们自己的本地环境配置，以及从源头隐藏你的敏感配置项。

现在，在你的生产服务器上，你项目的根目录里创建一个 `.env.php` 文件，包含你生产环境所对应的值。就像 `.env.local.php` 文件，生产环境 `.env.php` 文件不应该被包含在源码中。

> 注意：你可以为每一个应用程序支持的环境创建一个文件。例如，在 `development` 环境下将载入 `.env.development.php` 文件，如果它存在的话。然而，`production` 将永远使用 `.env.php` 文件。

维护模式

当你的应用程序处于维护模式中，所有进入到你应用程序的路由都将显示一个自定义的视图。这使得当你的应用程序更新或进行维护时，可以很容易的 "禁用" 你的应用程序。在你的 `app/start/global.php` 文件中已经准备了一个 `App::down` 方法的调用。当你的应用程序处于维护模式中时，该方法的响应将发送给用户。

要启用维护模式，可以简单的执行 `down` Artisan 命令：

```
php artisan down
```

要禁用维护模式，则使用 `up` 命令：

```
php artisan up
```

当你的应用程序处于维护模式时，若需显示一个自定义视图，你可以在应用程序的 `app/start/global.php` 文件中添加如下代码：

```
App::down(function()
{
    return Response::view('maintenance', array(), 503);
});
```

如果传递给 `down` 方法的闭包返回 `NULL`，那么在此次请求中将忽略维护模式。

维护模式 & 队列

在你的应用程序处于维护模式期间，不会有 队列工作 被处理。一旦应用程序退出维护模式，这些工作将继续正常处理。

Laravel Homestead

前言

Laravel努力为整个PHP开发过程提供令人愉快的开发体验，也包括开发者的本地开发环境。Vagrant（http://vagrantup.com）提供了一种既简单又优雅的方式来管理和装备虚拟机。

Laravel Homestead是一个官方的、预封装的Vagrant“箱子”，它提供给你一个奇妙的开发环境而不需要你在本机上安装PHP、HHVM、web服务器和其它的服务器软件。不用再担心搞乱你的操作系统！Vagrant箱子是完全可支配的。如果出现故障，你可以在几分种内完成销毁和重建箱子！

Homestead能运行在所有的Windows、Mac或Linux系统上，它包含了Nginx、PHP 5.6、MySQL、Postgres、Redis、Memcached和你开发神奇的Laravel应用程序需要的所有其它软件。

> 注意：如果你使用的是 Windows，你需要开启硬件虚拟支持(VT-x)。一般是要在 BIOS 中进行设置。

Homestead 目前基于 Vagrant 1.6 版本进行构建和测试。

包含的软件

- Ubuntu 14.04
- PHP 5.6
- HHVM
- Nginx
- MySQL
- Postgres
- Node (With Bower, Grunt, and Gulp)
- Redis
- Memcached
- Beanstalkd
- Laravel Envoy

- Fabric + HipChat Extension

安装和设置

安装**VirtualBox**和**Vagrant**

在启动Homestead环境之前，你必须安装VirtualBox（https://www.virtualbox.org/wiki/Downloads）和Vagrant（http://www.vagrantup.com/downloads.html）。这两个软件为所有主流的操作系统提供了简单易用的可视化安装界面。

添加**Vagrant**箱子

一旦VirtualBox和Vagrant安装完成，你应该添加“laravel/homestead”箱子到你的Vagrant安装目录下，在终端使用下面的命令，这将花费几分钟的时间来下载箱子，这取决于你的网速：

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

安装 **Homestead**

一旦箱子被添加到Vagrant安装目录下，你就可以通过 Composer 的 `global` 指令来安装 Homestead 命令行工具了：

```
composer global require "laravel/homestead=~2.0"
```

确保将 `~/.composer/vendor/bin` 目录添加到 PATH 环境变量中，这样就能在执行 `homestead` 指令时找到对应的可执行程序了。

一旦安装了 Homestead 命令行工具，请执行 `init` 来创建 `Homestead.yaml` 配置文件：

```
homestead init
```

生成的 `Homestead.yaml` 文件将被放置于 `~/.homestead` 目录下。如果你使用的是 Mac 或 Linux 操作系统，还可以通过执行 `homestead edit` 指令来编辑 `Homestead.yaml` 文件：

```
homestead edit
```

设置你的**SSH**密钥

接下来，你需要编辑 `Homestead.yaml` 文件。在这个文件里，你可以配置公共SSH密钥的路径，也可以配置主机与Homestead虚拟机的共享目录。

还没有SSH密钥？在Mac和Linux机器上，通常你可以使用下面的命令创建一个SSH密钥对：

```
ssh-keygen -t rsa -C "you@homestead"
```

在Windows机器上，你可以安装Git（http://git-scm.com/）工具，并使用Git自带的“Git Bash”命令行工具执行上面的命令。或者，你可以使用PuTTY（http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html）工具或PuTTYgen（http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html）工具。

一旦你创建了一个SSH密钥，就可以在“Homestead.yaml”文件里为“authorize”属性指定密钥的路径。

配置共享目录

“Homestead.yaml”文件里的“folders”属性列出所有你想与Homestead环境共享的目录。当这些目录中的文件发生了改变，它们将在本机和Homestead环境之间保持同步。你可以根据需要配置尽可能多的共享目录！

配置**Nginx**站点

不熟悉Nginx？没关系。Homestead环境里的“sites”属性允许你轻松地将一个“域”映射到一个目录。“Homestead.yaml”文件里包含一个示例站点配置。再强调一遍，你可以根据需要添加尽可能多的站点到Homestead环境里。Homestead能够为你的每一个Laravel项目提供一个方便的虚拟环境！

你可以创建任何基于 Homestead 的站点并使用 HHVM 。通过设置 `hhvm` 选项为 `true` 即可：

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

Bash 链接

想在你的Homestead箱子里加入Bash 链接，只需要在 `~/.homestead` 目录里简单的添加 `aliases` 文件即可。

启动**Vagrant**箱子

一旦你按照意愿编辑了“Homestead.yaml”文件，就可以在终端上的“Homestead”目录下执行 `homestead up` 命令。Vagrant将启动虚拟机，并自动配置共享目录和Nginx站点！如果需要销毁虚拟机，可以使用 `homestead destroy` 指令。`homestead list` 用于列出所有可用的 Homestead 指令。

不要忘记把你的Nginx站点的“域”添加到机器里的“hosts”文件里！“hosts”文件将把对本地域的请求

重定向到Homestead环境里。在Mac和Linux机器上，这个文件位于“/etc”目录。在Windows机器上，它位于“C:\Windows\System32\drivers\etc”目录。你添加到此文件的内容就像下面这样：

```
192.168.10.10  homestead.app
```

确保列出的 IP 和你在 `Homestead.yaml` 文件中设置的一致。一旦你把域名添加到 `hosts` 文件中，你就可以通过浏览器访问此站点了！

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

想知道如何连接数据库，请接着看！

日常使用

通过SSH连接

为了通过SSH连接到 Homestead 环境，只需在命令行窗口中输入 `homestead ssh` 指令即可。

连接到数据库

“homestead”数据库是为箱子外面的MySQL和Postres配置的。为了更加方便，Laravel的本地数据库配置默认设置为使用这个数据库。

想通过你主机上的Navicat或Sequel Pro连接MySQL或Postgres，你应该使用端口33060（MySQL）或54320（Postgres）来连接“127.0.0.1”。这两个数据库的用户名和密码都是“homestead” / “secret”。

> 注意：当从主机连接数据库时，你应该只使用非标准的端口。在你的Laravel配置文件中，你将使用默认的3306和5432端口，因为Laravel运行在虚拟机当中。

添加额外站点

一旦你的Homestead环境被分配并运行，你可能想为Laravel应用程序添加额外的Nginx站点。在一个Homestead环境中，你可以按意愿运行尽可能多的Laravel应用程序。有两种方法可以做到这一点。首先，你可以简单的添加站点到“Homestead.yaml”文件里，先对箱子执行“vagrant destroy”命令，然后再执行“vagrant provision”命令。

或者，你可以使用Homestead环境里的“serve”脚本。想使用“serve”脚本，先SSH到Homestead环境并运行下面的命令：

```
serve domain.app /home/vagrant/Code/path/to/public/directory
```

> 注意：在执行“serve”命令后，不要忘记添加新站点到你机器的“hosts”文件里！

端口

下面的端口被转发到你的Homestead环境：

- **SSH:** 2222 -> 转发到 22
- **HTTP:** 8000 -> 转发到 80
- **MySQL:** 33060 -> 转发到 3306
- **Postgres:** 54320 -> 转发到 5432

请求的生命周期

概述

在现实世界中使用工具时，如果理解了工具的工作原理，使用起来就会更加有底气。应用开发也是如此。当你理解了开发工具是如何工作的，使用起来就会更加自如。这篇文档的目标就是提供一个高层次的概述，使你对于Laravel框架的运行方式有一个较好的把握。在更好地了解了整个框架之后，框架的组件和功能就不再显得那么神秘，开发起应用来也更加得心应手。这篇文档包含了关于请求生命周期的高层次概述，以及启动文件和应用程序事件的相关内容。

如果你不能立即理解所有的术语，别灰心，可以先有一个大致的把握，在阅读文档其他章节的过程中继续积累和消化知识。

请求的生命周期

发送给应用程序的所有请求都经由 `public/index.php` 脚本处理。如果使用的是 Apache 服务器，Laravel中包含的 `.htaccess` 文件将对所有请求进行处理并传递给 `index.php`。这是Laravel从接受客户端请求到返回响应给客户端的整个过程的开始。若能对于Laravel的引导过程(bootstrap process)有一个大致的认识，将有助于理解框架，我们不妨先讨论这个。

到目前为止，学习Laravel引导过程所需掌握的最重要的概念就是 服务提供器。打开 `app/config/app.php` 配置文件，找到 `providers` 数组，你会发现一个服务提供器的列表。这些提供器充当了Laravel的主要引导机制。在我们深入服务提供器之前，先回到 `index.php`的讨论。当一个请求进入 `index.php` 文件，`bootstrap/start.php` 文件会被加载。这个文件会创建一个 Laravel `Application` 对象，该对象同时作为框架的 IoC 容器。

`Application` 对象创建完成后，框架会设置一些路径信息并运行 环境检测 。然后会执行位于Laravel源码内部的引导脚本，并根据你的配置文件设置时区、错误报告等其他信息。除了配置这些琐碎的配置选项以外，该脚本还会做一件非常重要的事情：注册所有为应用程序配置的服务提供器。

简单的服务提供器只包含一个方法：`register`。当应用程序对象通过自身的 `register` 方法注册某个服务提供器时，会调用该服务提供器的 `register` 方法。服务提供器通过这个方法向 IoC 容器 注

册一些东西。从本质上讲，每个服务提供器都是将一个或多个 闭包 绑定到容器中，你可以通过这些闭包访问绑定到应用程序的服务。例如，`QueueServiceProvider` 注册了多个闭包以便使用与 队列 相关的多个类。当然，服务提供器并不局限于向IoC容器注册内容，而是可以用于任何引导性质的任务。服务提供器可以注册事件监听器、视图合成器、Artisan命令等等。

在注册完所有服务提供器后，`app/start` 下的文件会被加载。最后，`app/routes.php` 文件会被加载。一旦 `routes.php` 文件被加载，Request 对象就被发送给应用程序对象，继而被派发到某个路由上。

我们总结一下：

1. 请求进入 `public/index.php` 文件。
2. `bootstrap/start.php` 文件创建应用程序对象并检测环境。
3. 内部的 `framework/start.php` 文件配置相关设置并加载服务提供器。
4. 加载应用程序 `app/start` 目录下的文件。
5. 加载应用程序的 `app/routes.php` 文件。
6. 将 Request 对象发送给应用程序对象，应用程序对象返回一个 Response 对象。
7. 将 Response 对象发回客户端。

你应该已经掌握了 Laravel 应用程序是如何处理发来的请求的。下面我们来看一下启动文件。

启动文件

应用程序的启动文件被存放在`app/start`目录中。默认情况下，该目录下包含三个文件：`global.php`、`local.php` 和 `artisan.php`文件。需要获取更多关于`artisan.php`的信息，可以参考文档Artisan 命令行。

`global.php`启动文件默认包含一些基本项目，例如日志的注册以及载入`app/filters.php` 文件。然而，你可以在该文件里做任何你想做的事情。无论在什么环境下，它都将会被自动包含进_每一个_request中。而`local.php` 文件仅在`local`环境下被执行。获取更多关于环境的信息，请查看文档配置。

当然，如果除了`local`环境你还有其他环境的话，你也可以为针对这些环境创建启动文件。这些文件将在应用程序运行在该环境中时被自动包含。假设你在 `bootstrap/start.php` 文件中配置了一个 `development` 环境，你可以创建一个 `app/start/development.php` 文件，在那个环境下任何进入应用程序的请求都会包含该文件。

启动文件里存放什么

启动文件主要用来存放任何“引导”性质的代码。例如，你可以在启动文件中注册视图合成器，配置

日志信息，或是进行一些PHP设置等。具体做什么取决于你。当然了，把所有引导代码都丢到启动文件里会使启动文件变得杂乱。对于大型应用而言，或是启动文件显得太杂乱了，请考虑将某些引导代码移至 服务提供器 中。

应用程序事件

注册应用程序事件

你还可以通过注册 `before`、`after`、`finish` 和 `shutdown`应用程序事件以便在处理request之前或后做一些操作：

```
App::before(function($request)
{
    //
});

App::after(function($request, $response)
{
    //
});
```

这些事件的监听器会在每个到达应用程序的请求处理之前（`before`）或之后（`after`）运行。可以利用这些事件来设置全局过滤器(filter)，或是对于发回客户端的响应(response)统一进行修改。你可以在某个启动文件中或者 服务提供器 中注册这些事件。

你也可以在 `matched` 事件上注册一个监听器，当一个传入请求已经和一个路由相匹配，但还未执行此路由之前，此事件就会被触发：

```
Route::matched(function($route, $request)
{
    //
});
```

当来自应用程序的响应发送至客户端后会触发 `finish` 事件。这个事件适合处理应用程序所需的最后的收尾工作。当所有 `finish` 事件的监听器都执行完毕后会立即触发 `shutdown` 事件，如果想在脚本结束前再做一些事情，这是最后的机会。不过在大多数情况下，你都不需要用到这些事件。

路由

基本路由

应用中的大多数路都会定义在 `app/routes.php` 文件中。最简单的Laravel路由由URI和闭包回调函数组成。

基本 **GET** 路由

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

基本 **POST** 路由

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

为多个动作注册同一个路由

```
Route::match(array('GET', 'POST'), '/', function()
{
    return 'Hello World';
});
```

注册一个可以响应任何**HTTP**动作的路由

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

```
});
```

仅支持**HTTPS**的路由

```
Route::get('foo', array('https', function()
{
    return 'Must be over HTTPS';
}));
```

实际开发中经常需要根据路由生成 URL，`URL::to`方法就可以满足此需求：

```
$url = URL::to('foo');
```

路由参数

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

可选路由参数

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

带有默认值的可选路由参数

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

用正则表达式限定的路由参数

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

Route::get('user/{id}', function($id)
{
    //
})
->where('id', '[0-9]+');
```

传递参数限定的数组

当然，必要的时候你还可以传递一个包含参数限定的数组作为参数：

```
Route::get('user/{id}/{name}', function($id, $name)
{
    //
})
->where(array('id' => '[0-9]+', 'name' => '[a-z]+'))
```

定义全局模式

如果希望在全局范围用指定正则表达式限定路由参数，可以使用 `pattern` 方法：

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

Route::get('user/{id}', function($id)
{
    // Only called if {id} is numeric.
});
```

访问路由参数

如果想在路由范围外访问路由参数，可以使用 `Route::input` 方法：

```
Route::filter('foo', function()
{
    if (Route::input('id') == 1)
    {
        //
    }
});
```

路由过滤器

路由过滤器提供了非常方便的方法来限制对应用程序中某些功能访问，例如对于需要验证才能访问的功能就非常有用。Laravel框架自身已经提供了一些过滤器，包括 `auth`过滤器、`auth.basic`过滤器、`guest`过滤器以及`csrf`过滤器。这些过滤器都定义在`app/filter.php`文件中。

定义一个路由过滤器

```
Route::filter('old', function()
{
    if (Input::get('age') < 200)
    {
        return Redirect::to('home');
    }
});
```

如果过滤器返回了response，那么该response将被认为对应的是此次request，路由将不会被执行，并且，此路由中所有定义在此过滤器之后的代码也都不会被执行。

为路由绑定过滤器

```
Route::get('user', array('before' => 'old', function()
{
    return 'You are over 200 years old!';
}));
```

将过滤器绑定为控制器**Action**

```
Route::get('user', array('before' => 'old', 'uses' => 'UserController@showProfile'));
```

为路由绑定多个过滤器

```
Route::get('user', array('before' => 'auth|old', function()
{
    return 'You are authenticated and over 200 years old!';
}));
```

通过数据绑定多个过滤器

```
Route::get('user', array('before' => array('auth', 'old'), function()
{
    return 'You are authenticated and over 200 years old!';
}));
```

指定过滤器参数

```
Route::filter('age', function($route, $request, $value)
{
    //
});

Route::get('user', array('before' => 'age:200', function()
{
    return 'Hello World';
}));
```

After filters receive a `$response` as the third argument passed to the filter:

```
Route::filter('log', function($route, $request, $response)
{
    //
});
```

基于模式的过滤器

你也可以只针对URI为一组路由指定过滤器。

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

```
    //
});

Route::when('admin/*', 'admin');
```

上述案例中，`admin`过滤器将会应用到所有以`admin/`开头的路由中。星号是通配符，将会匹配任意多个字符的组合。

还可以针对HTTP动作限定模式过滤器：

```
Route::when('admin/*', 'admin', array('post'));
```

过滤器类

过滤器的高级用法中，还可以使用类来替代闭包函数。由于过滤器类是通过IoC container实现解析的，所有，你可以在这些过滤器中利用依赖注入（dependency injection）的方法实现更好的测试能力。

注册过滤器类

```
Route::filter('foo', 'FooFilter');
```

默认情况下，FooFilter类里的filter方法将被调用：

```
class FooFilter {

    public function filter()
    {
        // Filter logic...
    }

}
```

如果你不想使用filter方法，那就指定另一个方法：

```
Route::filter('foo', 'FooFilter@foo');
```

命名路由

重定向和生成URL时，使用命名路由会更方便。你可以为路由指定一个名字，如下所示：

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

还可以为 controller action指定路由名称：

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

现在，你可以使用路由名称来创建URL和重定向：

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

$redirect = Redirect::route('profile');
```

可以使用`currentRouteName`方法来获取当前运行的路由名称:

```
$name = Route::currentRouteName();
```

路由组

有时你可能需要为一组路由应用过滤器。使用路由组就可以避免单独为每个路由指定过滤器了：

```
Route::group(array('before' => 'auth'), function()
{
    Route::get('/', function()
    {
        // Has Auth Filter
    });

    Route::get('user/profile', function()
    {
        // Has Auth Filter
    });
});
```

你也可以在组数组中使用`namespace`参数来指定此组里的控制器都在一个给定的命名空间里：

```
Route::group(array('namespace' => 'Admin'), function()
{
    //
});
```

子域名路由

Laravel中的路由功能还支持通配符子域名，并且能将域名中匹配通配符的参数传递给你:

注册子域名路由

```
Route::group(array('domain' => '{account}.myapp.com'), function()
{

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

```
});
```

路由前缀

可以通过`prefix`属性为组路由设置前缀：

```
Route::group(array('prefix' => 'admin'), function()
{

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

});
```

路由与模型绑定

模型绑定，为在路由中注入模型实例提供了便捷的途径。例如，你可以向路由中注入匹配用户ID的整个模型实例，而不是仅仅注入用户ID。首先，使用 `Route::model` 方法指定要被注入的模型：

给模型绑定参数

```
Route::model('user', 'User');
```

然后，定义一个包含`{user}`参数的路由：

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

由于我们已将`{user}`参数绑定到了`User`模型，因此可以向路由中注入一个`User`实例。例如，对`profile/1`的访问将会把ID为1的`User`实例注入到路由中。

> 注意：如果在数据库中无法匹配到对应的模型实例，404错误将被抛出。

如果你希望自定义"not found"行为，可以通过传递一个闭包函数作为 `model` 方法的第三个参数：

```
Route::model('user', 'User', function()
{
    throw new NotFoundHttpException;
});
```

如果你想自己实现路由参数的解析，只需使用`Route::bind`方法即可：

```
Route::bind('user', function($value, $route)
{
    return User::where('name', $value)->first();
```

```
});
```

抛出 **404** 错误

有两种从路由中手动触发404错误的方法。首先，你可以使用`App::abort`方法：

```
App::abort(404);
```

其次，你可以抛出`Symfony\Component\HttpKernel\Exception\NotFoundHttpException`异常。

更多关于处理404异常以及错误发生时自定义response的信息可以查看错误文档。

控制器路由

Laravel不光提供了利用闭包函数处理路由的功能，还可以路由到控制器，甚至支持创建 resource controllers。

参见文档 Controllers 以获取更多信息。

请求与输入

基本输入数据

您可以经由几个简洁的方法拿到用户的输入数据。不需要担心发出请求时使用的 HTTP 响应方式，取得输入数据的方式都是相同的。

取得特定输入数据

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

取得特定输入数据，若没有便则取默认值

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

确认是否有输入数据

```
if (Input::has('name'))
{
    //
}
```

取得所有发出请求时传入的输入数据

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

取得部分发出请求时传入的输入数据

```
$input = Input::only('username', 'password');

$input = Input::except('credit_card');
```

如果是「数组」形式的输入数据，可以使用「点」语法取得数组：

```
$input = Input::get('products.0.name');
```

> 提醒：有些 JavaScript 函数库如 Backbone 可能会送出 JSON 格式的输入数据，但是一样可以使用 `Input::get` 取得数据。

Cookies

Laravel 建立的 cookie 会加密并且加上认证记号，意味着如果cookie被客户端擅自改动，会导致 cookie 失效。

取得 **Cookie** 值

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

加上新的 **Cookie** 到回应

```
$response = Response::make('Hello World');

$response->withCookie(Cookie::make('name', 'value', $minutes));
```

加入 **Cookie** 队列到下一个回应

如果您想在回应被建立前设定 cookie，使用 `Cookie::queue()` 方法。Cookie 会在最后自动加到回应里。

```
Cookie::queue($name, $value, $minutes);
```

建立永久有效的 **Cookie**

```
$cookie = Cookie::forever('name', 'value');
```

旧输入数据

您可能想要在用户下一次发送请求前，保留这次的输入数据。例如，您可能需要在表单验证失败后重新填入之前输入的表单值。

将输入数据存成一次性 **Session**

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

将部分输入数据存成一次性 **Session**

```
Input::flashOnly('username', 'email');

Input::flashExcept('password');
```

您很可能常常需要在重新跳转至前一页，并将输入数据存成一次性 Session 。只要在重定向跳转方法串接的方法中传入输入数据，就能简单地完成。

```
return Redirect::to('form')->withInput();

return Redirect::to('form')->withInput(Input::except('password'));
```

> 提示：您可以使用 Session 类将不同请求数据存成其他一次性 Session。

取得旧输入数据

```
Input::old('username');
```

上传文件

取得上传文件

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

确认文件是否有上传

```
if (Input::hasFile('photo'))
{
    //
}
```

`file` 方法回传的对象是 `Symfony\Component\HttpFoundation\File\UploadedFile` 的实例，`UploadedFile` 继承了 PHP 的 `SplFileInfo` 类并且提供了很多方法和文件互动。

确认上传的文件是否有效

```
if (Input::file('photo')->isValid())
{
    //
}
```

移动上传文件

```
Input::file('photo')->move($destinationPath);

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

取得上传文件所在的路径

```
$path = Input::file('photo')->getRealPath();
```

取得上传文件的原始名称

```
$name = Input::file('photo')->getClientOriginalName();
```

取得上传文件的后缀名

```
$extension = Input::file('photo')->getClientOriginalExtension();
```

取得上传文件的大小

```
$size = Input::file('photo')->getSize();
```

取得上传文件的 **MIME** 类型

```
$mime = Input::file('photo')->getMimeType();
```

请求信息

`Request` 类提供很多方法检查 HTTP 请求，它继承了 `Symfony\Component\HttpFoundation\Request` 类，下面是一些使用方式。

取得请求 **URI**

```
$uri = Request::path();
```

取得请求方法

```
$method = Request::method();

if (Request::isMethod('post'))
{
    //
}
```

确认请求路径是否符合特定格式

```
if (Request::is('admin/*'))
{
    //
}
```

取得请求 **URL**

```
$url = Request::url();
```

取得请求 **URI** 部分片段

```
$segment = Request::segment(1);
```

取得请求 **header**

```
$value = Request::header('Content-Type');
```

从 **$_SERVER** 取得值

```
$value = Request::server('PATH_INFO');
```

确认是否为 **HTTPS** 请求

```
if (Request::secure())
{
    //
}
```

确认是否为 **AJAX** 请求

```
if (Request::ajax())
{
    //
}
```

确认请求是否有 **JSON Content Type**

```
if (Request::isJson())
{
    //
}
```

确认是否要求 **JSON** 回应

```
if (Request::wantsJson())
{
    //
}
```

确认要求的回应格式

`Request::format` 方法会基于 HTTP Accept 标头回传请求的回应格式：

```
if (Request::format() == 'json')
{
    //
}
```

视图（ **View** ）与响应（ **Response** ）

基本响应

从路由回传字串

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

建立自定义响应

`Response` 实例继承了 `Symfony\Component\HttpFoundation\Response` 类，其提供了很多方法建立 HTTP 响应。

```
$response = Response::make($contents, $statusCode);

$response->header('Content-Type', $value);

return $response;
```

如果想要使用 `Response` 类的方法，但最终回传视图给用户，您可以使用简便的 `Response::view` 方法：

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

附加 **Cookies** 到响应

```
$cookie = Cookie::make('name', 'value');

return Response::make($content)->withCookie($cookie);
```

重定向跳转

回传重定向跳转

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

回传重定向跳转并且加上快闪数据（ **Flash Data** ）

```
return Redirect::to('user/login')->with('message', 'Login Failed');
```

> 提示：`with` 方法会设定快闪数据到 session，所以可以使用 `Session::get` 取得数据。

回传根据路由名称的重定向跳转

```
return Redirect::route('login');
```

回传根据路由名称的重定向跳转，并给予路由参数赋值

```
return Redirect::route('profile', array(1));
```

回传根据路由名称的重定向跳转，并给予特定名称路由参数赋值

```
return Redirect::route('profile', array('user' => 1));
```

回传根据控制器动作的重定向跳转

```
return Redirect::action('HomeController@index');
```

回传根据控制器动作的重定向跳转，并给予参数赋值

```
return Redirect::action('UserController@profile', array(1));
```

回传根据控制器动作的重定向跳转，并给予特定名称参数赋值

```
return Redirect::action('UserController@profile', array('user' => 1));
```

Views

视图通常包含 HTML，并且提供便利的方式分开控制器和表现层。视图储存在 `app/views` 目录下。

一个简单的视图可能看起来如下：

```
<!-- View stored in app/views/greeting.php -->

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

视图可以像这样回传到用户浏览器：

```
Route::get('/', function()
{
    return View::make('greeting', array('name' => 'Taylor'));
});
```

`View::make` 方法传入的第二个参数是可以在视图里使用的数组数据。

传递数据到视图

```
// 使用通用方式
$view = View::make('greeting')->with('name', 'Steve');

// 使用魔术方法
$view = View::make('greeting')->withName('steve');
```

上面的例子里，将可以在视图里使用变量 `$name`，其值为 `Steve`。

您可以传入数组作为 `make` 方法第二个参数：

```
$view = View::make('greetings', $data);
```

您也可以设定所有视图共用数据：

```
View::share('name', 'Steve');
```

传递子视图到视图

有时候您可能想要传递子视图到另一个视图。例如，有一个子视图存在 `app/views/child/view.php`，可以像这样将它传递到另一个视图：

```
$view = View::make('greeting')->nest('child', 'child.view');

$view = View::make('greeting')->nest('child', 'child.view', $data);
```

如此可以在视图里渲染子视图：

```
<html>
    <body>
        <h1>Hello!</h1>
        <?php echo $child; ?>
    </body>
</html>
```

确认视图是否存在

如果您需要确认视图是否存在，使用 `View::exists` 方法：

```
if (View::exists('emails.customer'))
{
    //
}
```

视图组件

视图组件是当渲染视图时调用的回调函数或类方法。如果您想在每次渲染某些视图时绑定数据，视图组件可以把这样的逻辑组织在同一个地方。因此，视图组件的作用可能如 "view models" 或是 "presenters"。

定义一个组件

```
View::composer('profile', function($view)
{
    $view->with('count', User::count());
});
```

之后每当 `profile` 视图被渲染时，`count` 变量就会被绑定到视图。

您也可以把一个组件同时附加到很多视图。

```
View::composer(array('profile','dashboard'), function($view)
{
    $view->with('count', User::count());
});
```

如使用类作为组件，提供了可以从 IoC 容器自动解析组件的好处，您可以像这样做：

```
View::composer('profile', 'ProfileComposer');
```

一个视图组件类应该像这样定义：

```
class ProfileComposer {

    public function compose($view)
    {
        $view->with('count', User::count());
    }

}
```

定义很多组件

您可以使用 `composers` 方法群组视图组件：

```
View::composers(array(
    'AdminComposer' => array('admin.index', 'admin.profile'),
    'UserComposer' => 'user',
```

```
    'ProductComposer@create' => 'product'
));
```

> 提示：组件类没有一定要放在什么地方，您可以将它们放在任何地方，只要可以使用 `composer.json` 自动载入即可。

视图创建者

视图 创建者 几乎和组件运作方式一样；只是他们会在视图初始化时就立刻建立起来。要注册一个创建者，只要使用 `creator` 方法：

```
View::creator('profile', function($view)
{
    $view->with('count', User::count());
});
```

特殊响应

建立 **JSON** 响应

```
return Response::json(array('name' => 'Steve', 'state' => 'CA'));
```

建立 **JSONP** 响应

```
return Response::json(array('name' => 'Steve', 'state' => 'CA'))->setCallback(Input::get('ca
```

建立下载文件响应

```
return Response::download($pathToFile);

return Response::download($pathToFile, $name, $headers);
```

> 提醒：管理文件下载的扩展包，Symfony HttpFoundation，要求下载文件名必须为 ASCII 。

响应宏

如果您想要自定义可以在很多路由和控制器重复使用的响应，可以使用 `Response::macro` 方法：

```
Response::macro('caps', function($value)
{
    return Response::make(strtoupper($value));
});
```

`macro` 方法第一个参数为宏名称，第二个参数为闭合函数。闭合函数会在 `Response` 调用宏名称的时候被执行：

```
return Response::caps('foo');
```

您可以把定义自己的宏放在 `app/start` 里面的文件。又或者，您可以将宏组织成独立的文件，并且从其中一个 `start` 文件里引入。

Controllers

Basic Controllers

Instead of defining all of your route-level logic in a single `routes.php` file, you may wish to organize this behavior using Controller classes. Controllers can group related route logic into a class, as well as take advantage of more advanced framework features such as automatic dependency injection.

Controllers are typically stored in the `app/controllers` directory, and this directory is registered in the `classmap` option of your `composer.json` file by default. However, controllers can technically live in any directory or any sub-directory. Route declarations are not dependent on the location of the controller class file on disk. So, as long as Composer knows how to autoload the controller class, it may be placed anywhere you wish.

Here is an example of a basic controller class:

```
class UserController extends BaseController {

    /**
     * Show the profile for the given user.
     */
    public function showProfile($id)
    {
        $user = User::find($id);

        return View::make('user.profile', array('user' => $user));
    }

}
```

All controllers should extend the `BaseController` class. The `BaseController` is also stored in the `app/controllers` directory, and may be used as a place to put shared controller logic. The `BaseController` extends the framework's `Controller` class. Now, we can route to this controller action like so:

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

If you choose to nest or organize your controller using PHP namespaces, simply use the fully qualified class name when defining the route:

```
Route::get('foo', 'Namespace\FooController@method');
```

> **Note:** Since we're using Composer to auto-load our PHP classes, controllers may live anywhere on the file system, as long as composer knows how to load them. The controller directory does not enforce any folder structure for your application. Routing to controllers is entirely de-coupled from the file system.

You may also specify names on controller routes:

```
Route::get('foo', array('uses' => 'FooController@method',
                                        'as' => 'name'));
```

To generate a URL to a controller action, you may use the `URL::action` method or the `action` helper method:

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

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

You may access the name of the controller action being run using the `currentRouteAction` method:

```
$action = Route::currentRouteAction();
```

Controller Filters

Filters may be specified on controller routes similar to "regular" routes:

```
Route::get('profile', array('before' => 'auth',
            'uses' => 'UserController@showProfile'));
```

However, you may also specify filters from within your controller:

```
class UserController extends BaseController {

    /**
     * Instantiate a new UserController instance.
     */
    public function __construct()
    {
        $this->beforeFilter('auth', array('except' => 'getLogin'));

        $this->beforeFilter('csrf', array('on' => 'post'));
```

```
        $this->afterFilter('log', array('only' =>
                           array('fooAction', 'barAction')));
    }

}
```

You may also specify controller filters inline using a Closure:

```
class UserController extends BaseController {

    /**
     * Instantiate a new UserController instance.
     */
    public function __construct()
    {
        $this->beforeFilter(function()
        {
            //
        });
    }

}
```

If you would like to use another method on the controller as a filter, you may use `@` syntax to define the filter:

```
class UserController extends BaseController {

    /**
     * Instantiate a new UserController instance.
     */
    public function __construct()
    {
        $this->beforeFilter('@filterRequests');
    }

    /**
     * Filter the incoming requests.
     */
    public function filterRequests($route, $request)
    {
        //
    }

}
```

Implicit Controllers

Laravel allows you to easily define a single route to handle every action in a controller. First, define the route using the `Route::controller` method:

```
Route::controller('users', 'UserController');
```

The `controller` method accepts two arguments. The first is the base URI the controller handles, while the second is the class name of the controller. Next, just add methods to your controller, prefixed with the HTTP verb they respond to:

```
class UserController extends BaseController {

    public function getIndex()
    {
        //
    }

    public function postProfile()
    {
        //
    }

    public function anyLogin()
    {
        //
    }

}
```

The `index` methods will respond to the root URI handled by the controller, which, in this case, is `users`.

If your controller action contains multiple words, you may access the action using "dash" syntax in the URI. For example, the following controller action on our `UserController` would respond to the `users/admin-profile` URI:

```
public function getAdminProfile() {}
```

RESTful Resource Controllers

Resource controllers make it easier to build RESTful controllers around resources. For example, you may wish to create a controller that manages "photos" stored by your application. Using the `controller:make` command via the Artisan CLI and the `Route::resource` method, we can quickly create such a controller.

To create the controller via the command line, execute the following command:

```
php artisan controller:make PhotoController
```

Now we can register a resourceful route to the controller:

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

This single route declaration creates multiple routes to handle a variety of RESTful actions on the photo resource. Likewise, the generated controller will already have stubbed methods for each of these actions with notes informing you which URIs and verbs they handle.

Actions Handled By Resource Controller

Verb	Path	Action	Route Name
GET	/resource	index	resource.index
GET	/resource/create	create	resource.create
POST	/resource	store	resource.store
GET	/resource/{resource}	show	resource.show
GET	/resource/{resource}/edit	edit	resource.edit
PUT/PATCH	/resource/{resource}	update	resource.update
DELETE	/resource/{resource}	destroy	resource.destroy

Sometimes you may only need to handle a subset of the resource actions:

```
php artisan controller:make PhotoController --only=index,show

php artisan controller:make PhotoController --except=index
```

And, you may also specify a subset of actions to handle on the route:

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

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

By default, all resource controller actions have a route name; however, you can override these names by passing a `names` array with your options:

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

Handling Nested Resource Controllers

To "nest" resource controllers, use "dot" notation in your route declaration:

```
Route::resource('photos.comments', 'PhotoCommentController');
```

This route will register a "nested" resource that may be accessed with URLs like the following: `photos/{photoResource}/comments/{commentResource}`.

```
class PhotoCommentController extends BaseController {

    public function show($photoId, $commentId)
    {
        //
    }

}
```

Adding Additional Routes To Resource Controllers

If it becomes necessary for you to add additional routes to a resource controller beyond the default resource routes, you should define those routes before your call to `Route::resource`:

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

Handling Missing Methods

When using `Route::controller`, a catch-all method may be defined which will be called when no other matching method is found on a given controller. The method should be named `missingMethod`, and receives the method and parameter array for the request:

Defining A Catch-All Method

```
public function missingMethod($parameters = array())
{
    //
}
```

If you are using resource controllers, you should define a `__call` magic method on the controller to handle any missing methods.

错误与日志

配置文件

日志处理程序注册在启动文件 `app/start/global.php` 文件里面。日志都默认储存在一个单独的文件中，您可以依照实际需求自定义日志。因为 Laravel 使用了非常流行的 Monolog 日志库，您可以利用 Monolog 提供的多种方式管理你的日志。

例如，如果您想每天使用一个文件记录日志而不是使用单独的庞大文件，您可以照着下面的例子更改开始配置文件：

```
$logFile = 'laravel.log';

Log::useDailyFiles(storage_path().'/logs/'.$logFile);
```

错误显示

错误显示默认为开启，意味着当错误发生时，将有错误页面显示详细的调用追踪和错误信息。您可以关掉错误显示的选项，把`app/config/app.php`里的`debug`选项改成`false`。

> 注意：强烈建议在生产环境中关掉错误显示。

错误处理

`app/start/global.php`里默认有一个处理所有异常的异常处理程序：

```
App::error(function(Exception $exception)
{
    Log::error($exception);
});
```

这是最基本的异常处理程序，然而您可以依照需求设定更多异常处理程序。异常处理程序会依照异常的类型提示(type-hint)被调用。例如，您可以创造一个只处理`RuntimeException`的异常处理程序：

```
App::error(function(RuntimeException $exception)
{
    // Handle the exception...
```

```
});
```

如果异常处理程序回传一个 response，response 会直接回传到浏览器，而其他异常处理程序将不会被调用：

```
App::error(function(InvalidUserException $exception)
{
    Log::error($exception);

    return 'Sorry! Something is wrong with this account!';
});
```

为了监听 PHP fetal errors，您可以利用`App::fatal`方法：

```
App::fatal(function($exception)
{
    //
});
```

如果您有很多异常处理程序，他们应该依照从最通用到最特定的顺序被定义。例如，一个对应处理类型为`Exception`的异常处理程序，应该被定义在一个对应处理自定义异常类型，如`Illuminate\Encryption\DecryptException`的异常处理程序之前。

何处定义异常处理程序

默认上没有注册异常处理程序的地方。Laravel 可以让您自由设定。选择之一是定义程序在`start/global.php`中，一般来说，这是一个让您方便写入任何"bootstrapping" 代码的地方。如果文件变得很拥挤，可以建立一个 `app/errors.php`文件，并且在`start/global.php`中引入。第三个选择是建立 service provider 以注册程序。再一次强调，这个问题并没有正确的答案，您直接选择一个让您觉得舒适的地方注册异常处理程序即可。

HTTP 异常处理

一些异常处理表示来自服务器的 HTTP 错误码，例如可能是「找不到页面」错误(404)，未授权错误(401)，或甚至是工程师导致的500错误。使用下列方法以回传这些回应：

```
App::abort(404);
```

或是您可以选择提供一个回应：

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

您可以在请求回应的生命周期中任何时间点使用这个方法。

404错误处理

您可以注册一个错误处理程序处理所有"404 Not Found"错误，让您可以简单的回传自定义的404错误页面。

```
App::missing(function($exception)
{
    return Response::view('errors.missing', array(), 404);
});
```

日志

Laravel 提供一个建立在强大的 Monolog 上的日志工具。Laravel 默认设定在应用程序里建立单一日志文件，这个文件储存在`app/storage/logs/laravel.log`。您可以像下面这样写入信息：

```
Log::info('This is some useful information.');

Log::warning('Something could be going wrong.');

Log::error('Something is really going wrong.');
```

日志工具提供了七种定义在 RFC 5424 的级别：**debug**, **info**, **notice**, **warning**, **error**, **critical**, and **alert**

可以在传入上下文相关的数组到 log 的方法里：

```
Log::info('Log message', array('context' => 'Other helpful information'));
```

Monolog 提供很多额外的方法可以记录。若有需要，您可以使用 Laravel 里使用的 Monolog 实例：

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

您可以注册事件捕捉所有传到日志的信息：

注册日志监听程序

```
Log::listen(function($level, $message, $context)
{
    //
});
```

深度历险

认证与安全性

设定

Laravel 的目标就是要让实现认证机制变得简单。事实上，几乎所有的设置默认就已经完成了。有关认证的配置文件都放在 `app/config/auth.php` 里，而在这些文件里也都包含了良好的注释描述每一个选项的所对应的认证行为及动作。

Laravel 默认在 `app/models` 文件夹内就包含了一个使用 Eloquent 认证驱动的`User` 模型。请记得在建立模型结构时，密码字段至少要有 60 个字串宽度。

假如您的应用程序并不是使用 Eloquent，您也可以使用 Laravel 的查询构造器做 `database` 认证驱动。

> 注意：在开始之前，请先确认您的 `users` (或其他同义) 数据库表包含一个名为 `remember_token` 长度为100的string类型、可接受 null 的字段。这个字段将会被用来储存「记住我」的 session token。

储存密码

Laravel 的 `Hash` 类提供了安全的 Bcrypt 哈希演算法：

对密码加密

```
$password = Hash::make('secret');
```

验证密码

```
if (Hash::check('secret', $hashedPassword))
{
    // The passwords match...
```

```
}
```

确认密码是否需要重新加密

```
if (Hash::needsRehash($hashed))
{
    $hashed = Hash::make('secret');
}
```

用户认证

您可以使用 `Auth::attempt` 方法来验证用户成功登录之前的信息确认：

```
if (Auth::attempt(array('email' => $email, 'password' => $password)))
{
    return Redirect::intended('dashboard');
}
```

需提醒的是 `email` 并不是一个必要的字段，在这里仅用于示范。您可以使用数据库里任何类似于「用户名称」的字段做为帐号的唯一标识。若用户尚未登入的话，认证筛选器会使用 `Redirect::intended` 方法重定向跳转用户至指定的 URL。我们可指定一个备用 URI，当预定重定向跳转位置不存在时使用。

当 `attempt` 方法被调用时，`auth.attempt` 事件 将会被触发。假如认证成功的话，则 `auth.login` 事件会接着被触发。

判定用户是否已登入

判定一个用户是否已经登入您的应用程序，您可以使用 `check` 这个方法：

```
if (Auth::check())
{
    // The user is logged in...
}
```

认证一个用户并且「记住」他

假如您想要在您的应用程序内提供「记住我」的选项，您可以在 `attempt` 方法的第二个参数复制为 `true`，这样就可以保留用户的认证身份 (或直到他手动登出为止)。当然，您的 `users` 数据库表必需包括一个字串类型的 `remember_token` 字段来储存「记住我」的标记。

```
if (Auth::attempt(array('email' => $email, 'password' => $password), true))
{
    // The user is being remembered...
}
```

注意：假如 `attempt` 方法回传 `true`，则表示用户已经登入您的应用程序。

通过记住我来认证用户

假如您让用户通过「记住我」的方式来登入，则您可以使用 `viaRemember` 方法来判定用户是否拥有「记住我」cookie 来认证用户登入：

```
if (Auth::viaRemember())
{
    //
}
```

条件认证用户

在认证过程中，您可能会想要增加额外的认证条件：

```
if (Auth::attempt(array('email' => $email, 'password' => $password, 'active' => 1)))
{
    // The user is active, not suspended, and exists.
}
```

> 注意：为了增加对 session 的保护，用户的 session ID 将会在用户认证完成后自动重新产生。

取得已登入的用户信息

当用户完成认证后，您就可以通过模型来取得相关的数据：

```
$email = Auth::user()->email;
```

取得登入用户的 ID，您可以使用 `id` 方法：

```
$id = Auth::id();
```

若想要通过用户的 ID 来登入应用程序，可直接使用 `loginUsingId` 方法：

```
Auth::loginUsingId(1);
```

验证用户信息而不要登入

`validate` 方法可以让您验证用户信息而不真的登入应用程序：

```
if (Auth::validate($credentials))
{
    //
}
```

在单一请求内登入用户

您也可以使用 `once` 方法来让用户在单一请求内登入。不会有任何 session 或 cookie 被产生：

```
if (Auth::once($credentials))
{
    //
}
```

将用户登出

```
Auth::logout();
```

手动登入用户

假如您需要将一个已存在的用户实例登入您的应用程序，只需要简单的调用 `login` 方法，并且将该实例作为参数传入即可：

```
$user = User::find(1);

Auth::login($user);
```

这个方式和用用户帐号密码来 `attempt` 的功能是一样的。

保护路由

路由过滤器可让用户仅能访问特定的链接。Laravel 默认提供 `auth` 过滤器，其被定义在 `app/filters.php` 文件内。

保护特定路由

```
Route::get('profile', array('before' => 'auth', function()
{
    // Only authenticated users may enter...
}));
```

CSRF 保护

Laravel 提供一个简易的方式来保护您的应用程序免于跨站攻击。

在表单内引入 **CSRF** 标记

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

验证表单的 **CSRF** 标记

```
Route::post('register', array('before' => 'csrf', function()
{
    return 'You gave a valid CSRF token!';
```

```
}));
```

HTTP 简易认证

HTTP 简易认证提供了一个快速的方式来认证用户而不用特定设置一个「登入」页。在您的路由内设定 `auth.basic` 过滤器则可启动这个功能：

用 **HTTP** 简易认证保护您的路由

```
Route::get('profile', array('before' => 'auth.basic', function()
{
    // Only authenticated users may enter...
}));
```

`basic` 过滤器默认将会使用 `email` 字段来做用户认证，假如您想要使用其他字段来做认证的话，可在您的 `app/filters.php` 文件内将想要拿来做认证的字段当成第一个参数传给 `basic` 方法：

```
Route::filter('auth.basic', function()
{
    return Auth::basic('username');
});
```

设定无状态的 **HTTP** 简易过滤器

一般在实现 API 认证时，往往会想要使用 HTTP 简易认证而不要产生任何 session 或 cookie。我们可以通过定义一个过滤器并回传 `onceBasic` 方法来达成：

```
Route::filter('basic.once', function()
{
    return Auth::onceBasic();
});
```

假如您是使用 PHP FastCGI，HTTP 简易认证默认是无法正常运作的。请在您的 `.htaccess` 文件内新增以下代码来启动这个功能：

```
RewriteCond %{HTTP:Authorization} ^(.+)$
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
```

忘记密码与重设

模型与数据库表

大多数的web应用程序都会提供用户忘记密码的功能。为了不让开发者重复实现这个功能，Laravel 提供了方便的方法来发送忘记密码通知及密码重设的功能。在开始之前，请先确认您的 `User` 模型实现了 `Illuminate\Auth\Reminders\RemindableInterface` 接口。当然，默认 Laravel 的

`User` 模型本身就已实现，并且引入`Illuminate\Auth\Reminders\RemindableTrait` 来包括所有需要实现的接口方法。

实现 **RemindableInterface**

```
use Illuminate\Auth\Reminders\RemindableTrait;
use Illuminate\Auth\Reminders\RemindableInterface;

class User extends Eloquent implements RemindableInterface {

    use RemindableTrait;

}
```

生成 **Reminder** 数据库表迁移

接下来，我们需要生成一个数据库表来储存重设密码标记。为了产生这个数据库表的迁移文件，需要执行 `auth:reminders-table` artisan 命令：

```
php artisan auth:reminders-table

php artisan migrate
```

密码重设控制器

然后我们已经准备好生成密码重设控制器，您可以使用 `auth:reminders-controller` artisan 命令来自动生成这个控制器，它会在您的 `app/controllers` 文件夹内创建一个 `RemindersController.php` 文件。

```
php artisan auth:reminders-controller
```

产生出来的控制器已经具备 `getRemind` 方法来显示您的忘记密码表单。您所需要做就是建立一个 `password.remind` 视图。这个视图需要具备一个 `email` 字段的表单，且这个表单应该要 POST 到 `RemindersController@postRemind` 动作。

一个简单的 `password.remind` 表单视图应该看起来像这样：

```
<form action="{{ action('RemindersController@postRemind') }}" method="POST">
    <input type="email" name="email">
    <input type="submit" value="Send Reminder">
</form>
```

除了 `getRemind` 外，控制器还包括了一个 `postRemind` 方法来处理发送忘记密码通知信给您的用户。这个方法会预期在 `POST` 参数内会有 `email` 字段。假如忘记密码通知信成功的寄发给用户，则会有一个 `status` 信息被暂存在 session 内；假如寄发失败的话，则取而代之的会有一个 `error` 信

息被暂存。

在 `postRemind` 方法内，您可以在发送出去前修改信息的内容：

```
Password::remind(Input::only('email'), function($message)
{
    $message->subject('Password Reminder');
});
```

您的用户将会收到一封电子邮件内有一个重设密码的链接指向 `getReset` 控制器方法。忘记密码标记是用来验证密码重设程序是否正确，也会传递给对应的控制器方法。这个动作已经设定会回传一个 `password.reset` 视图。这个 `token` 会被传递给视图，而您需要将这个 `token` 放在一个隐形的表单字段内。另外，您的重设密码表单应该包括 `email`、`password` 和 `password_confirmation` 字段。这个表单应该 POST 到 `RemindersController@postReset` 方法。

一个 `password.reset` 视图表单应该看起来像这样：

```
<form action="{{ action('RemindersController@postReset') }}" method="POST">
    <input type="hidden" name="token" value="{{ $token }}">
    <input type="email" name="email">
    <input type="password" name="password">
    <input type="password" name="password_confirmation">
    <input type="submit" value="Reset Password">
</form>
```

最后，`postReset` 方法则是专职处理重设过后的密码。在这个控制器方法里，Closure 传递 `Password::reset` 方法并且设定 `User` 内的 `password` 属性后调用 `save` 方法。当然，这个 Closure 会假定您的 `User` 模型是一个 Eloquent 模型 。当然，您可以自由地修改这个 Closure 内容来符合您的应用程序数据库储存方式。

假如密码成功的重设，则用户会被重定向跳转至您的应用程序根目录。同样的，您可以自由的更改重定向跳转的 URL；假如密码重设失败的话，用户会被重定向跳转至重设密码表单页，而且会有 `error` 信息被暂存在 session 内。

密码验证

默认 `Password::reset` 方法会验证密码符合且大于等于六个字串。您可以用 `Password::validator` 方法 (接受 Closure) 来调整这些默认值。通过这个 Closure，您可以用任何方式来做密码验证。需提醒的是，您不一定要验证密码是否符合，因为框架自动会帮您完成这项工作。

```
Password::validator(function($credentials)
{
    return strlen($credentials['password']) >= 6;
});
```

> 注意：密码重设标记默认会在一个小时后过期。您可以通过 `app/config/auth.php` 案内的 `reminder.expire` 选项来调整这个设定。

加密

Laravel 通过 mcrypt PHP 扩展来提供 AES 强度的加密算法：

加密一个值

```
$encrypted = Crypt::encrypt('secret');
```

> 注意：记得在 `app/config/app.php` 文件里设定一个 16, 24 或 32 字串的随机字做 `key`，否则这个加密算法结果将不够安全。

解密一个值

```
$decrypted = Crypt::decrypt($encryptedValue);
```

设定暗号及模式

您可以设定加密器的暗号及模式：

```
Crypt::setMode('ctr');

Crypt::setCipher($cipher);
```

认证驱动

Laravel 默认提供 `database` 及 `eloquent` 两种认证驱动。假如您需要更多有关增加额外认证驱动的详细信息，请参考 认证扩充文件

Laravel Cashier

介绍

Laravel Cashier 提供语义化，流畅的接口和 Stripe 的订购管理服务连接。它几乎处理了所有复杂的订购管理相关逻辑。除了基本的订购管理外，Cashier 还可以处理折扣券，订购转换，管理订购「数量」、服务有效期限，甚至产生收据的 PDF 。

配置文件

Composer

首先，把 Cashier 扩展包加到 `composer.json`：

```
"laravel/cashier": "~2.0"
```

服务提供者

接下来，在 `app` 配置文件注册 `Laravel\Cashier\CashierServiceProvider`。

迁移

使用 Cashier 前，我们需要增加几个字段到数据库。别担心，您可以使用 `cashier:table` Artisan 命令，建立迁移文件来新增必要字段。例如，要增加字段到 users 数据库表，使用 `php artisan cashier:table users` 。建立完迁移文件后，只要执行 `migrate` 命令即可。

设定模型

接下来，把 BillableTrait 和日期字段参数加到模型 (model) 里：

```
use Laravel\Cashier\BillableTrait;
use Laravel\Cashier\BillableInterface;

class User extends Eloquent implements BillableInterface {

    use BillableTrait;

    protected $dates = ['trial_ends_at', 'subscription_ends_at'];

}
```

Stripe Key

最后，在起始文件里加入 Stripe key：

```
User::setStripeKey('stripe-key');
```

订购方案

当有了模型实例，您可以很简单的处理客户订购的 Stripe 里的方案：

```
$user = User::find(1);

$user->subscription('monthly')->create($creditCardToken);
```

如果您想在处理订购的时候使用折扣券，可以使用 `withCoupon` 方法：

```
$user->subscription('monthly')
     ->withCoupon('code')
     ->create($creditCardToken);
```

`subscription` 方法会自动建立与 Stripe 的交易，以及将 Stripe customer ID 和其他相关帐款信息更新到数据库。如果您的方案有在 Stripe 设定试用期，试用到期时间也会自动记录起来。

如果您的方案有试用期间，但是没有在 Stripe 里设定，您必须在订购后手动储存试用到期时间。

```
$user->trial_ends_at = Carbon::now()->addDays(14);

$user->save();
```

自定义额外用户详细数据

如果您想自定义额外的顾客详细数据，您可以将数组作为 `create` 方法的第二个参数传入：

```
$user->subscription('monthly')->create($creditCardToken, [
    'email' => $email, 'description' => 'Our First Customer'
]);
```

想知道更多 Stripe 支持的额外字段，请阅读 Stripe 的线上文件 建立客户.

免信用卡试用

如果您提供免信用卡试用服务，把 `cardUpFront` 属性设为 `false`：

```
protected $cardUpFront = false;
```

建立帐号时，记得把试用到期时间记录起来：

```
$user->trial_ends_at = Carbon::now()->addDays(14);

$user->save();
```

订购转换

使用 `swap` 方法可以把用户转换到新的订购方案：

```
$user->subscription('premium')->swap();
```

如果用户还在试用期间，试用服务会跟之前一样可用。如果订单有「数量限制」，也会和之前一样。

订购数量

有时候订购行为会跟「数量限制」有关。例如，您的应用程序可能会跟用户每个月收取 $10 元。您可以使用 `increment` 和 `decrement` 方法简单的调整订购数量：

```
$user = User::find(1);

$user->subscription()->increment();

// Add five to the subscription's current quantity...
$user->subscription()->increment(5);

$user->subscription->decrement();

// Subtract five to the subscription's current quantity...
$user->subscription()->decrement(5);
```

取消订购

取消订购相当简单：

```
$user->subscription()->cancel();
```

当客户取消订购时，Cashier 会自动更新数据库的 `subscription_ends_at` 字段。这个字段会被用来判断 `subscribed` 方法是否该回传 `false` 。例如，如果顾客在三月一号取消订购，但是服务可以使用

到三月五号为止，那么 `subscribed` 方法在三月五号前都会传回 `true` 。

恢复订购

如果您想要恢复客户之前取消的订购，使用 `resume` 方法：

```
$user->subscription('monthly')->resume($creditCardToken);
```

如果客户取消订购后，在服务过期前恢复，他们不用在当下付款。他们的订购服务会重新激活，而付款时间还是依据之前的付款周期。

确认订购状态

要确认用户是否订购了服务，使用 `subscribed` 方法：

```
if ($user->subscribed())
{
    //
}
```

`subscribed` 方法很适合用在路由过滤：

```
Route::filter('subscribed', function()
{
    if (Auth::user() && ! Auth::user()->subscribed())
    {
        return Redirect::to('billing');
    }
});
```

您可以使用 `onTrial` 方法，确认用户是否还在试用期间：

```
if ($user->onTrial())
{
    //
}
```

要确认用户是否曾经订购但是已经取消了服务，可已使用 `cancelled` 方法：

```
if ($user->cancelled())
{
    //
}
```

您可能想确认用户是否已经取消订单，但是服务还没有到期。例如，如果用户在三月五号取消了订购，但是服务会到三月十号才过期。那么用户到三月十号前都是有效期间。注意， `subscribed` 方法在过期前都会回传 `true` 。

```
if ($user->onGracePeriod())
{
    //
}
```

`everSubscribed` 方法可以用来确认用户是否订购过您的方案：

```
if ($user->everSubscribed())
{
    //
}
```

`onPlan` 方法可以用方案 ID 来确认用户是否订购某方案：

```
if ($user->onPlan('monthly'))
{
    //
}
```

处理交易失败

如果顾客的信用卡过期了呢？无需担心，Cashier 包含了 Webhook 控制器，可以帮您简单的取消顾客的订单。只要把路由注册到控制器：

```
Route::post('stripe/webhook', 'Laravel\Cashier\WebhookController@handleWebhook');
```

失败的交易会经由控制器捕捉并进行处理。控制器会进行至多三次再交易尝试，都失败后才会取消顾客的订单。上面的 `stripe/webhook` URI 只是一个例子，您必须使用设定在 Stripe 里的 URI 才行。

处理其它 Stripe Webhooks

如果您想要处理额外的 Stripe webhook 事件，可以继承 Webhook 控制器。您的方法名称要对应到 Cashier 预期的名称，尤其是方法名称应该使用 `handle` 前缀，后面接着您想要处理的 Stripe webhook 。例如，如果您想要处理 `invoice.payment_succeeded` webhook ，您应该增加一个 `handleInvoicePaymentSucceeded` 方法到控制器。

```
class WebhookController extends Laravel\Cashier\WebhookController {

    public function handleInvoicePaymentSucceeded($payload)
    {
        // Handle The Event
    }

}
```

> 注意 除了更新您数据库里的订购信息以外，Webhook 控制器也可能会经由 Stripe API 取消

> 您的订购。

收据

您可以很简单的经由 `invoices` 方法拿到客户的收据数组：

```
$invoices = $user->invoices();
```

您可以使用这些辅助方法，列出收据的相关信息给客户看：

```
{{ $invoice->id }}

{{ $invoice->dateString() }}

{{ $invoice->dollars() }}
```

使用 `downloadInvoice` 方法产生收据的 PDF 下载。看吧，它的调用是多么的简单：

```
return $user->downloadInvoice($invoice->id, [
    'vendor'  => 'Your Company',
    'product' => 'Your Product',
]);
```

缓存

配置

Laravel对不同的缓存系统提供了统一的API. 请在 `app/config/cache.php` 文件中配置缓存信息. 你可以在此文件中指定整个应用程序默认使用哪种缓存驱动. Laravel支持系统以外的流行后端缓存如 Memcached 和 Redis.

缓存配置文件还包含了其他配置项, 都已记录在文件中, 请详细阅读. 默认情况下, Laravel被配置为使用 'file' 缓存驱动, 它将缓存数据序列化的存储在文件系统中, 对于大型的应用, 我们建议你使用内存缓存, 如 Memcached 或 APC.

缓存用法

将某一条数据存入缓存

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

使用**Carbon**对象并设置过期时间

```
$expiresAt = Carbon::now()->addMinutes(10);

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

当一条数据不在缓存中才存储

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

如果该数据实际上 已经添加 到缓存中, 那么 `add` 方法将返回 `true` . 否则, 此方法将返回 `false`.

检查缓存中是否存在某个**key**对应的数据

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

从缓存中取得一条**key**对应的数据

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

从缓存中取得数据, 如果不存在, 则返回指定的默认值

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

$value = Cache::get('key', function() { return 'default'; });
```

在缓存中永久存储数据

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

有时候你可能希望从缓存中取得数据, 并且数据不存在时还可以存储一个默认值, 这时可以使用 `Cache::remember` 方法:

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

你还可以结合使用 `remember` 和 `forever` 方法:

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

注意: 所有项目都是序列化的存储在缓存中, 所以你可以自由存储任何类型的数据.

在缓存中拉出一条数据

如果你需要从缓存中先取出一条数据, 然后删除它, 你可以使用 `pull` 方法:

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

从缓存中删除某条数据

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

递增 & 递减

除了 `file` 和 `database` , 其他驱动都支持 `increment` and `decrement` 操作:

递增某一个值

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

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

递减某一个值

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

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

缓存标签

> 注意: 使用 `file` 或 `database` 缓存驱动时不支持缓存标签. 此外, 在使用多个缓存标签时它们将存储为 "forever", 使用一个如 `memcached` 的驱动性能将会是最好, 它会自动清除过时的记录.

访问一个标记的缓存

缓存标签允许你在缓存中标记相关的项目, 刷新指定名称标记的所有缓存. 要访问一个标记的缓存, 可以使用 `tags` 方法.

你可以通过传递一个标记名称的有序列表作为参数, 或是一个标记名称的有序数数组, 来存储一个标记的缓存:

```
Cache::tags('people', 'authors')->put('John', $john, $minutes);

Cache::tags(array('people', 'artists'))->put('Anne', $anne, $minutes);
```

你可以在任何缓存存储方法中组合使用标签, 包括`remember`, `forever`, 和 `rememberForever`. 你也可以从标记的缓存中访问已缓存的项目, 以及使用其它的缓存方法, 如 `increment` 和 `decrement`.

从标记的缓存中访问项目

通过与保存时所用相同的标签, 作为参数列表来访问一个标记的缓存.

```
$anne = Cache::tags('people', 'artists')->get('Anne');

$john = Cache::tags(array('people', 'authors'))->get('John');
```

你可以通过一个名称或名称的列表来刷新所有的项目. 例如, 下面的语句将移除标记有任何'people', 'authors', 或两者都有的所有缓存. 所以, 无论是“Anne”和“John”将从缓存中被移除:

```
Cache::tags('people', 'authors')->flush();
```

相比之下, 下面的语句将移除标记中只包含 'authors' 的缓存, 因此 "John" 将被移除, 但不影响

"Anne".

```
Cache::tags('authors')->flush();
```

数据库缓存

当使用'数据库'缓存驱动时, 你将需要设置一个表来存储缓存数据. 以下是一个使用 `Schema` 声明的例子:

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

扩展框架

介绍

Laravel 提供许多可扩展的地方让您自定义框架核心组件的行为，或甚至完全地取代它们。例如，`HasherInterface` 接口定义了哈希工具，您可以基于应用程序的需求来实现它。您也可以扩展 `Request` 对象，让您加入自己的便利 "辅助" 方法。甚至您可以加入全新的认证、缓存和 session 驱动！

Laravel 组件一般以两种方式来扩展：在 IoC 容器里绑定新的实现，或用 "工厂" 设计模式实现的 `Manager` 类来注册扩展。在这个章节我们将会探索多种扩展框架的方法和查看必要的代码。

> 备注：记住，Laravel 组件通常用两种方法的其中之一来扩展：IoC 绑定和 `Manager` 类。管理者类作为 "工厂" 设计模式的实现，并负责实体化基于驱动的工具，例如：缓存和 session。

管理者和工厂

Laravel 有几个 `Manager` 类用来管理建立基于驱动的组件。这些类包括缓存、session、认证和队列组件。管理者类负责基于应用程序的设定建立一个特定的驱动实现。例如，`CacheManager` 类可以建立 APC、Memcached 、文件和各种其他的缓存驱动实现.

这些管理者都拥有 `extend` 方法，它可以简单地用来注入新的驱动解析功能到管理者。我们将会在下面随着如何注入自定义驱动支持给它们的例子，涵盖这些管理者的内容。

> 备注：建议花点时间来探索 Laravel 附带的各种 `Manager` 类，例如：`CacheManager` 和 `SessionManager`。看过这些类将会让您更彻底了解 Laravel 表面下是如何运作。所有的管理者类继承 `Illuminate\Support\Manager` 基底类，它提供一些有用、常见的功能给每一个管理者。

在哪里扩展

这份文件涵盖如何扩展各种 Laravel 的组件，但是您可能想知道要在哪里放置您的扩展代码。就

像其他大部分的启动代码，您可以自由的在您的 `start` 文件放置一些扩展，缓存和认证扩展是这个方法的好例子。其他扩展，像 `Session`，必须放置到服务提供者的 `register` 方法中，因为他们在请求生命周期的非常早期就被需要。

缓存

为了扩展 Laravel 缓存工具，我们将会对 `CacheManager` 使用 `extend` 方法，它被用来绑定一个自定义驱动解析器到管理者，并且是全部的管理者类通用的。例如，注册一个新的缓存驱动命名为 "mongo"，我们将执行以下操作：

```
Cache::extend('mongo', function($app)
{
    // Return Illuminate\Cache\Repository instance...
});
```

传递到 `extend` 方法的第一个参数是驱动的名称。这将会对应到您的 `app/config/cache.php` 配置文件里的 `driver` 选项。第二个参数是个应该回传 `Illuminate\Cache\Repository` 实体的闭包。`$app` 实体将被传递到闭包，它是 `Illuminate\Foundation\Application` 和 IoC 容器的实体。

要建立我们的自定义缓存驱动，首先需要实现 `Illuminate\Cache\StoreInterface` 接口。所以，我们的 MongoDB 缓存实现将会看起来像这样：

```
class MongoStore implements Illuminate\Cache\StoreInterface {

    public function get($key) {}
    public function put($key, $value, $minutes) {}
    public function increment($key, $value = 1) {}
    public function decrement($key, $value = 1) {}
    public function forever($key, $value) {}
    public function forget($key) {}
    public function flush() {}

}
```

我们只需要使用 MongoDB 连接来实现这些方法。当我们的实现完成，我们可以完成我们的自定义驱动注册：

```
use Illuminate\Cache\Repository;

Cache::extend('mongo', function($app)
{
    return new Repository(new MongoStore);
});
```

就像您可以在上面的例子看到的，当在建立自定义缓存驱动的时候，您可以使用基本的 `Illuminate\Cache\Repository` 类。通常不需要建立您自己的储存库类。

如果您在考虑要把您的自定义缓存驱动代码放在哪里，请考虑把它放上 Packagist！或者，您可以在您的应用程序的主要文件夹中建立一个 `Extensions` 命名空间。例如，如果应用程序命名为 `Snappy`，您可以在 `app/Snappy/Extensions/MongoStore.php` 放置缓存扩展。然而，必须牢记在心 Laravel 没有严格的应用程序架构，您可以依照喜好自由的组织您的应用程序。

> 备注：如果您曾经考虑要在哪放置一段代码，请总是考虑服务提供者。就像我们曾经讨论过的，用服务提供者来组织框架扩展是个组织您的代码的好方法.

Session

以自定义 session 驱动来扩展 Laravel 跟扩展缓存系统一样简单。再一次的，我们将会使用 `extend` 方法来注册我们的自定义代码：

```
Session::extend('mongo', function($app)
{
    // Return implementation of SessionHandlerInterface
});
```

在哪里扩展 **Session**

Session 需要用与其他扩展如 Cache 和 Auth 不同地方式扩展。因为 sessions 在请求生命周期的非常早期就被启用，注册扩展在 `start` 文件将会太晚。作为替代，将会需要服务提供者 。您应该放置您的 session 扩展代码在您的服务提供者的 `register` 方法，并且提供者应该被放置在 `providers` 设定数组里、默认的 `Illuminate\Session\SessionServiceProvider` 下面。

实现 **Session** 扩展

要注意我们的自定义缓存驱动应该实现 `SessionHandlerInterface`。这个接口在 PHP 5.4+ 核心被引入。如果您使用 PHP 5.3，Laravel 将会为您定义这个接口，所以您可以向下兼容。这个接口只包含少数我们需要实现的简单方法。一个空的 MongoDB 实现将会看起来像这样：

```
class MongoHandler implements SessionHandlerInterface {

    public function open($savePath, $sessionName) {}
    public function close() {}
    public function read($sessionId) {}
    public function write($sessionId, $data) {}
    public function destroy($sessionId) {}
    public function gc($lifetime) {}

}
```

因为这些方法不像缓存 `StoreInterface` 一样容易理解，让我们快速地看过这些方法做些什么：

- `open` 方法通常会被用在基于文件的 session 储存系统。因为 Laravel 附带一个 `file` session

驱动，您几乎不需要在这个方法放任何东西。您可以让它留空。PHP 要求我们去实现这个方法，事实上明显的是个差劲的接口设计 (我们将会晚点讨论它)。

- `close` 方法，就像 `open` 方法，通常也可以忽略。对大部份的驱动来说，并不需要它。
- `read` 方法应该回传与给定 `$sessionId` 关联的 session 数据的字串形态。当您的驱动取回或储存 session 数据时不需要做任何序列化或进行其他编码，因为 Laravel 将会为您进行序列化。
- `write` 方法应该写入给定 `$data` 字串与 `$sessionId` 的关联到一些永久存储系统，例如 MongoDB、Dynamo、等等。
- `destroy` 方法应该从永久存储移除与 `$sessionId` 关联的数据。
- `gc` 方法应该销毁所有比给定 `$lifetime` UNIX 时间戳记还旧的 session 数据。对于会自己到期的系统如 Memcached 和 Redis，这个方法可以留空。

当 `SessionHandlerInterface` 被实现完成，我们已经准备好用 Session 管理者注册它：

```
Session::extend('mongo', function($app)
{
    return new MongoHandler;
});
```

当 session 驱动已经被注册，我们可以在我们的 `app/config/session.php` 配置文件使用 `mongo` 驱动 。

> 备注: 记住，如果您写了个自定义 session 处理器，请在 Packagist 分享它！

认证

认证可以通过与缓存和 session 工具相同的方法来扩展。再一次的，我们将会使用我们已经熟悉的 `extend` 方法：

```
Auth::extend('riak', function($app)
{
    // Return implementation of Illuminate\Auth\UserProviderInterface
});
```

`UserProviderInterface` 实现只负责从持久化存储中抓取 `UserInterface` 实现，例如：MySQL、Riak，等等。这两个接口让 Laravel 认证机制无论用户数据如何储存或用什么种类的类来代表它都能继续运作。

让我们来看一下 `UserProviderInterface`：

```
interface UserProviderInterface {

    public function retrieveById($identifier);
    public function retrieveByToken($identifier, $token);
```

```
    public function updateRememberToken(UserInterface $user, $token);
    public function retrieveByCredentials(array $credentials);
    public function validateCredentials(UserInterface $user, array $credentials);

}
```

`retrieveById` 函数通常接收一个代表用户的数字键，例如：MySQL 数据库的自动递增 ID。符合 ID 的 `UserInterface` 实现应该被取回并被方法回传。

`retrieveByToken` 函数通过用户唯一的 `$identifier` 和储存在 `remember_token` 字段的 "记住我" `$token` 取得用户。跟前面的方法一样，应该回传 `UserInterface` 实现。

`updateRememberToken` 方法用新的 `$token` 更新 `$user` 的 `remember_token` 字段。新 token 的值 可以是在 "记住我" 成功地登入时生成的一个新的 token，或当用户登出时变为 null.

`retrieveByCredentials` 方法接收当尝试登入应用程序时，传递到 `Auth::attempt` 方法的凭证数组。这个方法应该接着 "查询" 背后的永久存储，看用户是否符合这些凭证。这个方法通常会对 `$credentials['username']` 用 "where" 条件查询。这个方法不应该尝试做任何密码验证或认证。

`validateCredentials` 方法应该通过比较给定 `$user` 与 `$credentials` 来验证用户。举例来说，这个方法可以比较 `$user->getAuthPassword()` 字串跟 `Hash::make` 后的 `$credentials['password']`。

现在我们已经看过 `UserProviderInterface` 的每个方法，接着我们来看一下 `UserInterface`。记住，提供者应该从 `retrieveById` 和 `retrieveByCredentials` 方法回传这个接口的实现：

```
interface UserInterface {

    public function getAuthIdentifier();
    public function getAuthPassword();

}
```

这个接口很简单。`getAuthIdentifier` 方法应该回传用户的 "主键"。在 MySQL 后台，同样，这将会是个自动递增的主键。`getAuthPassword` 应该回传用户哈希过的密码。这个接口让认证系统可以与任何用户类一起运作，无论您使用什么 ORM 或储存抽象层。默认，Laravel 包含一个实现这个接口的 `User` 类在 `app/models` 文件夹里，所以您可以参考这个类当作实现的例子。

最后，当我们已经实现了 `UserProviderInterface`，我们准备好用 `Auth` facade 来注册我们的扩展：

```
Auth::extend('riak', function($app)
{
    return new RiakUserProvider($app['riak.connection']);
});
```

在您用 `extend` 方法注册驱动之后，在您的 `app/config/auth.php` 配置文件切换到新驱动。

基于 IoC 的扩展

几乎每个 Laravel 框架引入的服务提供者会绑定对象到 IoC 容器中。您可以在 `app/config/app.php` 配置文件中找到应用程序的服务提供者清单。当您有时间的时候，您应该浏览一下这里面每一个提供者的源代码。通过这样做，您将会对每一个提供者为框架增加了什么功能有更多的了解，以及各种服务用什么键来绑定到 IoC 容器。

举个例子，`HashServiceProvider` 绑定一个 `hash` 键到 IoC 容器，它将解析成 `Illuminate\Hashing\BcryptHasher` 实例。您可以轻松地通过在您的应用程序中重写这个 IoC 绑定，扩展并重写这个类。例如：

```
class SnappyHashProvider extends Illuminate\Hashing\HashServiceProvider {

    public function boot()
    {
        App::bindShared('hash', function()
        {
            return new Snappy\Hashing\ScryptHasher;
        });

        parent::boot();
    }

}
```

要注意的是这个类扩展 `HashServiceProvider`，不是默认的 `ServiceProvider` 基底类。当您扩展了服务提供者，在您的 `app/config/app.php` 配置文件把 `HashServiceProvider` 换成您扩展的提供者名称。

这是扩展任何被绑定在容器的核心类的普遍方法。实际上，每个被以这种方式绑定在容器的核心类都可以被重写。再次强调，看过被框架引入的服务提供者将会使您熟悉每个类被绑在容器的哪里，以及它们是用什么键绑定的。这也是可以了解更多关于 Laravel 是如何结合它们的好方法。

扩展请求

因为它是框架非常基础的部件并且在请求周期的非常早期就被实例化，扩展 `Request` 类跟前面的例子有一点不同。

首先，继承 laravel 的 `Request` 基类：

```
<?php namespace QuickBill\Extensions;

class Request extends \Illuminate\Http\Request {

    // Custom, helpful methods here...

}
```

当您扩展了类，打开 `bootstrap/start.php` 文件。这个文件是当应用程序受到请求后非常早被引入的文件之一。需要注意的是第一个被执行的动作是建立 Laravel `$app` 实例：

```
$app = new \Illuminate\Foundation\Application;
```

当一个新的应用程序实例被建立，它将会建立一个新的 `Illuminate\Http\Request` 实例并用 `request` 键把它绑定到 IoC 容器。所以，我们需要一个方法去指定一个应该被用作 "默认" 请求类型的自定义类，对吧？ 并且值得庆幸的是，应用程序实例的 `requestClass` 方法就做了这件事！ 所以，我们可以在 `bootstrap/start.php` 文件的最上面加这行：

```
use Illuminate\Foundation\Application;

Application::requestClass('QuickBill\Extensions\Request');
```

每当您指定了自定义请求类，Laravel 将会在任何建立 `Request` 实体的时候使用这个类，便利地让您总是有一个可用的自定义请求类实例，甚至在单元测试也有！

事件

基本用法

Laravel 的 `Event` 类提供一个简单的观察者实现，允许您在应用程序里订阅与监听事件。

订阅事件

```
Event::listen('auth.login', function($user)
{
    $user->last_login = new DateTime;

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

触发事件

```
$event = Event::fire('auth.login', array($user));
```

订阅有优先顺序的事件

您也可以在订阅事件的时候指定一个优先顺序。有较高优先权的监听者会先被执行，当监听者有一样的优先权时将会依照订阅的顺序执行.

```
Event::listen('auth.login', 'LoginHandler', 10);

Event::listen('auth.login', 'OtherHandler', 5);
```

停止继续传递事件

您有时候会希望停止继续传递事件到其他监听者。您可以通过从监听者回传 `false` 来做到这件事：

```
Event::listen('auth.login', function($event)
{
    // Handle the event...

    return false;
});
```

在哪里注册事件

现在您知道怎么注册事件了，但是您或许会想知道要在 哪里 注册它们。不要担心，这是一个常见的问题。不幸地，这是一个很难回答的问题，因为您几乎可以在任何地方注册事件！但是，这里有一些提示。一样的，您可以在您的其中一个 `start` 文件注册事件，就像其他大部份的启动代码，例如：`app/start/global.php`。

如果您的 `start` 文件变得越来越拥挤，您可以建立一个分离的 `app/events.php` 文件，并从 `start` 文件引入它。这是个简单的解决方案，它保持您的事件注册与剩余的启动代码干净地分离。如果您喜欢基于类的方法，您可以在 服务提供者 注册您的事件。因为这些方法中没有一个是绝对正确的方案，基于您的应用程序大小选择一个让您感到舒服的方法。

通配符监听者

注册通配符事件监听者

当注册事件监听者，您可以使用星号(*) 指定通配符监听者：

```
Event::listen('foo.*', function($param)
{
    // Handle the event...
});
```

这个监听者将会处理所有 `foo.` 开头的事件。

您可以使用 `Event::firing` 方法准确的判定是什么事件被触发：

```
Event::listen('foo.*', function($param)
{
    if (Event::firing() == 'foo.bar')
    {
        //
    }
});
```

使用类作为监听者

在一些案例中，您或许会希望使用类取代闭包来处理事件。类事件监听者将会被 Laravel IoC container 处理，提供依赖注入的全部功能给您的监听者。

注册类监听者

```
Event::listen('auth.login', 'LoginHandler');
```

定义事件监听者类

`LoginHandler` 类默认将会调用 `handle` 方法：

```
class LoginHandler {

    public function handle($data)
    {
        //
    }

}
```

指定被订阅的方法

如果您不希望使用默认的 `handle` 方法, 您可以指定应该被订阅的方法：

```
Event::listen('auth.login', 'LoginHandler@onLogin');
```

事件队列

注册事件队列

使用 `queue` 和 `flush` 方法，您可以把事件加到队列等待触发，但是不立即触发它：

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

您可以执行 "flusher" 并触发全部的事件队列，使用 `flush` 方法：

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

事件订阅者

定义事件订阅者

事件订阅者是个可以从类自身里面订阅多个事件的类。订阅者应该定义 `subscribe` 方法，它将会被传递到事件配送器实例：

```
class UserEventHandler {

    /**
     * Handle user login events.
     */
    public function onUserLogin($event)
    {
        //
    }

    /**
     * Handle user logout events.
     */
    public function onUserLogout($event)
```

```
    {
        //
    }

    /**
     * Register the listeners for the subscriber.
     *
     * @param  Illuminate\Events\Dispatcher  $events
     * @return array
     */
    public function subscribe($events)
    {
        $events->listen('auth.login', 'UserEventHandler@onUserLogin');

        $events->listen('auth.logout', 'UserEventHandler@onUserLogout');
    }

}
```

注册事件订阅者

当订阅者被定义时，它或许会使用 `Event` 类注册。

```
$subscriber = new UserEventHandler;

Event::subscribe($subscriber);
```

您也可以使用 Laravel IoC container 去处理您的订阅者。简单地传递订阅者的名字给 `subscribe` 方法就可以做到：

```
Event::subscribe('UserEventHandler');
```

Facades

介绍

Facades（一种设计模式，通常翻译为外观模式）提供了一个"static"（静态）接口去访问注册到IoC容器中的类 。Laravel 含有很多 facades，你可能不知道其实你在某些地方已经使用过它们了。Laravel的"facades"就像"静态代理方法"，为IoC容器中的类服务，相比传统的静态方法，在保持了更强的可测试性和灵活性的同时，它提供更简洁，更富有表现力的语法。

有时候，你可能想为你的应用程序或包创建自己的 facades，所以，让我们来讨论一下如何开发和使用这些类。

> 注意：在深入 facades 之前，我们强烈建议你多了解一下 Laravel IoC 容器。

说明

在 Laravel 应用程序中，facade 是提供从容器中访问对象的类。`Facade` 类实现了该机制。Laravel的facades，和其他自定义的facades，都需要继承基类`Facade`

你的 facade 类只需要实现一个方法：`getFacadeAccesor` 。`getFacadeAccessor` 方法的工作是定义如何从容器中取得对象。Facades 基类构建了 `__callStatic()` 魔术方法来从 facade 延迟访问取得对象。

因此，当你使用facade调用，类似`Cache::get`，Laravel会从IoC容器取得Cache管理类并调用`get`方法。在技术层上说，Laravel Facades是一种便捷的语法使用Laravel IoC容器作为服务定位器.

实际用例

在以下示例中，调用Laravel缓存系统，咋一看该代码，可能会认为`get`静态方法是在`Cache`类执行。

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

然后，如果我们查看`Illuminate\Support\Facades\Cache`类，你会发现该类没有任何`get`静态方法：

```
class Cache extends Facade {

    /**
     * Get the registered name of the component.
     *
     * @return string
     */
    protected static function getFacadeAccessor() { return 'cache'; }

}
```

`Cache`类继承`Facade`这个基类，并且定义了个`getFacadeAccessor()`方法。注意，该方法的工作是返回绑定到`IoC`的名字。

当用户引用任何在`Cache` facade 中的静态方法，Laravel 从 IoC 容器绑定中取得 `cache`，并且执行请求的对象方法（在该例子中为`get`）。

所以，我们 `Cache::get` 执行可以重写为：

```
$value = $app->make('cache')->get('key');
```

创建Facades

要为自己的应用程序或者包创建一个facade是非常简单的。你只需要做三件事情：

- 一个 IoC 绑定。
- 一个 facade 类。
- 一个 facade 别名配置。

让我们来看个例子。这里，我们定义一个`PaymentGateway\Payment`类。

```
namespace PaymentGateway;

class Payment {

    public function process()
    {
        //
    }

}
```

这个类可以放在`app/models`目录，或者其他任何Composer能够自动载入的位置。

我们需要能够在 IoC 容器中取得该类。所以，让我们增加一个绑定：

```
App::bind('payment', function()
{
    return new \PaymentGateway\Payment;
```

```
});
```

最好注册该绑定的位置是创建一个新的名为`PaymentServiceProvider`服务提供器，并且将该绑定加入到 `register` 方法。接下来你就可以配置 Laravel `app/config/app.php` 配置文件来加载该服务提供器。

接下来，我们就可以创建我们的 facade 类：

```
use Illuminate\Support\Facades\Facade;

class Payment extends Facade {

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

}
```

最后，如果你想，我们可以为我们 facade 设置一个别名到 `app/config/app.php` 配置文件里的 `aliases` 数组。现在，我们能够调用 Payment 类实例的 `process` 方法。

```
Payment::process();
```

关于自动载入别名一些注意点

在`aliases`数组中的有些类接口可能是不可行的，因为PHP不会尝试去自动载入未定义的类型约定。假设`\ServiceWrapper\ApiTimeoutException`别名是`ApiTimeoutException`，如果`catch(ApiTimeoutException $e)` 是在`\ServiceWrapper`命名空间之外使用，它将会永远不会被捕获到，即使真的有一个异常被抛出。一个类似的问题存在在那些进行类型约定的别名类上。唯一的变通方法是摒弃别名，在每个你想使用类型约定的文件的开始的地方使用`use`。

模拟Facades

单元测试是 facades 工作的重要体现。事实上，可测试性是 facedes 存在的主要原因。要了解更多信息，查看文档模拟 facades部分。

Facade 类参考

下面你会找到所有的facade以及其包含的类。这是一个非常有用的工具，可以根据给定的facade快速定位到API文档。适用于IoC 绑定 的也同时给出了其key。

Facade	Class	IoC Binding
App	Illuminate\Foundation\Application	`app`
Artisan	Illuminate\Console\Application	`artisan`

Auth	Illuminate\Auth\AuthManager	`auth`
Auth (Instance)	Illuminate\Auth\Guard	
Blade	Illuminate\View\Compilers\BladeCompiler	`blade.compiler`
Cache	Illuminate\Cache\Repository	`cache`
Config	Illuminate\Config\Repository	`config`
Cookie	Illuminate\Cookie\CookieJar	`cookie`
Crypt	Illuminate\Encryption\Encrypter	`encrypter`
DB	Illuminate\Database\DatabaseManager	`db`
DB (Instance)	Illuminate\Database\Connection	
Event	Illuminate\Events\Dispatcher	`events`
File	Illuminate\Filesystem\Filesystem	`files`
Form	Illuminate\Html\FormBuilder	`form`
Hash	Illuminate\Hashing\HasherInterface	`hash`
HTML	Illuminate\Html\HtmlBuilder	`html`
Input	Illuminate\Http\Request	`request`
Lang	Illuminate\Translation\Translator	`translator`
Log	Illuminate\Log\Writer	`log`
Mail	Illuminate\Mail\Mailer	`mailer`
Paginator	Illuminate\Pagination\Factory	`paginator`
Paginator (Instance)	Illuminate\Pagination\Paginator	
Password	Illuminate\Auth\Reminders\PasswordBroker	`auth.reminder`
Queue	Illuminate\Queue\QueueManager	`queue`
Queue (Instance)	Illuminate\Queue\QueueInterface	
Queue (Base Class)	Illuminate\Queue\Queue	
Redirect	Illuminate\Routing\Redirector	`redirect`
Redis	Illuminate\Redis\Database	`redis`
Request	Illuminate\Http\Request	`request`
Response	Illuminate\Support\Facades\Response	

Route	Illuminate\Routing\Router	`router`
Schema	Illuminate\Database\Schema\Blueprint	
Session	Illuminate\Session\SessionManager	`session`
Session (Instance)	Illuminate\Session\Store	
SSH	Illuminate\Remote\RemoteManager	`remote`
SSH (Instance)	Illuminate\Remote\Connection	
URL	Illuminate\Routing\UrlGenerator	`url`
Validator	Illuminate\Validation\Factory	`validator`
Validator (Instance)	Illuminate\Validation\Validator	
View	Illuminate\View\Factory	`view`
View (Instance)	Illuminate\View\View	

译者:mpandar(马胜盼)

表单与 HTML

开启表单

开启表单

```
{{ Form::open(array('url' => 'foo/bar')) }}
    //
{{ Form::close() }}
```

默认表单使用 POST 方法,当然您也可以指定传参其他表单的方法:

```
echo Form::open(array('url' => 'foo/bar', 'method' => 'put'))
```

> 注意:因为 HTML 表单只支持 `POST` 和 `GET` 方法,所以在使用 `PUT` 及 `DELETE` 的方法时,Laravel 将会自动加入隐藏的 `_method` 字段到表单中,来伪装表单传送的方法。

您也可以建立指向命名的路由或控制器至表单:

```
echo Form::open(array('route' => 'route.name'))

echo Form::open(array('action' => 'Controller@method'))
```

您也可以传递路由参数:

```
echo Form::open(array('route' => array('route.name', $user->id)))

echo Form::open(array('action' => array('Controller@method', $user->id)))
```

如果您的表单允许上传文件，可以加入 `files` 选项到参数中：

```
echo Form::open(array('url' => 'foo/bar', 'files' => true))
```

CSRF 保护

添加 **CSRF Token** 到表单

Laravel 提供了一个简易的方法，让您可以保护您的应用程序不受到 CSRF (跨网站请求伪造) 攻击。首先 Laravel 会自动在用户的 session 中放置随机的 token，别担心这些会自动完成。如果你为 `POST`、`PUT` 或 `DELETE` 方法调用了 `Form::open` 方法，这个 CSRF 参数会用隐藏字段的方式自动加到您的表单中。另外，您也可以使用 `token` 方法去产生这个隐藏的 CSRF 字段的 HTML 标签：

```
echo Form::token();
```

附加 **CSRF** 过滤器到路由

```
Route::post('profile', array('before' => 'csrf', function() `POST`, `PUT` or `DELETE`
{
    //
}));
```

表单模型绑定

开启模型表单

您经常会想要将模型内容加入至表单中，您可以使用 `Form::model` 方法实现：

```
echo Form::model($user, array('route' => array('user.update', $user->id)))
```

现在当您产生表单元素时，如 text 字段，模型的值将会自动比对到字段名称，并设定此字段值，举例来说，用户模型的 `email` 属性，将会设定到名称为 `email` 的 text 字段的字段值，不仅如此，当 Session 中有与字段名称相符的名称，Session 的值将会优先于模型的值，而优先顺序如下：

1. Session 的数据 (旧的输入值)
2. 明确传递的数据
3. 模型属性数据

这样可以允许您快速地建立表单，不仅是绑定模型数据，也可以在服务器端数据验证错误时，轻松的回填用户输入的旧数据！

> 注意：当使用 `Form::model` 方法时，必须确保有使用 `Form::close` 方法来关闭表单！

标签（Label）

产生标签（Label）元素

```
echo Form::label('email', 'E-Mail Address');
```

指定额外的 HTML 属性

```
echo Form::label('email', 'E-Mail Address', array('class' => 'awesome'));
```

> 注意：在建立标签时，任何您建立的表单元素名称与标签相符时，将会自动在 ID 属性建立与标签名称相同的 ID。

文字字段、多行文字字段、密码字段,、隐藏字段

产生文字字段

```
echo Form::text('username');
```

指定默认值

```
echo Form::text('email', 'example@gmail.com');
```

> 注意：*hidden* 和 *textarea* 方法和 *text* 方法使用属性参数是相同的。

产生密码输入字段

```
echo Form::password('password');
```

产生其他输入字段

```
echo Form::email($name, $value = null, $attributes = array());
echo Form::file($name, $attributes = array());
```

复选框和单选项钮

产生复选框或单选按钮

```
echo Form::checkbox('name', 'value');

echo Form::radio('name', 'value');
```

产生已选取的复选框或单选按钮

```
echo Form::checkbox('name', 'value', true);

echo Form::radio('name', 'value', true);
```

数字输入

生成一个数字输入域

```
echo Form::number('name', 'value');
```

文件输入

产生文件输入

```
echo Form::file('image');
```

> 注意:表单必须已将 `files` 选项设定为 `true`。

下拉式选单

产生下拉式选单

```
echo Form::select('size', array('L' => 'Large', 'S' => 'Small'));
```

产生选择默认值的下拉式选单

```
echo Form::select('size', array('L' => 'Large', 'S' => 'Small'), 'S');
```

产生群组清单

```
echo Form::select('animal', array(
    'Cats' => array('leopard' => 'Leopard'),
    'Dogs' => array('spaniel' => 'Spaniel'),
));
```

产生范围的下拉式选单

```
echo Form::selectRange('number', 10, 20);
```

产生月份名称的清单

```
echo Form::selectMonth('month');
```

按钮

产生提交按钮

```
echo Form::submit('Click Me!');
```

> 注意：需要产生按钮（Button）元素吗？可以试着使用 *button* 方法去产生按钮（Button）元素，button 方法与 *submit* 使用属性参数是相同的。

自定义宏

注册表单宏

您可以轻松的定义您自己的表单类的辅助方法叫「宏（macros）」，首先只要注册 宏 ，并给预期名称及封闭函数：

```
Form::macro('myField', function()
{
    return '<input type="awesome">';
});
```

现在您可以使用注册的名称调用您的宏：

调用自定义表单宏

```
echo Form::myField();
```

产生 URL

更多产生 URL 的信息，请参阅文件辅助方法。

辅助方法

数组

array_add

如果给定的键不在数组中，`array_add` 函数会把给定的键值对加到数组中。

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

$array = array_add($array, 'key', 'value');
```

array_divide

`array_divide` 函数回传两个数组，一个包含原本数组的键，另一个包含原本数组的值。

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

list($keys, $values) = array_divide($array);
```

array_dot

`array_dot` 函数把多维数组扁平化成一维数组，并用 "逗点" 符号表示深度。

```
$array = array('foo' => array('bar' => 'baz'));

$array = array_dot($array);

// array('foo.bar' => 'baz');
```

array_except

`array_except` 函数从数组移除给定的键值对。

```
$array = array_except($array, array('keys', 'to', 'remove'));
```

array_fetch

`array_fetch` 函数回传包含被选择的巢状元素的扁平化数组。

```
$array = array(
    array('developer' => array('name' => 'Taylor')),
    array('developer' => array('name' => 'Dayle')),
);

$array = array_fetch($array, 'developer.name');

// array('Taylor', 'Dayle');
```

array_first

`array_first` 函数回传数组中第一个通过给定的测试为真的元素。

```
$array = array(100, 200, 300);

$value = array_first($array, function($key, $value)
{
    return $value >= 150;
});
```

也可以传递默认值当作第三个参数：

```
$value = array_first($array, $callback, $default);
```

array_last

`array_last` 函数回传数组中最后一个通过给定的测试为真的元素。

```
$array = array(350, 400, 500, 300, 200, 100);

$value = array_last($array, function($key, $value)
{
    return $value > 350;
});

// 500
```

也可以传递默认值当作第三个参数：

```
$value = array_last($array, $callback, $default);
```

array_flatten

`array_flatten` 函数将会把多维数组扁平化成一维。

```
$array = array('name' => 'Joe', 'languages' => array('PHP', 'Ruby'));

$array = array_flatten($array);

// array('Joe', 'PHP', 'Ruby');
```

array_forget

`array_forget` 函数将会用 "逗点" 符号从深度巢状数组移除给定的键值对。

```
$array = array('names' => array('joe' => array('programmer')));

array_forget($array, 'names.joe');
```

array_get

`array_get` 函数将会使用 "逗点" 符号从深度巢状数组取回给定的值。

```
$array = array('names' => array('joe' => array('programmer')));

$value = array_get($array, 'names.joe');

$value = array_get($array, 'names.john', 'default');
```

> 备注: 想要把 `array_get` 用在对象上？请使用 `object_get`。

array_only

`array_only` 函数将会只从数组回传给定的键值对。

```
$array = array('name' => 'Joe', 'age' => 27, 'votes' => 1);

$array = array_only($array, array('name', 'votes'));
```

array_pluck

`array_pluck` 函数将会从数组拉出给定键值对的清单。

```
$array = array(array('name' => 'Taylor'), array('name' => 'Dayle'));

$array = array_pluck($array, 'name');

// array('Taylor', 'Dayle');
```

array_pull

`array_pull` 函数将会从数组回传给定的键值对，并移除它。

```
$array = array('name' => 'Taylor', 'age' => 27);

$name = array_pull($array, 'name');
```

array_set

`array_set` 函数将会使用 "逗点" 符号在深度巢状数组中指定值。

```
$array = array('names' => array('programmer' => 'Joe'));

array_set($array, 'names.editor', 'Taylor');
```

array_sort

`array_sort` 函数通过给定闭包的结果来排序数组。

```
$array = array(
    array('name' => 'Jill'),
    array('name' => 'Barry'),
);

$array = array_values(array_sort($array, function($value)
{
    return $value['name'];
}));
```

array_where

使用给定的闭包过滤数组。

```
$array = array(100, '200', 300, '400', 500);

$array = array_where($array, function($key, $value)
{
    return is_string($value);
});

// Array ( [1] => 200 [3] => 400 )
```

head

回传数组中第一个元素。对 PHP 5.3.x 的方法链很有用。

```
$first = head($this->returnsArray('foo'));
```

last

回传数组中最后一个元素。对方法链很有用。

```
$last = last($this->returnsArray('foo'));
```

路径

app_path

取得 `app` 文件夹的完整路径。

```
$path = app_path();
```

base_path

取得应用程序安装根目录的完整路径。

public_path

取得 `public` 文件夹的完整路径。

storage_path

取得 `app/storage` 文件夹的完整路径。

字串

camel_case

把给定的字串转换成 `驼峰式命名`。

```
$camel = camel_case('foo_bar');

// fooBar
```

class_basename

取得给定类的类名称，不含任何命名空间的名称。

```
$class = class_basename('Foo\Bar\Baz');

// Baz
```

e

对给定字串执行 `htmlentities`，并支持 UTF-8。

```
$entities = e('<html>foo</html>');
```

ends_with

判断句子结尾是否有给定的字串。

```
$value = ends_with('This is my name', 'name');
```

snake_case

把给定的字串转换成 `蛇形命名`。

```
$snake = snake_case('fooBar');

// foo_bar
```

str_limit

限制字串的字串数量。

```
str_limit($value, $limit = 100, $end = '...')
```

例子：

```
$value = str_limit('The PHP framework for web artisans.', 7);

// The PHP...
```

starts_with

判断句子是否开头有给定的字串。

```
$value = starts_with('This is my name', 'This');
```

str_contains

判断句子是否有给定的字串。

```
$value = str_contains('This is my name', 'my');
```

str_finish

加一个给定字串到句子结尾。多余一个的给定字串则移除。

```
$string = str_finish('this/string', '/');

// this/string/
```

str_is

判断字串是否符合给定的模式。星号可以用来当作通配符。

```
$value = str_is('foo*', 'foobar');
```

str_plural

把字串转换成它的多数形态 (只有英文)。

```
$plural = str_plural('car');
```

str_random

产生给定长度的随机字串。

```
$string = str_random(40);
```

str_singular

把字串转换成它的单数形态 (只有英文)。

```
$singular = str_singular('cars');
```

studly_case

把给定字串转换成 `首字大写命名`。

```
$value = studly_case('foo_bar');

// FooBar
```

trans

翻译给定的语句。等同 `Lang::get`。

```
$value = trans('validation.required'):
```

trans_choice

随着词形变化翻译给定的语句。等同 `Lang::choice`。

```
$value = trans_choice('foo.bar', $count);
```

网址

action

产生给定控制器行为的网址。

```
$url = action('HomeController@getIndex', $params);
```

route

产生给定路由名称的网址。

```
$url = route('routeName', $params);
```

asset

产生资产的网址。

```
$url = asset('img/photo.jpg');
```

link_to

产生给定网址的 HTML 链接。

```
echo link_to('foo/bar', $title, $attributes = array(), $secure = null);
```

link_to_asset

产生给定资产的 HTML 链接。

```
echo link_to_asset('foo/bar.zip', $title, $attributes = array(), $secure = null);
```

link_to_route

产生给定路由的 HTML 链接。

```
echo link_to_route('route.name', $title, $parameters = array(), $attributes = array());
```

link_to_action

产生给定控制器行为的 HTML 链接。

```
echo link_to_action('HomeController@getIndex', $title, $parameters = array(), $attributes =
```

secure_asset

产生给定资产的 HTTPS HTML 链接。

```
echo secure_asset('foo/bar.zip', $title, $attributes = array());
```

secure_url

产生给定路径的 HTTPS 完整网址。

```
echo secure_url('foo/bar', $parameters = array());
```

url

产生给定路径的完整网址。

```
echo url('foo/bar', $parameters = array(), $secure = null);
```

其他

csrf_token

取得现在 CSRF token 的值。

```
$token = csrf_token();
```

dd

打印给定变量并结束脚本执行。

```
dd($value);
```

value

如果给定的值是个 `闭包`，回传 `闭包` 的回传值。不是的话，则回传值。

```
$value = value(function() { return 'bar'; });
```

with

回传给定对象。对 PHP 5.3.x 的建构式方法链很有用。

```
$value = with(new Foo)->doWork();
```

IoC 容器

Introduction

Laravel使用IoC（Inversion of Control，控制倒转，这是一个设计模式，可以先查看下百科）容器这个强有力的工具管理类依赖。依赖注入（也是一种设计模式，一般用于实现IoC）是一个不用编写固定代码来处理类之间依赖的方法，相反的，这些依赖是在运行时注入的，这样允许处理依赖时具有更大的灵活性。

理解 Laravel IoC容器是构建强大应用程序所必要的，也有助于Laravel 核心本身。

基本用例

绑定一个类型到容器

IoC 容器有两种方法来解决依赖关系：通过闭包回调或者自动解析。首先，我们来探究一下闭包回调。首先，需要绑定一个“类型”到容器中：

```
App::bind('foo', function($app)
{
    return new FooBar;
});
```

从容器中取得一个类型

```
$value = App::make('foo');
```

当执行 `App::make` 方法，闭包函数被执行并返回结果。

绑定一个”共享“类型到容器

有时，你只想将绑定到容器的类型处理一次，然后接下来从容器中取得的都应该是相同实例：

```
App::singleton('foo', function()
{
```

```
    return new FooBar;
});
```

绑定一个已经存在的类型实例到容器

你也可以使用`instance`方法，将一个已经存在的对象接口绑定到容器中：

```
$foo = new Foo;

App::instance('foo', $foo);
```

哪里去注册绑定呢

IoC绑定，很像事件句柄或者路由过滤，通常在"bootstrap code(引导代码)"之后完成。换句话说，它们在你的应用程序准备处理请求，也即是在一个路由或者控制器被实际执行之前执行。和其他引导代码一样，`start`文件通常作为IoC绑定注册一种方法。另外，你可以创建一个`app/ioc.php`（文件名不一定一样）文件，并在`start`文件中包含它。

如果你的应用程序有很大量IoC绑定，或者你想根据不同的分类将IoC绑定分割到不同的文件，你可以尝试在服务提供器中进行绑定

自动解析

取得一个类

IoC容器足够强大，在许多场景下不需要任何配置就能取得类。例如

```
class FooBar {

    public function __construct(Baz $baz)
    {
        $this->baz = $baz;
    }

}

$fooBar = App::make('FooBar');
```

注意我们虽然没有在容器中注册`FooBar`类，容器仍然可以取得该类，甚至自动注入`Baz`依赖！

当某个类型没有绑定到容器，IoC容器将使用 PHP 的反射工具来检查类和读取构造器的类型提示。使用这些信息，容器可以自动构建类实例。

绑定一个接口实现

然而，在某些情况下，一个类可能依赖某个接口实现，而不是一个 “具体的类”。当在这种情况

下，`App::bind`方法必须通知容器注入哪个接口实现：

```
App::bind('UserRepositoryInterface', 'DbUserRepository');
```

现在考虑下这个控制器：

```
class UserController extends BaseController {

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

}
```

由于我们将 `UserRepositoryInterface` 绑定了具体类，`DbUserRepository` 在该控制器创建时将会被自动注入到该控制器。

实际用例

Laravel 提供了几个方法使用 IoC 容器增强应用程序可扩展性和可测试性。一个主要的例子是取得控制器。所有控制器都通过 IoC 容器取得，意味着可以在控制器构造方法中对依赖的类型提示，它们将自动被注入。

对控制器的依赖关系做类型提示

```
class OrderController extends BaseController {

    public function __construct(OrderRepository $orders)
    {
        $this->orders = $orders;
    }

    public function getIndex()
    {
        $all = $this->orders->all();

        return View::make('orders', compact('all'));
    }

}
```

在这个例子中，`OrderRepository` 将会自动注入到控制器。意味着当 单元测试 模拟请求时，`OrderRepository` 将会绑定到容器以及注入到控制器中，允许无痛与数据库层交互。

IoC 使用的其他例子

过滤器, composers, 和 事件句柄也能够从IoC容器中获取到。当注册它们的时候，只需要把它们

使用的类名简单给出即可：

```
Route::filter('foo', 'FooFilter');

View::composer('foo', 'FooComposer');

Event::listen('foo', 'FooHandler');
```

服务提供器

服务器提供器是将一组相关 IoC 注册到单一路径的有效方法。将它们看做是一种引导组件的方法。在服务器提供器里，你可以注册自定义的验证驱动器，使用 IoC 容器注册应用程序仓库类，甚至是自定义 `Artisan` 命令。

事实上，大多数核心 Laravel 组件包含服务提供器。应用程序所有注册在服务提供器的均列在 `app/config/app.php` 配置文件的 `providers` 数组中。

定义服务提供器

要创建服务提供器，只需继承 `Illuminate\Support\ServiceProvider` 类并且定义一个 `register` 方法：

```
use Illuminate\Support\ServiceProvider;

class FooServiceProvider extends ServiceProvider {

    public function register()
    {
        $this->app->bind('foo', function()
        {
            return new Foo;
        });
    }

}
```

注意在 `register` 方法，应用程序通过 `$this->app` 属性访问 IoC 容器。一旦你已经创建了提供器并且想将它注册到应用程序中，只需简单的放入 app 配置文件里 `providers` 数组中。

运行时注册服务提供器

你也可以使用 `App::register` 方法在运行时注册服务提供器：

```
App::register('FooServiceProvider');
```

容器事件

注册获取事件监听者

容器在每次获取对象时都触发一个事件。你可以通过使用 `resolving` 方法来监听该事件：

```
App::resolvingAny(function($object)
{
    //
});

App::resolving('foo', function($foo)
{
    //
});
```

注意获取到的对象将会传入回调函数中。

译者：mpandar（马胜盼）

本地化

介绍

Laravel 的 `Lang` 类提供方便的方法来取得多种语言的字串，让您简单地在应用程序里支持多种语言。

语言文件

语言字串储存在 `app/lang` 文件夹的文件里。在这个文件夹里应该要有子文件夹给每一个应用程序支持的语言。

```
/app
    /lang
        /en
            messages.php
        /es
            messages.php
```

语言文件例子

语言文件简单地返回键跟字串的数组。例如：

```
<?php

return array(
    'welcome' => 'Welcome to our application'
);
```

在执行时变换默认语言

应用程序的默认语言被储存在 `app/config/app.php` 配置文件。您可以在任何时候用 `App::setLocale` 方法变换现行语言：

```
App::setLocale('es');
```

设定备用语言

您也可以设定 "备用语言", 它将会在当现行语言没有给定的语句时被使用。就像默认语言, 备用语言也可以在 `app/config/app.php` 配置文件设定:

```
'fallback_locale' => 'en',
```

基本用法

从语言文件取得句子

```
echo Lang::get('messages.welcome');
```

传递给 `get` 方法的字串的第一个部分是语言文件的名称, 第二个部分是应该被取得的句子的名称。

> 备注: 如果句子不存在, `get` 方法将会返回键的名称。

您也可以使用 `trans` 辅助方法, 它是 `Lang::get` 方法的别名。

```
echo trans('messages.welcome');
```

在句子中做替代

您也可以在语言文件中定义占位符:

```
'welcome' => 'Welcome, :name',
```

接着, 传递替代用的第二个参数给 `Lang::get` 方法:

```
echo Lang::get('messages.welcome', array('name' => 'Dayle'));
```

判断语言文件是否有句子

```
if (Lang::has('messages.welcome'))
{
    //
}
```

复数

复数是个复杂的问题, 不同语言对于复数有很多种复杂的规则。您可以简单地在您的语言文件里管理它。您可以用 "管道" 字串区分字串的单数和复数形态:

```
'apples' => 'There is one apple|There are many apples',
```

接着您可以用 `Lang::choice` 方法取得语句：

```
echo Lang::choice('messages.apples', 10);
```

您也可以提供一个地区参数来指定语言。举个例，如果您想要使用俄语 (ru)：

```
echo Lang::choice('товар|товара|товаров', $count, array(), 'ru');
```

因为 Laravel 的翻译器继承了 Symfony 翻译组件，您也可以很容易地建立更明确的复数规则：

```
'apples' => '{0} There are none|[1,19] There are some|[20,Inf] There are many',
```

验证

要验证本地化的错误和信息，可以看一下 验证的文件.

重写扩展包的语言文件

许多扩展包附带它们自有的语句。您可以通过放置文件在 `app/lang/packages/{locale}/{package}` 文件夹重写它们，而不是改变扩展包的核心文件来调整这些句子。所以，举个例子，如果您需要重写 `skyrim/hearthfire` 扩展包在 `messages.php` 的英文语句，您可以放置语言文件在：`app/lang/packages/en/hearthfire/messages.php`。您可以只定义您想要重写的语句在这个文件里，任何您没有重写的语句将会仍从扩展包的语言文件载入。

邮件发送

设定

Laravel 利用一个热门的函数库 SwiftMailer 来建立一个干净简单的 API。邮件配置文件为 `app/config/mail.php`，里面有些选项可以让您更改您的 SMTP 连接地址、凭证以及全局的寄件者 `from`。您可以使用任何的 SMTP 服务器。如果您想要使用 PHP 内建的 `mail` 函数来发送邮件，您可以将配置文件中的 `driver` 改为 `mail` 即可。`sendmail` 的驱动一样可以使用。

第三方邮件服务 API 驱动

Laravel 也包含了 Mailgun 和 Mandrill 服务的 HTTP API 的驱动方式。这些 API 比使用 SMTP 服务器更简单快速。这些驱动都需要在您的应用程序里预先安装 Guzzle 4 HTTP 函数库。您可以通过在您的 `composer.json` 文件中增加一行如下，来将 Guzzle 4 安装到您的项目中：

```
"guzzlehttp/guzzle": "~4.0"
```

Mailgun 驱动

使用 Mailgun 驱动前，先在 `app/config/mail.php` 配置文件中将 `driver` 设定为 `mailgun`。再来，如果您的项目中没有 `app/config/services.php` 请先建立他。并确定它包含了如下的内容：

```
'mailgun' => array(
    'domain' => 'your-mailgun-domain',
    'secret' => 'your-mailgun-key',
),
```

Log 驱动

如果您的 `app/config/mail.php` 配置文件的 `driver` 被设定为 `log` 时，所有的邮件将会被写进日志中，且并不会真的被寄出。这主要用在快速本地除错和内容验证上。

基本用法

`Mail::send` 方法是用来发送电子邮件：

```
Mail::send('emails.welcome', array('key' => 'value'), function($message)
```

```
{
    $message->to('foo@example.com', 'John Smith')->subject('Welcome!');
});
```

`send` 方法的第一个参数为邮件内容的视图名称。第二个参数 `$data` 为传递至视图的变量，一般是以联合数组的形式出现，所有数据项都可以在视图中通过 `$key` 引用。而第三个参数为一个闭包让您可以指定更多发送电子邮件的信息选项。

> 注意：变量默认被传递进邮件的视图中，且允许内嵌键值数组。所以最好避免传递名为 `message` 的变量到您的视图中造成冲突。

除了使用 HTML 视图外，您也可以指定一个纯文字的视图：

```
Mail::send(array('html.view', 'text.view'), $data, $callback);
```

或者您也可以指定视图类为 `html` 或 `text`：

```
Mail::send(array('text' => 'view'), $data, $callback);
```

您也可以为电子邮件信息指定其他选项如附加文件或是任何的副本收件者：

```
Mail::send('emails.welcome', $data, function($message)
{
    $message->from('us@example.com', 'Laravel');

    $message->to('foo@example.com')->cc('bar@example.com');

    $message->attach($pathToFile);
});
```

当附加文件到一个电子邮件中时，您也可以指定一个 MIME type 及/或 一个显示名称：

```
$message->attach($pathToFile, array('as' => $display, 'mime' => $mime));
```

> 注意：在 `Mail::send` 的闭包中使用 SwiftMailer 的 message 扩充实例，允许您可以调用任何类中的方法来建立您的电子邮件。

内嵌附件

内嵌图片到邮件中通常是一件很麻烦的事，然而Laravel 提供一个便利的方式让您内嵌图片到您的电子邮件当中且接收相对应的 CID。

内嵌图片到电子邮件的视图中

```
<body>
    Here is an image:
```

```
    <img src="<?php echo $message->embed($pathToFile); ?>">
</body>
```

内嵌数据到电子邮件的视图中

```
<body>
    Here is an image from raw data:

    <img src="<?php echo $message->embedData($data, $name); ?>">
</body>
```

注意 `$message` 这个变量一定会被 `Mail` 类传递到电子邮件的视图当中。

邮件队列

加入一个邮件到队列中

发送电子邮件会大幅的延长您应用程序的反应时间，许多开发者选择让电子邮件放进队列中，并在后台(非即时)发送。Laravel 使用内建的 unified queue API 让您可以方便的将要发送的电子邮件加入到队列之中，只要使用 `Mail` 类的 `queue` 方法：

```
Mail::queue('emails.welcome', $data, function($message)
{
    $message->to('foo@example.com', 'John Smith')->subject('Welcome!');
});
```

您也可以通过使用 `later` 方法来指定发送电子邮件的延迟秒数：

```
Mail::later(5, 'emails.welcome', $data, function($message)
{
    $message->to('foo@example.com', 'John Smith')->subject('Welcome!');
});
```

假如您想要指定一个特别的队列或管道来发送邮件，您可以使用 `queueOn` 和 `laterOn` 方法：

```
Mail::queueOn('queue-name', 'emails.welcome', $data, function($message)
{
    $message->to('foo@example.com', 'John Smith')->subject('Welcome!');
});
```

邮件与本地开发

在开发应用程序的寄信功能时，通常会希望在本地或是开发环境中关闭发送功能。您可以调用 `Mail::pretend` 方法，或是在 `app/config/mail.php` 配置文件的 `pretend` 为 `true` 可以达到。当在 `pretend` 模式开启下，邮件将会被写到您的应用程序日志中取代实际寄出信件。

开启 **"Pretend"** 发送模式

```
Mail::pretend();
```

Package Development

简介

包是向Laravel中添加功能最重要的途径。它通过一种强大的方式几乎可以包含任意功能，比如处理日期的扩展包Carbon，完整的BDD测试框架扩展包Behat.

当然，还有很多不同类型的包。有些包是独立的，这意味着它们可以在任何框架中工作，而不仅仅是Laravel。上面提到的Carbon和Behat就是独立的包。要在Laravel中使用这些包只需要在`composer.json`文件中指明。

另一方面，有些包仅支持Laravel。在上一个Laravel版本中，这些类型的包我们称为"bundles"。这些包可以增强Laravel应用的路由、控制器、视图、配置和迁移。由于开发独立的包不需要专门的过程，因此，本手册主要涵盖针对Laravel开发独立的包。

所有Laravel包都是通过Packagist和Composer发布的，因此很有必要学习这些功能强大的PHP包发布管理工具。

创建包

为Laravel创建一个包的最简单方式是使用Artisan的`workbench`命令。首先，你需要在`app/confg/workbench.php`文件中配置一些参数。在该文件中，你会看到`name`和`email`两个参数，这些值是用来为您新创建的包生成`composer.json`文件的。一旦你提供了这些值，就可以开始构建一个新包了！

执行**Artisan**的**workbench**命令

```
php artisan workbench vendor/package --resources
```

厂商名称(vendor name)是用来区分不同作者构建了相同名字的包。例如,如果我(Taylor Otwell)创建了个名为"Zapper"的包,厂商名就可以叫做`Taylor`,包名可以叫做`Zapper`。默认情况下,workbench命令建的包是不依赖任何框架的;不过,`resources`命令将会告诉workbench生成关联Laravel的一些特殊目录,例如migrations、views、config等。

一旦执行了`workbench`命令,新创建的包就会出现在Laravel安装跟目录下的`workbench`目录中。接下来就应该为你创建的包注册`ServiceProvider`了。你可以通过在`app/config/app.php`文件里的`provides`数组中添加该包。这将通知Laravel在应用程序开始启动时加载该包。服务提供者(Service providers)使用`[Package]ServiceProvider`样式的命名方式。所以,以上案例中,你需要将`Taylor\Zapper\ZapperServiceProvider`添加到providers数组。

一旦注册了provider,你就可以开始写代码了!然而,在此之前,建议你查看以下部分来了解更多关于包结构和开发流程的知识。

> 注意: 如果你的Service providers提示无法找到, 在项目根目录执行`php artisan dump-autoload`

包结构

执行`workbench`命令之后,你的包结构将被初始化,并能够与laravel框架完美融合:

包的基本目录结构

```
/src
    /Vendor
        /Package
            PackageServiceProvider.php
    /config
    /lang
    /migrations
    /views
/tests
/public
```

让我们来深入了解该结构。`src/Vendor/Package`目录是所有`class`的主目录,包括`ServiceProvider`。`config`、`lang`、`migrations`和`views`目录,就如你所猜测,包含了你创建的包的相应资源。包可以包含这些资源中的任意几个,就像一个"常规"的应用。

服务提供器

服务提供器只是包的引导类。默认情况下，他们包含两个方法：`boot`和`register`。你可以在这些方法内做任何事情，例如：包含路由文件、注册IoC容器的绑定、监听事件或者其他任何你想做的事情。

`register`方法在服务提供器注册时被立即调用，而`boot`方法仅在请求被路由到之前调用。因此，如果服务提供器中的动作（action）依赖另一个已经注册的服务提供器，或者你正在覆盖另一个服务提供其绑定的服务，就应该使用`boot`方法。

当使用`workbench`命令创建包时，`boot`方法已经包含了如下的动作：

```
$this->package('vendor/package');
```

该方法告诉Laravel如何为应用程序加载视图、配置或其他资源。通常情况下，你没有必要改变这行代码，因为它会根据workbench的默认约定将包设置好的。

默认情况下，一旦注册了一个包，那么它的资源可以通过"package"方法在`vendor/package`中找到。你也可以向`package` 方法中传入第二个参数来重写这个方法。例如

```
//向 `package` 方法中传入一个自定义的命名空间
$this->package('vendor/package', 'custom-namespace');

//现在，这个包的资源现在可以通过这个自定义的命名空间来访问
$view = View::make('custom-namespace::foo');
```

Laravel并没有为service provider提供“默认”的存放地点。您可以根据自己的喜好，将它们放置在任何地方，您也可以将它们统一组织在一个`Providers`命名空间里，并放置在应用的`app`目录下。这些文件可以被放置在任何地方，只需保证Composer的自动加载组件知道如何加载这些类。

如果你改变了你的包得资源的位置，比如配置文件或者视图，你需要在`package`函数中传递第三个参数，指定你的资源位置

```
$this->package('vendor/package', null, '/path/to/resources');
```

延迟加载服务提供器

如果你写了一个服务提供器，但没有注册任何资源，比如配置文件或视图等，你可以选择"延迟"加载你的提供器。延迟加载的服务提供器在这个服务真正被IoC容器需要的时候才会被加载和注册。如果这个服务提供器没有被当前的请求路由需要，那么这个提供器永远不会被加载

如果想延时载入你的服务提供器，只需要在提供器重设置`defer`属性为`true`：

```
protected $defer = true;
```

接下来你需要重写继承自基类`Illuminate\Support\ServiceProvider`的`provides`方法，并返回绑定了

你的提供器，对应的IoC容器中类型的数组集合。例如，你的提供器在IoC容器注册了`package.service` 和 `package.another-service`两个类型，你的`provides`的方法看起来应该是这个样子：

```
public function provides()
{
    return array('package.service', 'package.another-service');
}
```

包约定

要使用包中的资源，例如配置或视图，需要用双冒号语法：

从包中载入视图

```
return View::make('package::view.name');
```

获取包的某个配置项

```
return Config::get('package::group.option');
```

> 注意：如果你包中包含迁移，请为迁移名（migration name）添加包名作为前缀，以避免与其他包中的类名冲突。

开发流程

当开发一个包时，能够使用应用程序上文是相当有用的，这样将允许你很容易的解决视图模板的等问题。所以，我们开始，安装一个全新的Laravel框架，使用`workbench`命令创建包结构。

在使用`workbench`命令创建包后。你可以在`workbench/[vendor]/[package]`目录使用`git init`，并在workbench中直接`git push`！这将允许你在应用程序上下文中方便开发而不用为反复使用`composer update`命令苦扰。

当包存放在`workbench`目录时，你可能担心Composer如何知道自动加载包文件。当workbench目录存在，Laravel将智能扫描该目录，在应用程序开始时加载它们的Composer自动加载文件！

如果你需要重新生成包的自动加载文件，你可以使用`php artisan dump-autoload`命令，这个命令将会重新为您的整个项目生成自动加载文件，也包括你的工作台(workbenches)中的包

运行**Artisan**的自动加载命令

```
php artisan dump-autoload
```

包路由

在之前的Laravel版本中，`handlers`用来指定哪个URI包会响应。然而，在Laravel4中，一个包可以响应任意URI。要在包中加载路由文件，只需在服务提供器的`boot`方法`include`它。

在服务提供器中包含路由文件

```
public function boot()
{
    $this->package('vendor/package');

    include __DIR__.'/../../routes.php';
}
```

> 注意: 如果你的包中使用了控制器，你需要确保正确配置了`composer.json`文件的auto-load字段.

包配置

访问包配置文件

有时创建的包可能会需要配置文件。这些配置文件应该和应用程序配置文件相同方法定义。并且，当使用 `$this->package`方法来注册服务提供器时，那么就可以使用“双冒号”语法来访问：

```
Config::get('package::file.option');
```

访问包单一配置文件

然而，如果你包仅有一个配置文件，你可以简单命名为`config.php`。当你这么做时，你可以直接访问该配置项，而不需要特别指明文件名：

```
Config::get('package::option');
```

Registering A Resource Namespace Manually

手动注册资源的命名空间

有时候，你可能希望在`$this->package`特有方法之外注册包的资源，比如视图。通常只有在这些资源不在约定的位置的时候才应该这么做。如果需要手动注册资源，你可以使用`View`, `Lang`, 和 `Config` 中的`addNamespace`方法实现：

```
View::addNamespace('package', __DIR__.'/path/to/views');
```

一旦这个资源命名空间被注册，你就可以使用这个空间名，并使用"双冒号"语法去访问这个资源

```
return View::make('package::view.name');
```

`View`, `Lang`, 和 `Config`类中的`addNamespace`方法的参数都是一样的

级联配置文件

但其他开发者安装你的包时，他们也许需要覆盖一些配置项。然而，如果从包源代码中改变值，他们将会在下次使用Composer更新包时又被覆盖。替代方法是使用artisan命令`config:publish`：

```
php artisan config:publish vendor/package
```

当执行该命令，配置文件就会拷贝到`app/config/packages/vendor/package`，开发者就可以安全的更改配置项了。

> 注意：开发者也可以为该包创建指定环境下的配置文件，替换某些配置项然后并放置在`app/config/packages/vendor/package/environment`.

包视图

如果在你的应用程序中使用某个包，你可能会定制这个包的视图。你可以很轻松的将这个包的视图文件导入到你自己的`app/views`目录，只需使用`view:publish`这个Artisan命令

```
php artisan view:publish vendor/package
```

这个命令会将包的视图文件迁移到`app/views/packages`目录，如果这个目录不存在，将会被创建。一旦这些视图文件被公开创建，你就可以按照你自己的喜好去修改他们，这些导出的视图会比包自己的视图文件优先载入。

包迁移

为工作台**(Workbench)**的包创建迁移

你可以很容易在包中创建和运行迁移。要为工作台里的包创建迁移，使用`--bench`选项：

```
php artisan migrate:make create_users_table --bench="vendor/package"
```

为工作台**(Workbench)**的包运行迁移

```
php artisan migrate --bench="vendor/package"
```

为已安装的包执行迁移

要为已经通过Composer安装在vendor目录下的包执行迁移，你可以直接使用`--package`：

```
php artisan migrate --package="vendor/package"
```

包Assets

将包Assets移动到Public

有些包可能含有assets，例如JavaScript，CSS，和图片。然而，我们无法链接到`vendor`或`workbench`目录里的`assets`，所以我们需要可以将这些`assets`移入应用程序的`public`目录。`asset:publish`命令可以实现：

```
php artisan asset:publish

php artisan asset:publish vendor/package
```

如果这个包仍在`workbench`中，那么请使用`--bench`指令:

```
php artisan asset:publish --bench="vendor/package"
```

这个命令将会把assets移入与“供应商”和“包名”相对应的`public/packages`目录下面。因此，包名为`userscape/kudos`的assets将会被移至`public/packages/userscape/kudos`。通过使用这个assets发布方法，可以让您安全的在包中的view内访问assets路径。

发布包

当你创建的包准备发布时，你应该将包提交到 Packagist 仓库。如果你的包只针对Laravel，最好在包的`composer.json`文件中添加`laravel`标签

还有，在发布的版本中添加tag，以便开发者能当请求你的包在他们`composer.json`文件中依赖稳定版本。如果稳定版本还没有好，考虑直接在Composer中使用`branch-alias`。

一旦你的包发布，放舒心，继续在由`workbench`创建的包中，结合应用程序上下文进行开发。对于在发布包后继续进行开发，这是相当便利的。

一些组织使用他们私有分支包为他们自己开发者。如果你对这感兴趣，查看Composer团队构建的 Satis 文档。

译者：mpandar（马胜盼）

分页

设置

在其他的框架中，实现分页是令人感到苦恼的事，但是 Laravel 能让它实现得很轻松。在 `app/config/view.php` 文件中有设置选项可以设定相关参数，`pagination` 选项需要指定用哪个视图来建立分页，而 Laravel 默认包含两种视图。

`pagination::slider` 视图将会基于现在的页面智能的显示「范围」的页数链接，`pagination::simple` 视图将仅显示「上一页」和「下一页」的按钮。两种视图都兼容 **Twitter Bootstrap** 框架

使用

有几种方法可用来操作分页数据。最简单的是在查询构造器或 Eloquent 模型使用 `paginate` 方法。

对数据库结果分页

```
$users = DB::table('users')->paginate(15);
```

对 **Eloquent** 模型分页

您也可以对 Eloquent 模型分页：

```
$allUsers = User::paginate(15);

$someUsers = User::where('votes', '>', 100)->paginate(15);
```

传送给 `paginate` 方法的参数是您希望每页要显示的数据条数，只要您取得查询结果后，您可以在视图中显示，并使用 `links` 方法去建立分页链接：

```
<div class="container">
    <?php foreach ($users as $user): ?>
        <?php echo $user->name; ?>
    <?php endforeach; ?>
</div>

<?php echo $users->links(); ?>
```

这就是所有建立分页系统的步骤了！您会注意到我们还没有告知 Laravel 我们目前的页面是哪一页，放心，这个信息 Laravel 会自动帮您做好。

如果您想要指定自定义的视图来使用分页，您可以使用 `links` 方法：

```
<?php echo $users->links('view.name'); ?>
```

您也可以通过以下方法获得额外的分页信息：

- `getCurrentPage`
- `getLastPage`
- `getPerPage`
- `getTotal`
- `getFrom`
- `getTo`
- `count`

「简单分页」

如果您只是要在您的分页视图显示「上一页」和「下一页」链接，您可以使用 `simplePaginate` 方法来执行更高效率的搜寻。当您不需要精准的显示页码在视图上时，且数据集比较大时，这个方法非常有用：

```
$someUsers = User::where('votes', '>', 100)->simplePaginate(15);
```

手动建立分页

有的时候您可能会想要从数组中数据手动建立分页实例。您可以使用 `Paginator::make` 方法：

```
$paginator = Paginator::make($items, $totalItems, $perPage);
```

自定义分页 URL

您还可以通过 `setBaseUrl` 方法自定义使用的 URL：

```
$users = User::paginate();

$users->setBaseUrl('custom/url');
```

上面的例子将建立 URL，类似以下内容：http://example.com/custom/url?page=2

加入分页链接

您可以使用 `appends` 方法增加搜寻字串到分页链接中：

```
<?php echo $users->appends(array('sort' => 'votes'))->links(); ?>
```

这样会产生类似下列的链接：

```
http://example.com/something?page=2&sort=votes
```

如果您想要将「哈希片段 hash」加到分页的 URL中，您可以使用 `fragment` 方法：

```
<?php echo $users->fragment('foo')->links(); ?>
```

此方法调用后将产生 Url，看起来像这样：

```
http://example.com/something?page=2#foo
```

转换至 JSON

`Paginator` 类实现 `Illuminate\Support\Contracts\JsonableInterface` 接口的 `toJson` 公开方法。由路由返回的值，您可以将 `Paginator` 实例传换成 JSON。JSON 表单的实例会包含一些「元」信息，例如 `total`, `current_page`, `last_page`, `from` , `to`。该实例数据将可通过在 JSON 数组中 `data` 的键取得。

自定义表示器（Presenter）

默认的分页表示器是兼容 Bootstarp 的。不过，您也可以自定义表示器（presenter）

扩展抽象表示器（Presenter）

实现 `Illuminate\Pagination\Presenter` 类的抽象方法。Zurb Foundation 的表示器（presenter）例子如下：

```
class ZurbPresenter extends Illuminate\Pagination\Presenter {

    public function getActivePageWrapper($text)
    {
        return '<li class="current"><a href="">'.$text.'</a></li>';
    }

    public function getDisabledTextWrapper($text)
    {
        return '<li class="unavailable"><a href="">'.$text.'</a></li>';
    }

    public function getPageLinkWrapper($url, $page, $rel = null)
    {
        return '<li><a href="'.$url.'">'.$page.'</a></li>';
    }
```

```
}
```

使用自定义表示器（**Presenter**）

首先，在 `app/views` 建立新的视图，这将会作为您的自定义表示器（presenter）。并且用新的视图取代 `app/config/view.php` 的 `pagination` 设定。最后，类似下方的代码会放在您的自定义表示器（presenter）视图中。

```
<ul class="pagination">
    <?php echo with(new ZurbPresenter($paginator))->render(); ?>
</ul>
```

队列

设定

Laravel 队列组件提供了一个统一的队列服务API，队列允许您将一个任务延后执行，例如可以将邮件延后至您指定的时间再发送，进而大幅的加快您开发网站应用程序的效率。

队列的配置文件在 `app/config/queue.php`，在这个文件里您将可以找到框架中每种不同的队列服务的连接配置，其中包含了 Beanstalkd，IronMQ，Amazon SQS，Redis，以及同步(本地端使用)驱动设定。

下列的 composer.json 表示您的队列服务所需要的依赖环境：

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

基本用法

推送一个工作至队列

要推送一个新的工作至队列，请使用 `Queue::push` 方法：

```
Queue::push('SendEmail', array('message' => $message));
```

定义一个工作处理程序

`push` 方法的第一个参数为应该处理这个工作的类方法名称，第二个参数是一个要传递至处理程序的数组，一个工作处理程序应该参照下列定义：

```
class SendEmail {

    public function fire($job, $data)
    {
```

```
        //
    }

}
```

注意如果您未在第一个参数指定工作处理的类及方法(如 SendEmail@process), 那 `fire` 为类中默认必需的方法用来接收被推送至队列 `Job` 实例以及 `data`。

指定一个特定的处理程序方法

假如您想要用一个 `fire` 以外的方法处理工作, 您可以在推送工作时指定方法如下:

```
Queue::push('SendEmail@send', array('message' => $message));
```

指定队列使用特定连接

您也可指定队列工作送至指定的连接:

```
Queue::push('SendEmail@send', array('message' => $message), 'emails');
```

发送相同的数据去多个连接

如果您需要传送一样的数据去几个不同的队列服务器, 您也可以使用 `Queue::bulk` 方法:

```
Queue::bulk(array('SendEmail', 'NotifyUser'), $payload);
```

延迟执行一个工作

有时后您也希望延迟一个队列工作的执行, 举例来说您希望一个队列工作在客户注册 15 分钟后发送一个 e-mail, 您可以使用 `Queue::later` 方法来完成这件事情:

```
$date = Carbon::now()->addMinutes(15);

Queue::later($date, 'SendEmail@send', array('message' => $message));
```

在这个例子中, 我们使用 Carbon 日期函数库来指定我们希望队列工作希望延迟的时间, 另外您也可传送一个整数来设定您希望延迟的秒数。

删除一个处理中的工作

当您已经开始处理完成一个队列工作, 它就必需在队列中删除, 我们可以通过 `Job` 实例中的 `delete` 方法来完成这件事:

```
public function fire($job, $data)
{
    // Process the job...
```

```
    $job->delete();
}
```

释放一个工作回到队列中

假如您希望将一个工作释放回队列之中，您可以通过 `release` 方法来完成这件事情：

```
public function fire($job, $data)
{
    // Process the job...

    $job->release();
}
```

您也可以指定秒数来让这个工作延迟释放：

```
$job->release(5);
```

检查工作执行次数

当一个工作执行后发生错误，这个工作将会自动的释放回队列当中，您可以通过 `attempts` 方法来检查这个工作已经被执行的次数：

```
if ($job->attempts() > 3)
{
    //
}
```

取得一个工作的 **ID**

您也可以取得这个工作的识别码：

```
$job->getJobId();
```

队列闭包

您也可以推送一个闭包去队列，这个方法非常的方便及快速的来处理需要使用队列的简单的任务：

```
Queue::push(function($job) use ($id)
{
    Account::delete($id);

    $job->delete();
});
```

> 注记: 要让一个组件变量可以在队列闭包中可以使用我们会通过 `use` 命令，试着传送主键及

> 重复使用的相关模组在您的队列工作中，这可以避免其他的序列化行为。

当使用 Iron.io push queues 时，您应该在队列闭包中采取一些其他的预防措施，我们应该在执行工作收到队列数据时检查token是否真来自 Iron.io，举例来说您推送一个队列工作到 `https://yourapp.com/queue/receive?token=SecretToken`，接下来在您的工作收到队列的请求时，您就可以检查token的值是否正确。

执行一个队列监听

Laravel 内含一个 Artisan 命令，它将推送到队列的工作拉来下执行，您可以使用 `queue:listen` 命令，来执行这件常驻任务：

开始队列监听

```
php artisan queue:listen
```

您也可以指定特定队列连接让监听器使用：

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

注意当这个任务开始时，这将会一直持续执行到他被手动停止，您也可以使用一个处理监控如 Supervisor 来确保这个队列监听不会停止执行。

您也可以在 `listen` 命令中使用逗号分隔不同的队列连接来设定队列的重要性：

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

在这个范列中 `high-connection` 将总是会优先处理队列的工作，相对于 `low-connection`

指定工作超时参数

您也可以设定给每个工作允许执行的秒数：

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

指定队列休息时间

此外，您也可以指定让监听器在拉取新工作时要等待几秒：

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

注意队列只会在没有队列内容时停止工作，假如队列中仍有许多可执行的工作，队列监听将持续不中断的处理

处理队列上的一个工作

当您只想处理队列上的一个工作您可以使用 `queue:work` 命令：

```
php artisan queue:work
```

常驻队列处理器

`queue:work` 也包含了一个 `--daemon` 选项能强迫队列处理器可以持续处理工作，即使框架重新启动了也不会停止。这种方式比起 `queue:listen` 来说，可以更有效的减少CPU的使用量，不过代价是要增加了您布署时的复杂性。

当开始一个队列处理器处于常驻模式，使用 `--daemon` 标示：

```
php artisan queue:work connection --daemon

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

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

如您所见 `queue:work` 命令支持 `queue:listen` 大多相同的选项参数，您也可使用 `php artisan help queue:work` 命令来观看全部可用的选项参数。

布署常驻队列处理器

最简单的布署一个应用程序使用常驻队列处理器的方式就是将应用程序在开始布署时使用维护模式，您可以使用 `php artisan down` 命令来完成这件事情，当这个应用程序在维护模式，Laravel 将不会运行队列上的任何新工作，但会持续的处理已存在的工作，当过了一段时间足够您所有的正在执行的工作都已处理完以后(通常不会很久，大约 30-60 秒即可)，您可以停止处理器及继续处理您的布署工作。

最简单的队列处理器的办法就是在部署脚本中包含下列命令：

```
php artisan queue:restart
```

此命令将通知所有队列处理器在完成当前任务后重启。

> 注意：此命令依赖缓存系统来调度重启工作。默认情况下，APCu 不能在命令行中工作。如果你正在使用 APCu，将 `apc.enable_cli=1` 添加到 APCu 的配置中。

Coding For Daemon Queue Workers

Daemon queue workers do not restart the framework before processing each job. Therefore,

you should be careful to free any heavy resources before your job finishes. For example, if you are doing image manipulation with the GD library, you should free the memory with `imagedestroy` when you are done.

Similarly, your database connection may disconnect when being used by long-running daemon. You may use the `DB::reconnect` method to ensure you have a fresh connection.

推送队列

您可以利用 Laravel 4 强大的队列架构来进行推送队列工作，不需要执行任何的常驻事务或后台监听，目前只支持 Iron.io 驱动，在您开始前建立一个 Iron.io 帐号及新增您的 Iron 凭证到 `app/config/queue.php` 配置文件。

注册一个推送队列订阅

接下来，您可以使用 `queue:subscribe` 命令注册一个URL来接收新的队列推送工作：

```
php artisan queue:subscribe queue_name http://foo.com/queue/receive
```

现在当您登入您的Iron仪表板，您将会看到您新的推送队列，以及订阅的 URL，您可以订阅许多的 URLs给您希望的队列，接下来建立一个 route 给您的 `queue/receive` 及从 `Queue::marshal` 方法回传回应：

```
Route::post('queue/receive', function()
{
    return Queue::marshal();
});
```

`marshal` 方法会将工作处理到正确的类，而发送工作到队列中只要使用一样的 `Queue::push` 方法。

已失败的工作

事情往往不会如您预期的一样，有时后您推送工作到队列会失败，别担心，Laravel 包含一个简单的方法去指定一个工作最多可以被执行几次，在工作被执行到一定的次数时，他将会新增至 `failed_jobs` 数据库表里，然后失败工作的数据库表名称可以在 `app/config/queue.php` 里进行设定：

要新增一个migration建立 `failed_jobs` 数据库表，您可以使用 `queue:failed-table` 命令：

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

您可以指定一个最大值来限制一个工作应该最多被执行几次通过 `--tries` 这个选项参数在您执

行 `queue:listen` 的时后：

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

假如您会想注册一个事件，这个事件会将会在队列失败时被调用，您可以使用 `Queue::failing` 方法，这个事件是一个很好的机会让您可以通知您的团队通过 e-mail 或 HipChat。

```
Queue::failing(function($connection, $job, $data)
{
    //
});
```

要看到所有的失败工作，您可以使用 `queue:failed` 命令：

```
php artisan queue:failed
```

`queue:failed` 命令将会列出工作的 ID、连接、队列名称及失败的时间，工作的 ID 也可以重新执行一个已经失败的工作，例如一个已经失败的工作他的 ID 是 5，我们可以使用下面的命令：

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

假如您会想删除一个已失败的工作，您可以使用 `queue:forget` 命令：

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

要删除全部失败的工作您可以使用 `queue:flush` 命令：

```
php artisan queue:flush
```

会话

配置信息

由于 HTTP 协定是无状态（Stateless）的，所以 session 提供一种储存用户数据的方法。Laravel 支持了多种 session 后端驱动，并通过清楚、统一的 API 提供使用。也内建支持如 Memcached，Redis 和数据库的后端驱动。

session 的配置文件配置在 `app/config/session.php` 中，请务必看一下 session 配置文件中可用的选项设定及注释。Laravel 默认使用 `file` 的 session 驱动，它在多数的应用中可以良好运作。

保留键值

Laravel 框架在内部有使用 `flash` 作为 session 的键值，所以应该您在使用 session 时应避免使用此名称。

Session 用法

储存数据到 **Session** 中

```
Session::put('key', 'value');
```

储存数据进 **Session** 数组值中

```
Session::push('user.teams', 'developers');
```

从 **Session** 取回数据

```
$value = Session::get('key');
```

从 **Session** 取回数据，若无则回传默认值

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

$value = Session::get('key', function() { return 'default'; });
```

从 **Session** 取回数据，并删除

```
$value = Session::pull('key', 'default');
```

从 **Session** 取出所有数据

```
$data = Session::all();
```

判断数据在 **Session** 中是否存在

```
if (Session::has('users'))
{
    //
}
```

移除 **Session** 中指定的数据

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

清空整个 **Session**

```
Session::flush();
```

重新产生 **Session ID**

```
Session::regenerate();
```

快闪数据（**Flash Data**）

有时您可能希望暂存一些数据，并只在下次请求有效。您可以使用 `Session::flash` 方法来达成目的：

```
Session::flash('key', 'value');
```

刷新当前快闪数据，延长到下次请求

```
Session::reflash();
```

只刷新指定快闪数据

```
Session::keep(array('username', 'email'));
```

数据库 **Sessions**

当使用 `database` session 驱动时，您必需创建一张储存 session 的数据库表。下面的例子使用 `Schema` 来建表：

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

当然您也可以使用 Artisan 命令 `session:table` 来建表：

```
php artisan session:table

composer dump-autoload

php artisan migrate
```

Session 驱动

session 配置文件中的 "driver" 定义了 session 数据将以哪种方式被储存。Laravel 提供了许多良好的驱动：

- `file` - sessions 将储存在 `app/storage/sessions` 文件夹中。
- `cookie` - sessions 将安全储存在加密的 cookies 中。
- `database` - sessions 将储存在您的应用程序数据库中。
- `memcached` / `redis` - sessions 将储存在一个高速缓存的系统中。
- `array` - sessions 将单纯的以 PHP 数组储存，只存在当次请求。

> 注意：运行 unit tests 时，session 驱动会被设定为 `array`，所以不会留下任何 session 数据。

远程连接

配置文件

Laravel 可以简单的通过 SSH 方式登录到远程服务器并执行相关操作命令，让您可以简单在远程执行的建立 Artisan 任务。`SSH` facade 提供了此类行为的入口。

相关的配置文件存在 `app/config/remote.php` 中，此文件包含所有您需要设定的远程连接配置，`connections` 数组里有以远程服务器名称作为键值的列表。只要在 `connections` 数组设定好认证，您就准备好可以执行远程任务了。记得 `SSH` 可以使用密码或 SSH key 进行身份认证。

> 提示：需要在远程服务器执行很多任务吗？请阅读 Envoy 任务执行!

基本用法

在在默认服务器执行命令

使用 `SSH::run` 方法，在默认的远程服务器执行命令：

```
SSH::run(array(
    'cd /var/www',
    'git pull origin master',
));
```

在特定服务器执行命令

您也可以使用 `into` 方法在特定的服务器上执行命令：

```
SSH::into('staging')->run(array(
    'cd /var/www',
    'git pull origin master',
));
```

捕捉命令的输出

您可以经由传入闭合函数到 `run` 方法，捕捉远程命令的即时输出：

```
SSH::run($commands, function($line)
{
    echo $line.PHP_EOL;
});
```

任务

如果您需要定义一组一起执行的命令，您可以用 `define` 方法定义一个「任务」：

```
SSH::into('staging')->define('deploy', array(
    'cd /var/www',
    'git pull origin master',
    'php artisan migrate',
));
```

您可以用 `task` 方法执行定义过的任务：

```
SSH::into('staging')->task('deploy', function($line)
{
    echo $line.PHP_EOL;
});
```

SFTP 下载

`SSH` 类里有简单的方式可以下载文件，使用 `get` 和 `getString` 方法：

```
SSH::into('staging')->get($remotePath, $localPath);

$contents = SSH::into('staging')->getString($remotePath);
```

SFTP 上传

`SSH` 类里也有简单的方式可以上传文件或甚至是字串到远程服务器，使用 `put` 和 `putString` 方法：

```
SSH::into('staging')->put($localFile, $remotePath);

SSH::into('staging')->putString($remotePath, 'Foo');
```

编辑远程日志

Laravel 有一个有用的命令可以让您在任何远程服务器的 `laravel.log` 尾端附加日志内容。使用 Artisan 的 `tail` 命令以及指定远程连线的服务器名称：

```
php artisan tail staging

php artisan tail staging --path=/path/to/log.file
```

Envoy 任务执行

Laravel Envoy 提供了简洁，轻量的语法能让您对远程服务器执行任务操作。使用 Blade 风格的语法，您可以简单的设置部署任务，执行 Artisan 命令。

> 提醒: Envoy 需要 PHP 5.4 或更高的版本，并且只能在 Mac / Linux 发行版本下执行。

安装

首先，使用 Composer `global` 命令安装 Envoy：

```
composer global require "laravel/envoy=~1.0"
```

记得将 `~/.composer/vendor/bin` 路径加入 PATH，如此在终端机执行 `envoy` 命令时才找得到。

再来，在项目根目录建立 `Envoy.blade.php` 文件。这里有个例子可以让您作为示例：

```
@servers(['web' => '192.168.1.1'])

@task('foo', ['on' => 'web'])
    ls -la
@endtask
```

如您所见，`@servers` 数组建立在文件的开头。您可以在定义任务时，在 `on` 选项里参照这些服务器。在您的 `@task` 定义里，写入想要在远程服务器执行的 Bash code。

`init` 命令可以简单的建立一个基本的 Envoy 文件：

```
envoy init user@192.168.1.1
```

执行任务

以 envoy 的 `run` 命令去执行您设定的任务：

```
envoy run foo
```

如有需要，您可以传递命令行参数到 Envoy 文件：

```
envoy run deploy --branch=master
```

您也可以经由您所熟悉的 Blade 语法使用这些参数：

```
@servers(['web' => '192.168.1.1'])

@task('deploy', ['on' => 'web'])
    cd site
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

Bootstrapping

您可以在 Envoy 文件里使用 `@setup` 语法定义 PHP 变量和执行一般的 PHP 代码：

```
@setup
    $now = new DateTime();

    $environment = isset($env) ? $env : "testing";
@endsetup
```

您也可以使用 `@include` 引入 PHP 文件：

```
@include('vendor/autoload.php');
```

多服务器

您可以简单的在多个服务器执行任务。只要在任务定义里列出服务器名称：

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2']])
    cd site
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

默认任务会循序的在每个服务器上执行。意味着任务会在第一个服务器执行完后才会换到下一个。

平行执行

如果您想在多个服务器上同时执行任务，只要简单的在任务定义里加上 `parallel` 选项：

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2'], 'parallel' => true])
    cd site
    git pull origin {{ $branch }}
```

```
    php artisan migrate
@endtask
```

任务宏

宏让您可以使用一个命令就循序执行一组任务。例如：

```
@servers(['web' => '192.168.1.1'])

@macro('deploy')
    foo
    bar
@endmacro

@task('foo')
    echo "HELLO"
@endtask

@task('bar')
    echo "WORLD"
@endtask
```

现在 `deploy` 宏可以经由一个简单的命令执行：

```
envoy run deploy
```

提醒通知

HipChat

您可能想要在执行完任务后，发送通知到团队的 HipChat 聊天室，使用简单的 `@hipchat` 定义：

```
@servers(['web' => '192.168.1.1'])

@task('foo', ['on' => 'web'])
    ls -la
@endtask

@after
    @hipchat('token', 'room', 'Envoy')
@endafter
```

您也可以自定义发送到 hipchat 聊天室的信息，任何在 `@setup` 里定义，或是经由 `@include` 引入的变量都可以使用在信息里：

```
@after
    @hipchat('token', 'room', 'Envoy', "$task ran on [$environment]")
@endafter
```

这是一个令人惊艳的简单方式，让您的团队保持通知在服务器执行的任务。

Slack

下面的语法可以发送通知到 Slack：

```
@after
    @slack('team', 'token', 'channel')
@endafter
```

更新 Envoy

简单的执行 `self-update` 命令即可更新 Envoy：

```
envoy self-update
```

如果您的 Envoy 安装在 `/usr/local/bin`，您可能需要加上 `sudo`：

```
composer global update
```

模板

控制器布局

在Laravel框架中使用模板的一种方法就是通过控制器布局。通过在控制器中指定`layout`属性，指定的视图就会被创建，并作为默认数据，在actions中返回。

在控制器中定义布局（**Layouts**）

```
class UserController extends BaseController {

    /**
     * The layout that should be used for responses.
     */
    protected $layout = 'layouts.master';

    /**
     * Show the user profile.
     */
    public function showProfile()
    {
        $this->layout->content = View::make('user.profile');
    }

}
```

Blade模板

Blade是Laravel框架下的一个简单但又强大的模板引擎。不同于控制器布局，Blade模板引擎由 模板继承 和 模板片段 驱动。所有的Blade模板文件必须使用 `.blade.php` 文件扩展名。

定义一个**Blade**布局

```
<!-- Stored in app/views/layouts/master.blade.php -->

<html>
    <body>
        @section('sidebar')
            This is the master sidebar.
        @show

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

```
</html>
```

使用一个**Blade**布局

```
@extends('layouts.master')

@section('sidebar')


    <p>This is appended to the master sidebar.</p>
@stop

@section('content')
    <p>This is my body content.</p>
@stop
```

注意视图中片段只是简单的替换其`extend`的Blade布局中相应片段。通过在模板片段中使用指令，布局的内容可以包含一个子视图，这样你就可以在布局片段中添加诸如侧边栏、底部信息等内容。

有时候，有些片段可能不能确定被定义了，你可以使用`@yield`结构给出一个默认值。如下，第二个值即是默认值。

```
@yield('section', 'Default Content');
```

其他 Blade模板 控制结构

输出数据

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

The current UNIX timestamp is {{{ time() }}}.
```

检测是否存在后输出数据

有时，你可能希望输出一个变量，但又不能确定这个变量是否被设置。直接点，你可能想这么做：

```
{{{ isset($name) ? $name : 'Default' }}}
```

然而，除了写一个三目运算符，Blade有如下简写方法：

```
{{{ $name or 'Default' }}}
```

显示带有大括号的文本

如果你想显示带有大括号的字符串，你可以在文本前放`@`符号，这样会忽略Blade解析行为

```
@{{ This will not be processed by Blade }}
```

当然，用户提供的全部字符串都应该都被转义（主要对html标签，利用htmlentities编码转义）。如果想转义输出，你可以使用三个大括号语法：

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

如果不希望数据被转义，可以使用双大括号语法：

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

> 注意：一定要小心输出的用户提供的内容。使用三个大括号的语法能够直接输出内容中的HTML标签。

If标签

```
@if (count($records) === 1)
    I have one record!
@elseif (count($records) > 1)
    I have multiple records!
@else
    I don't have any records!
@endif

@unless (Auth::check())
    You are not signed in.
@endunless
```

循环

```
@for ($i = 0; $i < 10; $i++)
    The current value is {{ $i }}
@endfor

@foreach ($users as $user)
    <p>This is user {{ $user->id }}</p>
@endforeach

@while (true)
    <p>I'm looping forever.</p>
@endwhile
```

包含子视图

```
@include('view.name')
```

你也可以传递数组数据到被包含的视图

```
@include('view.name', array('some'=>'data'))
```

覆盖片段

如果想覆盖一整个片段，可以使用`overwrite`指令：

```
@extends('list.item.container')

@section('list.item.content')
    <p>This is an item of type {{ $item->type }}</p>
@overwrite
```

输出多语言

```
@lang('language.line')

@choice('language.line', 1);
```

注释

```
{{-- This comment will not be in the rendered HTML --}}
```

扩展Blade

Blade允许用户定义自己的控制结构。当一个Blade文件被编译后，会调用用户自定义的扩展，用来处理视图内容，从简单的`str_replace`操作，到很复杂的表达式，总之，你可以做任何事情。

Blade的编译器附带了帮助函数`createMatcher`和`createPlainMatcher`，这两个函数可以生成自定义指令。

`createPlainMatcher`函数主要用于没有参数传递的指令，类似`@endif`和`@stop`，而`createMatcher`则用于那些有参数传递的指令。

下面的例子创建`@datetime($var)`指令，它只是简单的对`$var`调用`->format()`方法：

```
Blade::extend(function($view, $compiler)
{
    $pattern = $compiler->createMatcher('datetime');

    return preg_replace($pattern, '$1<?php echo $2->format('m/d/Y H:i'); ?>', $view);
});
```

测试

介绍

Laravel 在创立时就有考虑到单元测试。事实上，它能立即使用被引入的 PHPUnit 做测试，而且默认已经为您的应用程序建立了 `phpunit.xml` 文件。除了 PHPUnit 以外，Laravel 也利用 Symfony HttpKernel、DomCrawler 和 BrowserKit 组件让您在测试的时候模拟一个网页浏览器来检查和处理您的视图。

在 `app/tests` 文件夹中有提供一个测试例子。在安装好新的 Laravel 应用程序之后，只要在命令行上执行 `phpunit` 来进行测试流程。

定义并执行测试

要建立一个测试案例，只要在 `app/tests` 文件夹建立新的测试文件。测试类必须继承自 `TestCase`，接着您可以如平常使用 PHPUnit 一般去定义测试方法。

测试类例子

```
class FooTest extends TestCase {

    public function testSomethingIsTrue()
    {
        $this->assertTrue(true);
    }

}
```

您可以从命令行执行 `phpunit` 命令来执行应用程序的所有测试。

> 注意：如果您定义自己的 `setUp` 方法，请记得调用 `parent::setUp`。

测试环境

当执行单元测试的时候，Laravel 会自动将环境设置在 `testing` 中。另外 Laravel 会在测试环境中加载 `session` 和 `cache` 的配置文件，在测试环境中这两个驱动会被设定为 `array` (空数组)，表示在测试的时候不对 session 或 cache 数据进行持久化操作。必要时你也可以创建其他的测试环境。

在测试中调用路由

从单一测试中调用路由

您可以使用 `call` 方法方便地调用您的其中一个路由来进行测试:

```
$response = $this->call('GET', 'user/profile');

$response = $this->call($method, $uri, $parameters, $files, $server, $content);
```

接着您可以检查 `Illuminate\Http\Response` 对象:

```
$this->assertEquals('Hello World', $response->getContent());
```

在测试中调用控制器

您也可以在测试中调用控制器：

```
$response = $this->action('GET', 'HomeController@index');

$response = $this->action('GET', 'UserController@profile', array('user' => 1));
```

`getContent` 方法会回传求值后的字符串内容. 如果您的路由回传一个 `View`, 您可以通过 `original` 属性获取它:

```
$view = $response->original;

$this->assertEquals('John', $view['name']);
```

您可以使用 `callSecure` 方法去调用 HTTPS 路由:

```
$response = $this->callSecure('GET', 'foo/bar');
```

> 注意: 在测试环境中，路由过滤器是被禁用的。如果要启用它们，必须增加 `Route::enableFilters()` 到您的测试中。

DOM Crawler

您也可以通过调用路由来取得 DOM Crawler 实例来检查内容：

```
$crawler = $this->client->request('GET', '/');
```

```
$this->assertTrue($this->client->getResponse()->isOk());

$this->assertCount(1, $crawler->filter('h1:contains("Hello World!")'));
```

如需更多如何使用 DOM Crawler 的信息，请参考它的官方文件.

模拟 Facades

当测试的时候，您或许常会想要模拟调用 Laravel 静态 facade。举个例子，参考下面的控制器行为:

```
public function getIndex()
{
    Event::fire('foo', array('name' => 'Dayle'));

    return 'All done!';
}
```

我们可以在 facade 上使用 `shouldReceive` 方法，模拟调用 `Event` 类，它将会回传一个 Mockery mock 对象实例。

模拟 Facade

```
public function testGetIndex()
{
    Event::shouldReceive('fire')->once()->with('foo', array('name' => 'Dayle'));

    $this->call('GET', '/');
}
```

> 注意: 您不应该模拟 `Request` facade。当执行您的测试，应该传递想要的输入数据给 `call` 方法。

框架 Assertions

Laravel 附带几个 `assert` 方法，能让测试更简单一点:

Assert 回应为 OK

```
public function testMethod()
{
    $this->call('GET', '/');

    $this->assertResponseOk();
}
```

Assert 回应状态码

```
$this->assertResponseStatus(403);
```

Assert 回应为重定向

```
$this->assertRedirectedTo('foo');

$this->assertRedirectedToRoute('route.name');

$this->assertRedirectedToAction('Controller@method');
```

Assert 回应带数据的视图

```
public function testMethod()
{
    $this->call('GET', '/');

    $this->assertViewHas('name');
    $this->assertViewHas('age', $value);
}
```

Assert 回应带数据的 Session

```
public function testMethod()
{
    $this->call('GET', '/');

    $this->assertSessionHas('name');
    $this->assertSessionHas('age', $value);
}
```

Assert 回应带错误信息的 Session

```
public function testMethod()
{
    $this->call('GET', '/');

    $this->assertSessionHasErrors();

    // Asserting the session has errors for a given key...
    $this->assertSessionHasErrors('name');

    // Asserting the session has errors for several keys...
    $this->assertSessionHasErrors(array('name', 'age'));
}
```

Assert 回应带数据的旧输入内容

```
public function testMethod()
{
    $this->call('GET', '/');

    $this->assertHasOldInput();
```

```
}
```

辅助方法

`TestCase` 类包含几个辅助方法让应用程序的测试更为简单。

从测试里设定和刷新 **Sessions**

```
$this->session(['foo' => 'bar']);

$this->flushSession();
```

设定目前经过验证的用户

您可以使用 `be` 方法设定目前经过验证的用户:

```
$user = new User(array('name' => 'John'));

$this->be($user);
```

您可以从测试中使用 `seed` 方法重新填充您的数据库:

从测试中重新填充数据库

```
$this->seed();

$this->seed($connection);
```

更多建立填充数据的信息可以在文件的 迁移与数据填充 部分找到。

重置应用程序

您可能已经知道，您可以通过 `$this->app` 在任何测试方法中获取您的 Laravel `应用程序本体` / IoC 容器。这个应用程序对象实例会在每个测试类里被重置。如果您希望在给定的方法中手动强制重置应用程序，则可以在测试方法使用 `refreshApplication` 方法。这将会重置任何额外的绑定，例如那些从测试案例执行开始被放到 IoC 容器的 mocks。

验证

基本用法

Laravel 通过 `Validation` 类让您可以简单、方便的验证数据正确性及查看相应的验证错误信息。

基本验证例子

```
$validator = Validator::make(
    array('name' => 'Dayle'),
    array('name' => 'required|min:5')
);
```

上文中通过 `make` 这个方法来的第一个参数来设定所需要被验证的数据名称，第二个参数设定该数据可被接受的规则。

使用数组来定义规则

多个验证规则可以使用"|"符号分隔，或是单一数组作为单独的元素分隔。

```
$validator = Validator::make(
    array('name' => 'Dayle'),
    array('name' => array('required', 'min:5'))
);
```

验证多个字段

```
$validator = Validator::make(
    array(
        'name' => 'Dayle',
        'password' => 'lamepassword',
        'email' => 'email@example.com'
    ),
    array(
        'name' => 'required',
        'password' => 'required|min:8',
        'email' => 'required|email|unique:users'
    )
);
```

当一个 `Validator` 实例被建立，`fails`（或 `passes`）这两个方法就可以在验证时使用，如下：

```
if ($validator->fails())
{
    // The given data did not pass validation
}
```

假如验证失败，您可以从验证器中接收错误信息。

```
$messages = $validator->messages();
```

您可能不需要错误信息，只想取得无法通过验证的规则，您可以使用 'failed' 方法：

```
$failed = $validator->failed();
```

验证文件

`Validator` 类提供了一些规则用来验证文件，例如 `size`，`mimes` 等等。当需要验证文件时，您仅需将它们和您其他的数据一同送给验证器即可。

使用错误信息

当您调用一个 `Validator` 实例的 `messages` 方法后，您会得到一个命名为 `MessageBag` 的变量，该变量里有许多方便的方法能让您取得相关的错误信息。

查看一个字段的第一个错误信息

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

查看一个字段的所有错误信息

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

查看所有字段的所有错误信息

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

判断一个字段是否有错误信息

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

```
    //
}
```

错误信息格式化输出

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

> 注意: 默认错误信息以 Bootstrap 兼容语法输出。

查看所有错误信息并以格式化输出

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

错误信息 & 视图

当您开始进行验证数据时，您会需要一个简易的方法去取得错误信息并回传到您的视图中，在 Laravel 中您可以很方便的处理这些操作，您可以通过下面的路由例子来了解：

```
Route::get('register', function()
{
    return View::make('user.register');
});

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

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

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

需要记住的是，当验证失败后，我们会使用 `withErrors` 方法来将 `Validator` 实例进行重新导向。这个方法会将错误信息存入 session 中，这样才能在下个请求中被使用。

然而，我们并不需要特别去将错误信息绑定在我们 GET 路由的视图中。因为 Laravel 会确认在 Session 数据中检查是否有错误信息，并且自动将它们绑定至视图中。所以请注意，**`$errors`** 变量存在于所有的视图中，所有的请求里，让您可以直接假设 `$errors` 变量已被定义且可以安全地使用。`$errors` 变量是 `MessageBag` 类的一个实例。

所以，重新导向之后，您可以自然的在视图中使用 `$errors` 变量：

```
<?php echo $errors->first('email'); ?>
```

命名错误清单

假如您在一个页面中有许多的表单, 您可能希望为错误命名一个`MessageBag`. 这样能方便您针对特定的表单查看其错误信息, 我们只要简单的在 `withErrors` 的第二个参数设定名称即可:

```
return Redirect::to('register')->withErrors($validator, 'login');
```

接着您可以从一个 `$errors` 变量中取得已命名的 `MessageBag` 实例:

```
<?php echo $errors->login->first('email'); ?>
```

现有的的验证规则

以下是现有的的验证规则清单与他们的函数名称:

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

accepted

字段值为 *yes*, *on*, 或是 *1* 时，验证才会通过。这在确认"服务条款"是否同意时很有用。

active_url

字段值通过 PHP 函数 `checkdnsrr` 来验证是否为一个有效的网址。

after:*date*

验证字段是否是在指定日期之后。这个日期将会使用 PHP `strtotime` 函数验证。

alpha

字段仅全数为字母字串时通过验证。

alpha_dash

字段值仅允许字母、数字、破折号（-）以及底线（_）

alpha_num

字段值仅允许字母、数字

array

字段值仅允许为数组

before:*date*

验证字段是否是在指定日期之前。这个日期将会使用 PHP `strtotime` 函数验证。

between:*min*,*max*

字段值需介于指定的 *min* 和 *max* 值之间。字串、数值或是文件都是用同样的方式来进行验证。

boolean

需要验证的字段必须可以转换为 boolean 类型的值。可接受的输入是`true`、`false`、`1`、`0`、`"1"` 和 `"0"`。

confirmed

字段值需与对应的字段值 `foo_confirmation` 相同。例如，如果验证的字段是 `password`，那对应的字段 `password_confirmation` 就必须存在且与 `password` 字段相符。

date

字段值通过 PHP `strtotime` 函数验证是否为一个合法的日期。

date_format:*format*

字段值通过 PHP `date_parse_from_format` 函数验证符合 *format* 制定格式的日期是否为合法日期。

different:*field*

字段值需与指定的字段 *field* 值不同。

digits:*value*

字段值需为数字且长度需为 *value*。

digits_between:*min*,*max*

字段值需为数字，且长度需介于 *min* 与 *max* 之间。

email

字段值需符合 email 格式。

exists:*table*,*column*

字段值需与存在于数据库 *table* 中的 *column* 字段值其一相同。

Exists 规则的基本使用方法

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

指定一个自定义的字段名称

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

您可以指定更多条件且那些条件将会被新增至 "where" 查询里：

```
'email' => 'exists:staff,email,account_id,1'
/* 这个验证规则为 email 需存在于 staff 这个数据库表中 email 字段中且 account_id=1 */
```

通过`NULL`搭配"where"的缩写写法去检查数据库的是否为`NULL`

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

image

文件必需为图片(jpeg, png, bmp 或 gif)

in:*foo*,*bar*,...

字段值需符合事先给予的清单的其中一个值

integer

字段值需为一个整数值

ip

字段值需符合 IP 位址格式。

max:*value*

字段值需小于等于 *value*。字串、数字和文件则是判断 `size` 大小。

mimes:*foo*,*bar*,...

文件的 MIME 类需在给定清单中的列表中才能通过验证。

MIME规则基本用法

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

min:*value*

字段值需大于等于 *value*。字串、数字和文件则是判断 `size` 大小。

not_in:*foo*,*bar*,...

字段值不得为给定清单中其一。

numeric

字段值需为数字。

regex:*pattern*

字段值需符合给定的正规表示式。

注意: 当使用`regex`模式时, 您必须使用数组来取代"|"作为分隔, 尤其是当正规表示式中含有"|"字串。

required

字段值为必填。

required_if:*field*,*value*,...

字段值在 *field* 字段值为 *value* 时为必填。

required_with:*foo*,*bar*,...

字段值 仅在 任一指定字段有值情况下为必填。

required_with_all:*foo*,*bar*,...

字段值 仅在 所有指定字段皆有值情况下为必填。

required_without:*foo*,*bar*,...

字段值 仅在 任一指定字段没有值情况下为必填。

required_without_all:*foo*,*bar*,...

字段值 仅在 所有指定字段皆没有值情况下为必填。

same:*field*

字段值需与指定字段 *field* 等值。

size:*value*

字段值的尺寸需符合给定 *value* 值。对于字串来说，*value* 为需符合的字串长度。对于数字来说，*value* 为需符合的整数值。对于文件来说，*value* 为需符合的文件大小（单位 kb）。

timezone

字段值通过 PHP `timezone_identifiers_list` 函数来验证是否为有效的时区。

unique:*table*,*column*,*except*,*idColumn*

字段值在给定的数据库中需为唯一值。如果 `column（字段）` 选项没有指定，将会使用字段名称。

唯一**(Unique)**规则的基本用法

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

指定一个自定义的字段名称

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

强制唯一规则忽略指定的 **ID**

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

增加额外的 **Where** 条件

您也可以指定更多的条件式到 "where" 查询语句中：

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

上述规则为只有 `account_id` 为 `1` 的数据列会做唯一规则的验证。

url

字段值需符合 URL 的格式。

> 注意：此函数会使用 PHP `filter_var` 方法验证。

添加条件验证规则

某些情况下，您可能 只想 当字段有值时，才进行验证。这时只要增加 `sometimes` 条件进条件列表中，就可以快速达成：

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

在上述例子中，`email` 字段只会在当其在 `$data` 数组中有值的情况下才会被验证。

复杂的条件式验证

有时，您可以希望给指定字段在其他字段长度有超过 100 时才验证是否为必填。或者您希望有两个字段，当其中一字段有值时，另一字段将会有一个默认值。增加这样的验证条件并不复杂。首先，利用您尚未更动的 静态规则 创建一个 `Validator` 实例：

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

假设我们的网页应用程序是专为游戏收藏家所设计。如果游戏收藏家收藏超过一百款游戏，我们希望他们说明为什么他们拥有这么多游戏。如，可能他们经营一家二手游戏商店，或是他们可能只是享受收集的乐趣。有条件的加入此需求，我们可以在 `Validator` 实例中使用 `sometimes` 方法。

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

传递至 `sometimes` 方法的第一个参数是我们要条件式认证的字段名称。第二个参数是我们想加入验证规则。`闭包（Closure）` 作为第三个参数传入，如果回传值为 `true` 那该规则就会被加入。这个方法可以轻而易举的建立复杂的条件式验证。您也可以一次对多个字段增加条件式验证：

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

> 注意：传递至您的 `Closure` 的 `$input` 参数为 `Illuminate\Support\Fluent` 的实例且用来作为获

> 取您的输入及文件的对象。

自定义错误信息

如果有需要，您可以设置自定义的错误信息取代默认错误信息。这里有几个方式可以设定自定义消息。

传递自定义消息进验证器

```
$messages = array(
    'required' => 'The :attribute field is required.',
);

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

> 注意：在验证中，`:attribute` 占位符会被字段的实际名称给取代。您也可以在验证信息中使用其他的占位符。

其他的验证占位符

```
$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',
);
```

为特定属性给予一个自定义信息

有时您只想为一个特定字段指定一个客制错误信息：

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

在语言包文件中指定自定义消息

某些状况下，您可能希望在语言包文件中设定您的自定义消息，而非直接将他们传递给 `Validator`。要达到目的，将您的信息增加至 `app/lang/xx/validation.php` 文件的 `custom` 数组中。

```
'custom' => array(
    'email' => array(
        'required' => 'We need to know your e-mail address!',
    ),
),
```

自定义验证规则

注册自定义验证规则

Laravel 提供了各种有用的验证规则，但是，您可能希望可以设定自定义验证规则。注册生成自定义的验证规则的方法之一就是使用 `Validator::extend` 方法：

```
Validator::extend('foo', function($attribute, $value, $parameters)
{
    return $value == 'foo';
});
```

自定义验证器闭包接收三个参数：要被验证的 `$attribute(属性)` 的名称，属性的值 `$value`，传递至验证规则的 `$parameters` 数组。

您同样可以传递一个类和方法到 `extend` 方法中，取代原本的闭包：

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

注意，您同时需要为您的自定义规则订立一个错误信息。您可以使用行内自定义信息数组或是在认证语言文件里新增。

扩展 **Validator** 类

除了使用闭包回呼(Closure callbacks)来扩展 Validator 外，您一样可以直接扩展 Validator 类。您可以写一个扩展自 `Illuminate\Validation\Validator` 的验证器类。您也可以增加验证方法到以 `validate` 为开头的类中：

```
<?php

class CustomValidator extends Illuminate\Validation\Validator {

    public function validateFoo($attribute, $value, $parameters)
    {
        return $value == 'foo';
    }

}
```

拓展自定义验证器

接下来，您需要注册您自定义验证器扩展：

```
Validator::resolver(function($translator, $data, $rules, $messages)
{
    return new CustomValidator($translator, $data, $rules, $messages);
});
```

当创建自定义验证规则时，您可能有时需要为错误信息定义自定义的占位符。您可以如上所述创

建一个自定义的验证器，然后增加 `replaceXXX` 函数进验证器中。

```
protected function replaceFoo($message, $attribute, $rule, $parameters)
{
    return str_replace(':foo', $parameters[0], $message);
}
```

如果您想要增加一个自定义信息 "replacer" 但不扩展 `Validator` 类，您可以使用 `Validator::replacer` 方法：

```
Validator::replacer('rule', function($message, $attribute, $rule, $parameters)
{
    //
});
```

数据库

数据库使用基础

配置

Laravel让连接和使用数据库变得异常简单.数据库配置文件是 `app/config/database.php`.你可以在配置文件中定义所有你的数据库连接,以及指定默认连接.这个文件中已经提供了所有支持的数据库系统连接例子.

目前Laravel支持四种数据库系统: MySQL, Postgres, SQLite, 和 SQL Server.

读 / 写 连接 (读写分离配置)

你可能需要使用一个连接来处理 SELECT(读)数据操作,而另一个链接来处理 INSERT, UPDATE, 和 DELETE 这样的操作(即读写分离).Laravel是这项工作变得简单,它会在你直接发送查询语句、使用查询构建器或者Eloquent ORM时自动选择适当的连接(来实现读写分离).

以下是一个读/写 连接配置的例子:

```
'mysql' => array(
    'read' => array(
        'host' => '192.168.1.1',
    ),
    'write' => array(
        'host' => '196.168.1.2'
    ),
    'driver'    => 'mysql',
    'database'  => 'database',
    'username'  => 'root',
    'password'  => '',
    'charset'   => 'utf8',
    'collation' => 'utf8_unicode_ci',
    'prefix'    => '',
),
```

注意看两个增加的配置数组: `read` 和 `write`.这两个配置数组值都是一个键值对: `host`(用来标明连接的服务器).其余部分的配置由于 `read` 和 `write` 是相同的,因此他们都在 `mysql` 下面被合并到了一起.因此,我们只需要修改 `read` 和 `write` 配置中我们需要覆盖配置的值即可.在这个例子里, `192.168.1.1` 将会做为"读"连接 而 `192.168.1.2`将会做为"写"连接.数据库的用户名密码,表前缀,字

符集和其他的配置项目会在 `mysql` 配置下为两个连接所共用.

执行crud语句

一旦你已经配置好了数据库的连接,你就可以直接使用 `DB` 类来发送sql请求了.

运行一个 **Select** 查询

```
$results = DB::select('select * from users where id = ?', array(1));
```

这里的 `select` 方法将会一直返回一个 `array`(数组) 结果集.

运行一个 **Insert(插入)** 请求

```
DB::insert('insert into users (id, name) values (?, ?)', array(1, 'Dayle'));
```

运行一个 **Update(更新)** 请求

```
DB::update('update users set votes = 100 where name = ?', array('John'));
```

运行一个 **Delete(删除)** 请求

```
DB::delete('delete from users');
```

> 注意: `update`(更新) 和 `delete`(删除) 将会返回操作的影响行数(affect rows).

执行非**crud**操作

```
DB::statement('drop table users');
```

监听数据库操作事件

你可以使用 `DB::listen` 方法来监听query事件:

```
DB::listen(function($sql, $bindings, $time)
{
    //
});
```

事务

你可以用 `transaction` 方法来调用一个事务集合:

```
DB::transaction(function()
{
    DB::table('users')->update(array('votes' => 1));
```

```
    DB::table('posts')->delete();
});
```

> 注意: 任何 `transaction` 抛出的异常都将导致事务自动回滚.

有时你需要自己手动开启一个事务:

```
DB::beginTransaction();
```

你可以用过 `rollback` 方法手动回滚事务:

```
DB::rollback();
```

最后,你可以使用 `commit` 方法来提交:

```
DB::commit();
```

同时使用多个数据库系统

你可能使用很多的数据库系统,你可以使用 `DB::connection` 方法来选择使用它们:

```
$users = DB::connection('foo')->select(...);
```

你可能需要在数据库系统的层面上操作数据库, 使用PDO实例即可:

```
$pdo = DB::connection()->getPdo();
```

使用reconnect方法重新连接一个指定的数据库系统:

```
DB::reconnect('foo');
```

你可以使用 `disconnect` 方法来手动断开数据库连接,防止PDO连接数超过 `max_connections` 的限制:

```
DB::disconnect('foo');
```

查询日志

Laravel默认会为当前请求执行的的所有查询生成日志并保存在内存中。因此，在某些特殊的情况下，比如一次性向数据库中插入大量数据，就可能导致内存不足。在这种情况下, 你可以通过 `disableQueryLog` 方法来关闭查询日志:

```
DB::connection()->disableQueryLog();
```

调用 `getQueryLog` 方法可以同时获取多个查询执行后的日志:

```
$queries = DB::getQueryLog();
```

查询构造器

介绍

数据库查询构造器 (query builder) 提供方便流畅的接口来建立、执行数据库查询语法。在您的应用程序里面，它可以被使用在大部分的数据 库操作，而且它在所有支持的数据库系统上都可以执行。

> 注意: Laravel 查询构造器使用 PDO 参数绑定，以保护应用程序免于SQL注入攻击 (SQL injection)，因此传入的参数不需过滤额外的特殊字符串。

Selects

从数据库表中取得所有的数据列

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

foreach ($users as $user)
{
    var_dump($user->name);
}
```

从数据库表中取得单一数据列

```
$user = DB::table('users')->where('name', 'John')->first();

var_dump($user->name);
```

从数据库表中取得单一数据列的单一字段

```
$name = DB::table('users')->where('name', 'John')->pluck('name');
```

取得单一字段值的列表

```
$roles = DB::table('roles')->lists('title');
```

这个方法将会回传含有数据库表 role 的 title 字段值的数组。您也可以通过下面的方法，为回传的数组指定自定义键值。

```
$roles = DB::table('roles')->lists('title', 'name');
```

指定查询子句 **(Select Clause)**

```
$users = DB::table('users')->select('name', 'email')->get();

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

$users = DB::table('users')->select('name as user_name')->get();
```

增加查询子句到现有的的查询中

```
$query = DB::table('users')->select('name');

$users = $query->addSelect('age')->get();
```

使用 **where** 及运算符

```
$users = DB::table('users')->where('votes', '>', 100)->get();
```

"or" 语法

```
$users = DB::table('users')
                    ->where('votes', '>', 100)
                    ->orWhere('name', 'John')
                    ->get();
```

使用 **Where Between**

```
$users = DB::table('users')
                    ->whereBetween('votes', array(1, 100))->get();
```

使用 **Where Not Between**

```
$users = DB::table('users')
```

```
                    ->whereNotBetween('votes', array(1, 100))->get();
```

使用 **Where In** 与数组

```
$users = DB::table('users')
                    ->whereIn('id', array(1, 2, 3))->get();

$users = DB::table('users')
                    ->whereNotIn('id', array(1, 2, 3))->get();
```

使用 **Where Null** 找有未设定的值的数据

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

排序**(Order By)**, 分群**(Group By)**, 及 **Having**

```
$users = DB::table('users')
                    ->orderBy('name', 'desc')
                    ->groupBy('count')
                    ->having('count', '>', 100)
                    ->get();
```

偏移**(Offset)** 及 限制**(Limit)**

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

Joins

查询构造器也可以使用 join 语法, 看看下面的例子：

基本的 **Join** 语法

```
DB::table('users')
            ->join('contacts', 'users.id', '=', 'contacts.user_id')
            ->join('orders', 'users.id', '=', 'orders.user_id')
            ->select('users.id', 'contacts.phone', 'orders.price')
            ->get();
```

Left Join 语法

```
DB::table('users')
        ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
        ->get();
```

您也可以指定更进阶的 join 子句

```
DB::table('users')
        ->join('contacts', function($join)
```

```
        {
            $join->on('users.id', '=', 'contacts.user_id')->orOn(...);
        })
        ->get();
```

如果您想在您的 join 中使用 where 型式的子句, 您可以在 join 子句里使用 `where` 或 `orWhere` 方法。下面的方法将会比较 contacts 数据库表中的 user_id 的字段值, 而不是比较两个字段。

```
DB::table('users')
        ->join('contacts', function($join)
        {
            $join->on('users.id', '=', 'contacts.user_id')
                 ->where('contacts.user_id', '>', 5);
        })
        ->get();
```

进阶 **Wheres**

群组化参数

有些时候您需要更进阶的 where 子句, 如 "where exists" 或多维数组参数。Laravel 的查询构造器也可以处理这样的情况；

```
DB::table('users')
            ->where('name', '=', 'John')
            ->orWhere(function($query)
            {
                $query->where('votes', '>', 100)
                      ->where('title', '<>', 'Admin');
            })
            ->get();
```

上面的查询语法会产生下方的 SQL：

```
select * from users where name = 'John' or (votes > 100 and title <> 'Admin')
```

Exists 语法

```
DB::table('users')
            ->whereExists(function($query)
            {
                $query->select(DB::raw(1))
                      ->from('orders')
                      ->whereRaw('orders.user_id = users.id');
            })
            ->get();
```

上面的查询语法会产生下方的 SQL：

```
select * from users
where exists (
```

```
    select 1 from orders where orders.user_id = users.id
)
```

聚合

查询构造器也提供各式各样的聚合方法，如 `count`, `max`, `min`, `avg` 及 `sum`。

使用聚合方法

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

$price = DB::table('orders')->max('price');

$price = DB::table('orders')->min('price');

$price = DB::table('orders')->avg('price');

$total = DB::table('users')->sum('votes');
```

Raw Expressions

有些时候您需要使用 raw expression 在查询语句里，这样的表达式会成为字串插入至查询中，因此要小心勿建立任何 SQL 注入的攻击点。要建立 raw expression，您可以使用 `DB::raw` 方法：

使用 **Raw Expression**

```
$users = DB::table('users')
                     ->select(DB::raw('count(*) as user_count, status'))
                     ->where('status', '<>', 1)
                     ->groupBy('status')
                     ->get();
```

新增

新增一条数据进数据库表

```
DB::table('users')->insert(
    array('email' => 'john@example.com', 'votes' => 0)
);
```

新增自动递增 **(Auto-Incrementing) ID** 的数据至数据库表

如果数据库表有自动递增的ID，可以使用 `insertGetId` 新增数据并回传该 ID：

```
$id = DB::table('users')->insertGetId(
    array('email' => 'john@example.com', 'votes' => 0)
);
```

> 注意：当使用 PostgreSQL 时，insertGetId 方法会预先自动递增字段名为 "id" 的值。

新增多条数据进数据库表

```
DB::table('users')->insert(array(
    array('email' => 'taylor@example.com', 'votes' => 0),
    array('email' => 'dayle@example.com', 'votes' => 0),
));
```

更新

更新数据库表中的数据

```
DB::table('users')
            ->where('id', 1)
            ->update(array('votes' => 1));
```

对字段递增或递减数值

```
DB::table('users')->increment('votes');

DB::table('users')->increment('votes', 5);

DB::table('users')->decrement('votes');

DB::table('users')->decrement('votes', 5);
```

您也可以同时更新其他字段

```
DB::table('users')->increment('votes', 1, array('name' => 'John'));
```

删除

删除数据库表中的数据

```
DB::table('users')->where('votes', '<', 100)->delete();
```

删除数据库表中的所有数据

```
DB::table('users')->delete();
```

清空数据库表

```
DB::table('users')->truncate();
```

Unions

查询构造器也提供一个快速的方法去"合并 (union)"两个查询的结果：

```
$first = DB::table('users')->whereNull('first_name');

$users = DB::table('users')->whereNull('last_name')->union($first)->get();
```

`unionAll` 方法也可以使用，它与 `union` 方法的使用方式一样。

悲观锁定 (Pessimistic Locking)

查询构造器提供了少数函数协助您在 SELECT 语句中做到“悲观锁定”。

您只要在 SELECT 查询语句中使用 `sharedLock`：

```
DB::table('users')->where('votes', '>', 100)->sharedLock()->get();
```

在 select 语法中，要"锁住更新(lock for update)"，您仅需在查询语句中使用 `lockForUpdate` 方法即可：

```
DB::table('users')->where('votes', '>', 100)->lockForUpdate()->get();
```

缓存查询结果

使用 `remember` 方法，您可以轻松的缓存查询结果：

```
$users = DB::table('users')->remember(10)->get();
```

这个例子中，查询结果将会缓存 10 分钟。当结果被缓存时，查询语句将不会被执行，而会从应用程序指定的缓存驱动器中载入缓存的结果。

如果您正在使用的是 支持标签的缓存驱动器，您也可以为缓存增加标签：

```
$users = DB::table('users')->cacheTags(array('people', 'authors'))->remember(10)->get();
```

Eloquent ORM

介绍

Laravel 的 Eloquent ORM 提供了漂亮、简洁的 ActiveRecord 实现来和数据库的互动。每个数据库表会和一个对应的「模型」互动。

在开始之前，记得把 `app/config/database.php` 里的数据库连接配置好。

基本用法

我们先从建立一个 Eloquent 模型开始。模型通常放在 `app/models` 目录下，但是您可以将它们放在任何地方，只要能通过 `composer.json` 被自动载入。

定义一个 **Eloquent** 模型

```
class User extends Eloquent {}
```

注意我们并没有告诉 Eloquent `User` 模型会使用哪个数据库表。若没有特别指定，系统会默认自动对应名称为「类名称的小写复数形态」的数据库表。所以，在上面的例子中，Eloquent 会假设 `User` 将把数据存在 `users` 数据库表。可以在类里定义 `table` 属性自定义要对应的数据库表。

```
class User extends Eloquent {

    protected $table = 'my_users';

}
```

> 注意：Eloquent 也会假设每个数据库表都有一个字段名称为 `id` 的主键。您可以在类里定义 `primaryKey` 属性来重写。同样的，您也可以定义 `connection` 属性，指定模型连接到专属的数据库连接。

定义好模型之后，您就可以从数据库表新增及获取数据了。注意在默认情况下，在数据库表里需要有 `updated_at` 和 `created_at` 两个字段。如果您不想设定或自动更新这两个字段，则将类里的 `$timestamps` 属性设为 `false`即可。

取出所有模型数据

```
$users = User::all();
```

根据主键取出一条数据

```
$user = User::find(1);

var_dump($user->name);
```

> 提示：所有查询构造器里的方法，查询 Eloquent 模型时也可以使用。

根据主键取出一条数据或抛出异常

有时，您可能想要在找不到模型数据时抛出异常，以捕捉异常并让 `App::error` 处理并显示 404 页面。

```
$model = User::findOrFail(1);

$model = User::where('votes', '>', 100)->firstOrFail();
```

要注册错误处理，可以监听 `ModelNotFoundException`

```
use Illuminate\Database\Eloquent\ModelNotFoundException;

App::error(function(ModelNotFoundException $e)
{
    return Response::make('Not Found', 404);
```

```
});
```

Eloquent 模型结合查询语法

```
$users = User::where('votes', '>', 100)->take(10)->get();

foreach ($users as $user)
{
    var_dump($user->name);
}
```

Eloquent 聚合查询

当然，您也可以使用查询构造器的聚合查询方法。

```
$count = User::where('votes', '>', 100)->count();
```

如果没办法使用流畅的接口产生出查询语句，也可以使用 `whereRaw`：

```
$users = User::whereRaw('age > ? and votes = 100', array(25))->get();
```

拆分查询

如果您要处理非常多（数千条）Eloquent 查询结果，使用 `chunk` 方法可以让您顺利工作而不会吃掉内存：

```
User::chunk(200, function($users)
{
    foreach ($users as $user)
    {
        //
    }
});
```

传到方法里的第一个参数表示每次「拆分」要取出的数据数量。第二个参数的闭合函数会在每次取出数据时被调用。

指定查询时连接数据库

您也可以指定在执行 Eloquent 查询时要使用哪个数据库连线。只要使用 `on` 方法：

```
$user = User::on('connection-name')->find(1);
```

Mass Assignment

在建立一个新的模型时，您把属性以数组的方式传入 `create` 方法，这些属性值会经由 mass-assignment 存成模型数据。这非常方便，然而，若盲目地将用户输入存到模型时，可能会造成严重

的安全隐患。如果盲目的存入用户输入，用户可以随意的修改任何以及所有模型的属性。基于这个理由，所有 Eloquent 模型默认会防止 mass-assignment 。

在模型里设定 `fillable` 或 `guarded` 属性作为开始。

定义模型 **Fillable** 属性

`fillable` 属性指定了哪些字段支持 mass-assignable 。可以设定在类里或是建立实例后设定。

```
class User extends Eloquent {

    protected $fillable = array('first_name', 'last_name', 'email');

}
```

在上面的例子里，只有三个属性 mass-assignable 。

定义模型 **Guarded** 属性

`guarded` 与 `fillable` 相反，是作为「黑名单」而不是「白名单」：

```
class User extends Eloquent {

    protected $guarded = array('id', 'password');

}
```

> 注意：使用 `guarded` 时，`Input::get()` 或任何用户可以控制的未过滤数据，永远不应该传入 `save` 或 `update` 方法，因为没有在「黑名单」内的字段可能被更新。

阻挡所有属性被 **Mass Assignment**

上面的例子中，`id` 和 `password` 属性不会被 mass assigned，而所有其他的属性则是 mass assignable。您也可以使用 guard 属性阻止所有属性被 mass assignment ：

```
protected $guarded = array('*');
```

新增，更新，删除

要从模型新增一条数据到数据库，只要建立一个模型实例并调用 `save` 方法即可。

储存新的模型数据

```
$user = new User;

$user->name = 'John';
```

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

> 注意：通常 Eloquent 模型主键值会自动递增。但是您若想自定义主键，将 `incrementing` 属性设成 `false` 。

也可以使用 `create` 方法存入新的模型数据，新增完后会回传新增的模型实例。但是在新增前，需要先在模型类里设定好 `fillable` 或 `guarded` 属性，因为 Eloquent 默认会防止 mass-assignment 。

在新模型数据被储存或新增后，若模型有自动递增主键，可以从对象取得 `id` 属性值：

```
$insertedId = $user->id;
```

在模型里设定 **Guarded** 属性

```
class User extends Eloquent {

    protected $guarded = array('id', 'account_id');

}
```

使用模型的 **Create** 方法

```
// 在数据库建立一条新的用户...
$user = User::create(array('name' => 'John'));

// 以属性找用户，若没有则新增并取得新的实例...
$user = User::firstOrCreate(array('name' => 'John'));

// 以属性找用户，若没有则建立新的实例...
$user = User::firstOrNew(array('name' => 'John'));
```

更新取出的模型

要更新模型，可以取出它，更改属性值，然后使用 `save` 方法：

```
$user = User::find(1);

$user->email = 'john@foo.com';

$user->save();
```

储存模型和关联数据

有时您可能不只想要储存模型本身，也想要储存关联的数据。您可以使用 `push` 方法达到目的：

```
$user->push();
```

您可以结合查询语句，批次更新模型：

```
$affectedRows = User::where('votes', '>', 100)->update(array('status' => 2));
```

> 注意：若使用 Eloquent 查询构造器批次更新模型，则不会触发`模型事件`。

删除模型

要删除模型，只要使用实例调用 `delete` 方法：

```
$user = User::find(1);

$user->delete();
```

依主键值删除模型

```
User::destroy(1);

User::destroy(array(1, 2, 3));

User::destroy(1, 2, 3);
```

当然，您也可以结合查询语句批次删除模型：

```
$affectedRows = User::where('votes', '>', 100)->delete();
```

只更新模型的时间戳

如果您只想要更新模型的时间戳，您可以使用 `touch` 方法：

```
$user->touch();
```

软删除

通过软删除方式删除了一个模型后，模型中的数据并不是真的从数据库被移除。而是会设定 `deleted_at` 时间戳。要让模型使用软删除功能，只要在模型类里加入 `SoftDeletingTrait` 即可：

```
use Illuminate\Database\Eloquent\SoftDeletingTrait;

class User extends Eloquent {

    use SoftDeletingTrait;

    protected $dates = ['deleted_at'];

}
```

要加入 `deleted_at` 字段到数据库表，可以在迁移文件里使用 `softDeletes` 方法：

```
$table->softDeletes();
```

现在当您使用模型调用 `delete` 方法时，`deleted_at` 字段会被更新成现在的时间戳。在查询使用软删除功能的模型时，被「删除」的模型数据不会出现在查询结果里。

强制查询软删除数据

要强制让已被软删除的模型数据出现在查询结果里，在查询时使用 `withTrashed` 方法：

```
$users = User::withTrashed()->where('account_id', 1)->get();
```

`withTrashed` 也可以用在关联查询：

```
$user->posts()->withTrashed()->get();
```

如果您只想查询被软删除的模型数据，可以使用 `onlyTrashed` 方法：

```
$users = User::onlyTrashed()->where('account_id', 1)->get();
```

要把被软删除的模型数据恢复，使用 `restore` 方法：

```
$user->restore();
```

您也可以结合查询语句使用 `restore` ：

```
User::withTrashed()->where('account_id', 1)->restore();
```

如同 `withTrashed` ， `restore` 方法也可以用在关联对象：

```
$user->posts()->restore();
```

如果想要真的从模型数据库删除，使用 `forceDelete` 方法：

```
$user->forceDelete();
```

`forceDelete` 方法也可以用在关联对象：

```
$user->posts()->forceDelete();
```

要确认模型是否被软删除了，可以使用 `trashed` 方法：

```
if ($user->trashed())
{
    //
}
```

时间戳

默认 Eloquent 会自动维护数据库表的 `created_at` 和 `updated_at` 字段。只要把这两个「时间戳」字段加到数据库表，Eloquent 就会处理剩下的工作。如果不想让 Eloquent 自动维护这些字段，把下面的属性加到模型类里：

关闭自动更新时间戳

```
class User extends Eloquent {

    protected $table = 'users';

    public $timestamps = false;

}
```

自定义时间戳格式

如果想要自定义时间戳格式，可以在模型类里重写 `getDateFormat` 方法：

```
class User extends Eloquent {

    protected function getDateFormat()
    {
        return 'U';
    }

}
```

范围查询

定义范围查询

范围查询可以让您轻松的重复利用模型的查询逻辑。要设定范围查询，只要定义有 `scope` 前缀的模型方法：

```
class User extends Eloquent {

    public function scopePopular($query)
    {
        return $query->where('votes', '>', 100);
    }

    public function scopeWomen($query)
    {
        return $query->whereGender('W');
    }

}
```

使用范围查询

```
$users = User::popular()->women()->orderBy('created_at')->get();
```

动态范围查询

有时您可能想要定义可接受参数的范围查询方法。只要把参数加到方法里：

```
class User extends Eloquent {

    public function scopeOfType($query, $type)
    {
        return $query->whereType($type);
    }

}
```

然后把参数值传到范围查询方法调用里：

```
$users = User::ofType('member')->get();
```

关联

当然，您的数据库表很可能跟另一张表相关联。例如，一篇 blog 文章可能有很多评论，或是一张订单跟下单客户相关联。Eloquent 让管理和处理这些关联变得很容易。Laravel 有很多种关联种类：

- 一对一
- 一对多
- 多对多
- 远层一对多关联
- 多态关联
- 多态的多对多关联

一对一

定义一对一关联

一对一关联是很基本的关联。例如一个 `User` 模型会对应到一个 `Phone` 。在 Eloquent 里可以像下面这样定义关联：

```
class User extends Eloquent {

    public function phone()
    {
        return $this->hasOne('Phone');
    }

}
```

传到 `hasOne` 方法里的第一个参数是关联模型的类名称。定义好关联之后，就可以使用 Eloquent 的动态属性取得关联对象：

```
$phone = User::find(1)->phone;
```

SQL 会执行如下语句：

```
select * from users where id = 1

select * from phones where user_id = 1
```

注意，Eloquent 假设对应的关联模型数据库表里，外键名称是基于模型名称。在这个例子里，默认 `Phone` 模型数据库表会以 `user_id` 作为外键。如果想要更改这个默认，可以传入第二个参数到 `hasOne` 方法里。更进一步，您可以传入第三个参数，指定关联的外键要对应到本身的哪个字段：

```
return $this->hasOne('Phone', 'foreign_key');

return $this->hasOne('Phone', 'foreign_key', 'local_key');
```

定义相对的关联

要在 `Phone` 模型里定义相对的关联，可以使用 `belongsTo` 方法：

```
class Phone extends Eloquent {

    public function user()
    {
        return $this->belongsTo('User');
    }

}
```

在上面的例子里，Eloquent 默认会使用 `phones` 数据库表的 `user_id` 字段查询关联。如果想要自己指定外键字段，可以在 `belongsTo` 方法里传入第二个参数：

```
class Phone extends Eloquent {

    public function user()
    {
        return $this->belongsTo('User', 'local_key');
    }

}
```

除此之外，也可以传入第三个参数指定要参照上层数据库表的哪个字段：

```
class Phone extends Eloquent {

    public function user()
```

```
    {
        return $this->belongsTo('User', 'local_key', 'parent_key');
    }

}
```

一对多

一对多关联的例子如，一篇 Blog 文章可能「有很多」评论。可以像这样定义关联：

```
class Post extends Eloquent {

    public function comments()
    {
        return $this->hasMany('Comment');
    }

}
```

现在可以经由动态属性取得文章的评论：

```
$comments = Post::find(1)->comments;
```

如果需要增加更多条件限制，可以在调用 `comments` 方法后面串接查询条件方法：

```
$comments = Post::find(1)->comments()->where('title', '=', 'foo')->first();
```

同样的，您可以传入第二个参数到 `hasMany` 方法更改默认的外键名称。以及，如同 `hasOne` 关联，可以指定本身的对应字段：

```
return $this->hasMany('Comment', 'foreign_key');

return $this->hasMany('Comment', 'foreign_key', 'local_key');
```

定义相对的关联

要在 `Comment` 模型定义相对应的关联，可使用 `belongsTo` 方法：

```
class Comment extends Eloquent {

    public function post()
    {
        return $this->belongsTo('Post');
    }

}
```

多对多

多对多关联更为复杂。这种关联的例子如，一个用户（ user ）可能用有很多身份（ role ），而一种身

份可能很多用户都有。例如很多用户都是「管理者」。多对多关联需要用到三个数据库表：`users`，`roles`，和 `role_user`。`role_user` 枢纽表命名是以相关联的两个模型数据库表，依照字母顺序命名，枢纽表里面应该要有 `user_id` 和 `role_id` 字段。

可以使用 `belongsToMany` 方法定义多对多关系：

```
class User extends Eloquent {

    public function roles()
    {
        return $this->belongsToMany('Role');
    }

}
```

现在我们可以从 `User` 模型取得 roles：

```
$roles = User::find(1)->roles;
```

如果不想使用默认的枢纽数据库表命名方式，可以传递数据库表名称作为 `belongsToMany` 方法的第二个参数：

```
return $this->belongsToMany('Role', 'user_roles');
```

也可以更改默认的关联字段名称：

```
return $this->belongsToMany('Role', 'user_roles', 'user_id', 'foo_id');
```

当然，也可以在 `Role` 模型定义相对的关联：

```
class Role extends Eloquent {

    public function users()
    {
        return $this->belongsToMany('User');
    }

}
```

远层一对多关联

「远层一对多关联」提供了方便简短的方法，可以经由多层间的关联取得远层的关联。例如，一个 `Country` 模型可能通过 `User` 关联到很多 `Post` 模型。数据库表间的关系可能看起来如下：

```
countries
    id - integer
    name - string

users
```

```
    id - integer
    country_id - integer
    name - string

posts
    id - integer
    user_id - integer
    title - string
```

虽然 `posts` 数据库表本身没有 `country_id` 字段，但 `hasManyThrough` 方法让我们可以使用 `$country->posts` 取得 country 的 posts。我们可以定义以下关联：

```
class Country extends Eloquent {

    public function posts()
    {
        return $this->hasManyThrough('Post', 'User');
    }

}
```

如果想要手动指定关联的字段名称，可以传入第三和第四个参数到方法里：

```
class Country extends Eloquent {

    public function posts()
    {
        return $this->hasManyThrough('Post', 'User', 'country_id', 'user_id');
    }

}
```

多态关联

多态关联可以用一个简单的关联方法，就让一个模型同时关联多个模型。例如，您可能想让 photo 模型同时和一个 staff 或 order 模型关联。可以定义关联如下：

```
class Photo extends Eloquent {

    public function imageable()
    {
        return $this->morphTo();
    }

}

class Staff extends Eloquent {

    public function photos()
    {
        return $this->morphMany('Photo', 'imageable');
    }

}
```

```
class Order extends Eloquent {

    public function photos()
    {
        return $this->morphMany('Photo', 'imageable');
    }

}
```

取得多态关联对象

现在我们可以从 staff 或 order 模型取得多态关联对象：

```
$staff = Staff::find(1);

foreach ($staff->photos as $photo)
{
    //
}
```

取得多态关联对象的拥有者

然而，多态关联真正神奇的地方，在于要从 `Photo` 模型取得 staff 或 order 对象时：

```
$photo = Photo::find(1);

$imageable = $photo->imageable;
```

`Photo` 模型里的 `imageable` 关联会回传 `Staff` 或 `Order` 实例，取决于这是哪一种模型拥有的照片。

多态关联的数据库表结构

为了理解多态关联的运作机制，来看看它们的数据库表结构：

```
staff
    id - integer
    name - string

orders
    id - integer
    price - integer

photos
    id - integer
    path - string
    imageable_id - integer
    imageable_type - string
```

要注意的重点是 `photos` 数据库表的 `imageable_id` 和 `imageable_type`。在上面的例子里，ID 字段会包含 staff 或 order 的 ID，而 type 是拥有者的模型类名称。这就是让 ORM 在取得 `imageable` 关

联对象时，决定要哪一种模型对象的机制。

多态的多对多关联

多态的多对多关联数据库表结构

除了一般的多态关联，也可以使用多对多的多态关联。例如，Blog 的 `Post` 和 `Video` 模型可以共用多态的 `Tag` 关联模型。首先，来看看数据库表结构：

```
posts
    id - integer
    name - string

videos
    id - integer
    name - string

tags
    id - integer
    name - string

taggables
    tag_id - integer
    taggable_id - integer
    taggable_type - string
```

现在，我们准备好设定模型关联了。`Post` 和 `Video` 模型都可以经由 `tags` 方法建立 `morphToMany` 关联：

```
class Post extends Eloquent {

    public function tags()
    {
        return $this->morphToMany('Tag', 'taggable');
    }

}
```

在 `Tag` 模型里针对每一种关联建立一个方法：

```
class Tag extends Eloquent {

    public function posts()
    {
        return $this->morphedByMany('Post', 'taggable');
    }

    public function videos()
    {
        return $this->morphedByMany('Video', 'taggable');
    }

}
```

关联查询

根据关联条件查询

在取得模型数据时，您可能想要以关联模型作为查询限制。例如，您可能想要取得所有「至少有一篇评论」的Blog 文章。可以使用 `has` 方法达成目的：

```
$posts = Post::has('comments')->get();
```

也可以指定运算子和数量：

```
$posts = Post::has('comments', '>=', 3)->get();
```

如果想要更进阶，可以使用 `whereHas` 和 `orWhereHas` 方法，在 `has` 查询里设置 "where" 条件 ：

```
$posts = Post::whereHas('comments', function($q)
{
    $q->where('content', 'like', 'foo%');

})->get();
```

动态属性

Eloquent 可以经由动态属性取得关联对象。Eloquent 会自动进行关联查询，而且会很聪明的知道应该要使用 `get`（用在一对多关联）或是 `first`（用在一对一关联）方法。可以经由和「关联方法名称相同」的动态属性取得对象。例如，如下面的模型对象 `$phone`：

```
class Phone extends Eloquent {

    public function user()
    {
        return $this->belongsTo('User');
    }

}

$phone = Phone::find(1);
```

或是像下面这样打印用户的 email ：

```
echo $phone->user()->first()->email;
```

可以简写如下：

```
echo $phone->user->email;
```

> 注意：若取得的是许多关联对象，会返回 `Illuminate\Database\Eloquent\Collection` 对象：

预载入

预载入是用来减少 N + 1 查询问题。例如，一个 `Book` 模型数据会关联到一个 `Author` 。关联会像下面这样定义：

```
class Book extends Eloquent {

    public function author()
    {
        return $this->belongsTo('Author');
    }

}
```

现在考虑下面的代码：

```
foreach (Book::all() as $book)
{
    echo $book->author->name;
}
```

上面的循环会执行一次查询取回所有数据库表上的书籍，然而每本书籍都会执行一次查询取得作者。所以若我们有 25 本书，就会进行 26次查询。

很幸运地，我们可以使用预载入大量减少查询次数。使用 `with` 方法指定想要预载入的关联对象：

```
foreach (Book::with('author')->get() as $book)
{
    echo $book->author->name;
}
```

现在，上面的循环总共只会执行两次查询：

```
select * from books

select * from authors where id in (1, 2, 3, 4, 5, ...)
```

使用预载入可以大大提高程序的性能。

当然，也可以同时载入多种关联：

```
$books = Book::with('author', 'publisher')->get();
```

甚至可以预载入巢状关联：

```
$books = Book::with('author.contacts')->get();
```

上面的例子中，`author` 关联会被预载入，author 的 `contacts` 关联也会被预载入。

预载入条件限制

有时您可能想要预载入关联，同时也想要指定载入时的查询限制。下面有一个例子：

```
$users = User::with(array('posts' => function($query)
{
    $query->where('title', 'like', '%first%');

}))->get();
```

上面的例子里，我们预载入了 user 的 posts 关联，并限制条件为 post 的 title 字段需包含 "first" 。

当然，预载入的闭合函数里不一定只能加上条件限制，也可以加上排序：

```
$users = User::with(array('posts' => function($query)
{
    $query->orderBy('created_at', 'desc');

}))->get();
```

延迟预载入

也可以直接从模型的 collection 预载入关联对象。这对于需要根据情况决定是否载入关联对象时，或是跟缓存一起使用时很有用。

```
$books = Book::all();

$books->load('author', 'publisher');
```

新增关联模型

附加一个关联模型

您常常会需要加入新的关联模型。例如新增一个 comment 到 post 。除了手动设定模型的 `post_id` 外键，也可以从上层的 `Post` 模型新增关联的 comment ：

```
$comment = new Comment(array('message' => 'A new comment.'));

$post = Post::find(1);

$comment = $post->comments()->save($comment);
```

上面的例子里，新增的 comment `post_id` 字段会被自动设定。

如果想要同时新增很多关联模型：

```
$comments = array(
    new Comment(array('message' => 'A new comment.')),
    new Comment(array('message' => 'Another comment.')),
```

```
    new Comment(array('message' => 'The latest comment.'))
);

$post = Post::find(1);

$post->comments()->saveMany($comments);
```

从属关联模型 (Belongs To)

要更新 `belongsTo` 关联时, 可以使用 `associate` 方法。这个方法会设定子模型的外键：

```
$account = Account::find(10);

$user->account()->associate($account);

$user->save();
```

新增多对多关联模型 (Many To Many)

您也可以新增多对多的关联模型。让我们继续使用 `User` 和 `Role` 模型作为例子。我们可以使用 `attach` 方法简单地把 roles 附加给一个 user：

附加多对多模型

```
$user = User::find(1);

$user->roles()->attach(1);
```

也可以传入要存在枢纽表中的属性数组：

```
$user->roles()->attach(1, array('expires' => $expires));
```

当然, 有 `attach` 就会有相反的 `detach`：

```
$user->roles()->detach(1);
```

使用 **Sync** 方法同时附加一个以上多对多关联

您也可以使用 `sync` 方法附加关联模型。`sync` 方法会把根据 ID 数组把关联存到枢纽表。附加完关联后, 枢纽表里的模型只会关联到 ID 数组里的 id ：

```
$user->roles()->sync(array(1, 2, 3));
```

Sync 时在枢纽表加入额外数据

也可以在把每个 ID 加入枢纽表时, 加入其他字段的数据：

```
$user->roles()->sync(array(1 => array('expires' => true)));
```

有时您可能想要使用一个命令，在建立新模型数据的同时附加关联。可以使用 `save` 方法达成目的：

```
$role = new Role(array('name' => 'Editor'));

User::find(1)->roles()->save($role);
```

上面的例子里，新的 `Role` 模型对象会在储存的同时关联到 user 模型。也可以传入属性数组把数据加到关联数据库表：

```
User::find(1)->roles()->save($role, array('expires' => $expires));
```

更新上层时间戳

当模型 `belongsTo` 另一个模型，比方说一个 `Comment` 属于一个 `Post`，如果能在子模型被更新时，更新上层的时间戳，这将会很有用。例如，当 `Comment` 模型更新时，您可能想要能够同时自动更新 `Post` 的 `updated_at` 时间戳。Eloquent 让事情变得很简单。只要在子关联的类里，把关联方法名称加入 `touches` 属性即可：

```
class Comment extends Eloquent {

    protected $touches = array('post');

    public function post()
    {
        return $this->belongsTo('Post');
    }

}
```

现在，当您更新 `Comment` 时，对应的 `Post` 会自动更新 `updated_at` 字段：

```
$comment = Comment::find(1);

$comment->text = 'Edit to this comment!';

$comment->save();
```

使用枢纽表

如您所知，要操作多对多关联需要一个中间的数据库表。Eloquent 提供了一些有用的方法可以和这张表互动。例如，假设 `User` 对象关联到很多 `Role` 对象。取出这些关联对象时，我们可以在关联模型上取得 `pivot` 数据库表的数据：

```
$user = User::find(1);
```

```
foreach ($user->roles as $role)
{
    echo $role->pivot->created_at;
}
```

注意我们取出的每个 `Role` 模型对象会自动给一个 `pivot` 属性。这属性包含了枢纽表的模型数据，可以像一般的 Eloquent 模型一样使用。

默认 `pivot` 对象只会有关联键的属性。如果您想让 `pivot` 可以包含其他枢纽表的字段，可以在定义关联方法时指定那些字段：

```
return $this->belongsToMany('Role')->withPivot('foo', 'bar');
```

现在可以在 `Role` 模型的 `pivot` 对象上取得 `foo` 和 `bar` 属性了。

如果您想要可以自动维护枢纽表的 `created_at` 和 `updated_at` 时间戳，在定义关联方法时加上 `withTimestamps` 方法：

```
return $this->belongsToMany('Role')->withTimestamps();
```

删除枢纽表的关联数据

要删除模型在枢纽表的所有关联数据，可以使用 `detach` 方法：

```
User::find(1)->roles()->detach();
```

`attach` 和 `detach` 都要求 ID 数组作为输入：

```
$user = User::find(1);

$user->roles()->detach([1, 2, 3]);

$user->roles()->attach([1 => ['attribute1' => 'value1'], 2, 3]);
```

更新枢纽表的数据

有时您只想更新枢纽表的数据，而没有要移除关联。如果您想更新枢纽表，可以像下面的例子使用 `updateExistingPivot` 方法：

```
User::find(1)->roles()->updateExistingPivot($roleId, $attributes);
```

自定义枢纽模型

Laravel 允许您自定义枢纽模型。要自定义模型，首先要建立一个继承 `Eloquent` 的「基本」模型类。在其他的 Eloquent 模型继承这个自定义的基本类，而不是默认的 `Eloquent` 。在基本模型类里，加

入下面的方法回传自定义的枢纽模型实例：

```
public function newPivot(Model $parent, array $attributes, $table, $exists)
{
    return new YourCustomPivot($parent, $attributes, $table, $exists);
}
```

Collections

所有 Eloquent 查询回传的数据，如果结果多于一条，不管是经由 `get` 方法或是 `relationship`，都会转换成 collection 对象回传。这个对象实现了 `IteratorAggregate` PHP 接口，所以可以像数组一般进行遍历。而 Collections 对象本身还拥有很多有用的方法可以操作模型数据。

确认 **Collection** 里是否包含特定键值

例如，我们可以使用 `contains` 方法，确认结果数据中，是否包含主键为特定值的对象。

```
$roles = User::find(1)->roles;

if ($roles->contains(2))
{
    //
}
```

Collection 也可以转换成数组或 JSON：

```
$roles = User::find(1)->roles->toArray();

$roles = User::find(1)->roles->toJson();
```

如果 collection 被类型转换成字串，会回传 JSON 格式：

```
$roles = (string) User::find(1)->roles;
```

Collections 遍历

Eloquent collections 里包含了一些有用的方法可以进行循环或是进行过滤：

```
$roles = $user->roles->each(function($role)
{
    //
});
```

Collection 过滤

过滤 collection 时，回调函数的使用方式和 array_filter 里一样。

```
$users = $users->filter(function($user)
```

```
{
    return $user->isAdmin();
});
```

> 注意：如果要在过滤 collection 之后转成 JSON，转换之前先调用 `values` 方法重设数组的键值。

遍历传入 **Collection** 里的每个对象到回调函数

```
$roles = User::find(1)->roles;

$roles->each(function($role)
{
    //
});
```

依照属性值排序

```
$roles = $roles->sortBy(function($role)
{
    return $role->created_at;
});
```

依照属性值排序

```
$roles = $roles->sortBy('created_at');
```

回传自定义的集合对象

有时您可能想要回传自定义的集合对象，让您可以在集合类里加入想要的方法。可以在 Eloquent 模型类里重写 `newCollection` 方法：

```
class User extends Eloquent {

    public function newCollection(array $models = array())
    {
        return new CustomCollection($models);
    }

}
```

获取器和修改器

定义获取器

Eloquent 提供了便利的方法，可以在取得或设定属性时进行转换。要定义获取器，只要在模型里加入类似 `getFooAttribute` 的方法。注意方法名称应该使用驼峰式大小写命名，而对应的 database

字段名称是底线分隔小写命名：

```
class User extends Eloquent {

    public function getFirstNameAttribute($value)
    {
        return ucfirst($value);
    }

}
```

上面的例子中，`first_name` 字段设定了一个获取器。注意传入方法的参数是原本的字段数据。

定义修改器

修改器的定义方式很雷同：

```
class User extends Eloquent {

    public function setFirstNameAttribute($value)
    {
        $this->attributes['first_name'] = strtolower($value);
    }

}
```

日期转换器

默认 Eloquent 会把 `created_at` 和 `updated_at` 字段属性转换成 Carbon 实例，它提供了很多有用的方法，并继承了 PHP 原生的 `DateTime` 类。

您可以经由重写模型的 `getDates` 方法，自定义哪个字段可以被自动转换，或甚至完全不要让日期转换成 Carbon：

```
public function getDates()
{
    return array('created_at');
}
```

当字段是表示日期的时候，可以将值设为 UNIX timestamp 、日期字串（ `Y-m-d` ）、日期时间（ date-time ）字串，当然还有 `DateTime` 或 `Carbon` 实例。

要完全关闭日期转换功能，只要从 `getDates` 方法回传空数组即可：

```
public function getDates()
{
    return array();
}
```

Model Events

Eloquent 模型有很多事件可以驱动，让您可以在模型操作的生命周期中不同时间点，使用下列方法绑定事件：`creating`、`created`、`updating`、`updated`、`saving`、`saved`、`deleting`、`deleted`、`restoring`、`restored`。

当一个对象初次被储存到数据库，`creating` 和 `created` 事件会被驱动。如果不是新对象而调用了 `save` 方法，`updating` / `updated` 事件会被驱动。而两者的 `saving` / `saved` 事件都会被驱动。

使用事件取消数据库操作

如果 `creating`、`updating`、`saving`、`deleting` 事件回传 `false` 的话，就会取消数据库操作：

```
User::creating(function($user)
{
    if ( ! $user->isValid()) return false;
});
```

设定模型 **Boot** 方法

Eloquent 模型有静态的 `boot` 方法，可以使用它方便的注册事件绑定。

```
class User extends Eloquent {

    public static function boot()
    {
        parent::boot();

        // Setup event bindings...
    }

}
```

模型观察者

要整合模型的事件处理，可以注册一个模型观察者。观察者类里要设定对应模型事件的方法。例如，观察者类里可能有 `creating`、`updating`、`saving` 方法，还有其他对应模型事件名称的方法：

例如，一个模型观察者类可能看起来如下：

```
class UserObserver {

    public function saving($model)
    {
        //
    }

    public function saved($model)
    {
```

```
        //
    }

}
```

可以使用 `observe` 方法注册一个观察者实例：

```
User::observe(new UserObserver);
```

转换成数组 / JSON

将模型数据转成数组

建立 JSON API 时，您可能常常需要把模型和关联对象转换成数组或 JSON 。所以 Eloquent 里已经包含了这些方法。要把模型和已载入的关联对象转成数组，可以使用 `toArray` 方法：

```
$user = User::with('roles')->first();

return $user->toArray();
```

记得也可以把模型 collection 转换成数组：

```
return User::all()->toArray();
```

把模型转换成 **JSON**

要把模型转换成 JSON，可以使用 `toJson` 方法：

```
return User::find(1)->toJson();
```

从路由回传模型

注意当模型或 collection 被类型转换成字符串时会自动转换成 JSON 格式，这意味着您可以直接从路由回传 Eloquent 对象！

```
Route::get('users', function()
{
    return User::all();
});
```

转换成数组或 **JSON** 时隐藏属性

有时您可能想要限制能出现在数组或 JSON 格式的属性数据，比如密码。只要在模型里增加 `hidden` 属性即可：

```
class User extends Eloquent {
```

```
    protected $hidden = array('password');

}
```

> 注意：要隐藏关联数据，要使用关联的方法名称，而不是动态获取的属性名称。

此外，可以使用 `visible` 属性定义白名单：

```
protected $visible = array('first_name', 'last_name');
```

有时候您可能想要增加不存在数据库字段的属性数据。这时候只要定义一个获取器即可：

```
public function getIsAdminAttribute()
{
    return $this->attributes['admin'] == 'yes';
}
```

定义好获取器之后，再把对应的属性名称加到模型里的 `appends` 属性：

```
protected $appends = array('is_admin');
```

把属性加到 `appends` 数组之后，在模型数据转换成数组或 JSON 格式时就会有对应的值。

结构生成器

介绍

Laravel 的 `结构生成器` 提供一个数据库无关的操作数据库表方法（即不关心应用将使用何种数据库，只考虑数据表架构操作），它可以让 Laravel 很好的支持各种数据库类型，并且能在不同系统间提供一致性的 API 操作。

建立与删除数据库表

要建立一个新的数据库表，可使用 `Schema::create` 方法：

```
Schema::create('users', function($table)
{
    $table->increments('id');
});
```

传入 `create` 方法的第一个参数是数据库表名称，第二个参数是 `Closure` 并接收 `Blueprint` 对象被用来定义新的数据库表。

要修改数据库表名称，可使用 `rename` 方法：

```
Schema::rename($from, $to);
```

要指定特定连接来操作，可使用 `Schema::connection` 方法：

```
Schema::connection('foo')->create('users', function($table)
{
    $table->increments('id');
});
```

要移除数据库表, 可使用 `Schema::drop` 方法:

```
Schema::drop('users');

Schema::dropIfExists('users');
```

加入字段

更新现有的数据库表, 可使用 `Schema::table` 方法:

```
Schema::table('users', function($table)
{
    $table->string('email');
});
```

数据库表产生器提供多种字段类型可使用, 在您建立数据库表时也许会用到:

命令	功能描述
`$table->bigIncrements('id');`	ID 自动增量, 使用相当于 "big integer" 类型.
`$table->bigInteger('votes');`	相当于 BIGINT 类型
`$table->binary('data');`	相当于 BLOB 类型
`$table->boolean('confirmed');`	相当于 BOOLEAN 类型
`$table->char('name', 4);`	相当于 CHAR 类型
`$table->date('created_at');`	相当于 DATE 类型
`$table->dateTime('created_at');`	相当于 DATETIME 类型
`$table->decimal('amount', 5, 2);`	相当于 DECIMAL 类型, 并带有精度与尺度
`$table->double('column', 15, 8);`	相当于 DOUBLE 类型
`$table->enum('choices', array('foo', 'bar'));`	相当于 ENUM 类型
`$table->float('amount');`	相当于 FLOAT 类型
`$table->increments('id');`	相当于 Incrementing 类型(数据库表主键)
`$table->integer('votes');`	相当于 INTEGER 类型
`$table->longText('description');`	相当于 LONGTEXT 类型
`$table->mediumInteger('numbers');`	相当于 MEDIUMINT 类型
`$table->mediumText('description');`	相当于 MEDIUMTEXT 类型

`$table->morphs('taggable');`	加入整数 `taggable_id` 与字串 `taggable_type`
`$table->nullableTimestamps();`	与 `timestamps()` 相同，但允许 NULL
`$table->smallInteger('votes');`	相当于 SMALLINT 类型
`$table->tinyInteger('numbers');`	相当于 TINYINT 类型
`$table->softDeletes();`	加入 **deleted_at** 字段于软删除使用
`$table->string('email');`	相当于 VARCHAR 类型 equivalent column
`$table->string('name', 100);`	相当于 VARCHAR 类型，并指定长度
`$table->text('description');`	相当于 TEXT 类型
`$table->time('sunrise');`	相当于 TIME 类型
`$table->timestamp('added_on');`	相当于 TIMESTAMP 类型
`$table->timestamps();`	加入 **created_at** 和 **updated_at** 字段
`$table->rememberToken();`	加入 `remember_token` 使用 VARCHAR(100) NULL
`->nullable()`	标示此字段允许 NULL
`->default($value)`	定义此字段的默认值
`->unsigned()`	设定整数是无分正负

在 **MySQL** 使用 **After** 方法

若您使用 MySQL 数据库，您可以使用 `after` 方法来指定字段的顺序:

```
$table->string('name')->after('email');
```

修改字段名称

要修改字段名称，可在结构生成器内使用 `renameColumn` 方法，请确认在修改前 `composer.json` 文件内已经加入 `doctrine/dbal`。

```
Schema::table('users', function($table)
{
    $table->renameColumn('from', 'to');
});
```

> 注意: `enum` 字段类型不支持修改字段名称

移除字段

要移除字段，可在结构生成器内使用 `dropColumn` 方法，请确认在移除前 `composer.json` 文件内已经加入 `doctrine/dbal`。

移除数据库表字段

```
Schema::table('users', function($table)
{
    $table->dropColumn('votes');
});
```

移除数据库表多个字段

```
Schema::table('users', function($table)
{
    $table->dropColumn(array('votes', 'avatar', 'location'));
});
```

检查是否存在

检查数据库表是否存在

您可以轻松的检查数据库表或字段是否存在，使用 `hasTable` 和 `hasColumn` 方法:

```
if (Schema::hasTable('users'))
{
    //
}
```

检查字段是否存在

```
if (Schema::hasColumn('users', 'email'))
{
    //
}
```

加入索引

结构生成器支持多种索引类型，有两种方法可以加入，方法一，您可以在定义字段时顺道附加上去，或者是分开另外加入:

```
$table->string('email')->unique();
```

或者，您可以独立一行来加入索引，以下是支持的索引类型:

命令	功能描述

`$table->primary('id');`	加入主键
`$table->primary(array('first', 'last'));`	加入复合键
`$table->unique('email');`	加入唯一索引
`$table->index('state');`	加入基本索引

外键

Laravel 也支持数据库表的外键约束:

```
$table->foreign('user_id')->references('id')->on('users');
```

例子中, 我们关注字段 `user_id` 参照到 `users` 数据库表的 `id` 字段。

您也可以指定选择在 "on delete" 和 "on update" 进行约束动作:

```
$table->foreign('user_id')
      ->references('id')->on('users')
      ->onDelete('cascade');
```

要移除外键, 可使用 `dropForeign` 方法。外键的命名约定如同其他索引:

```
$table->dropForeign('posts_user_id_foreign');
```

> 注意: 当外键有参照到自动增量时, 记得设定外键为 `unsigned` 类型。

移除索引

要移除索引您必须指定索引名称, Laravel 默认有脉络可循的索引名称。简单地链接这些数据库表与索引的字段名称和类型。举例如下:

命令	功能描述
`$table->dropPrimary('users_id_primary');`	从 "users" 数据库表移除主键
`$table->dropUnique('users_email_unique');`	从 "users" 数据库表移除唯一索引
`$table->dropIndex('geo_state_index');`	从 "geo" 数据库表移除基本索引

移除时间戳记和软删除

要移除 `timestamps`, `nullableTimestamps` 或 `softDeletes` 字段类型, 您可以使用以下方法:

命令	功能描述

`$table->dropTimestamps();`	移除 **created_at** 和 **updated_at** 字段
`$table->dropSoftDeletes();`	移除 **deleted_at** 字段

储存引擎

要设定数据库表的储存引擎, 可在结构生成器设定 `engine` 属性:

```
Schema::create('users', function($table)
{
    $table->engine = 'InnoDB';

    $table->string('email');
});
```

数据库迁移 & 数据填充

简介

Migrations是一种数据库版本控制功能.它允许团队开发者修改数据库结构,并使其保持最新状态.Migrations通常和结构生成器 配合使用来管理您的应用程序结构.

创建数据迁移

使用 Artisan 命令行的 `migrate:make` 命令创建一个迁移:(在命令行模式下使用)

```
php artisan migrate:make create_users_table
```

所有的迁移都被存放在 `app/database/migrations` 文件夹下,文件以时间戳命名以方便Laravel框架按时间来界定这些文件顺序.

您可以在创建迁移的时候使用 `--path` 选项, 用来指定迁移文件存放的路径.该路径是你安装框架根目录的相对路径:

```
php artisan migrate:make foo --path=app/migrations
```

`--table` 和 `--create` 选项用来指定表名以及是否创建一个新表:

```
php artisan migrate:make add_votes_to_user_table --table=users

php artisan migrate:make create_users_table --create=users
```

运行数据迁移

运行所有迁移(使你的所有表保持最新)

```
php artisan migrate
```

运行某个路径下的所有迁移(指定迁移文件路径)

```
php artisan migrate --path=app/foo/migrations
```

运行某个包下的所有迁移(安装或升级某个扩展包对应数据库时候使用)

```
php artisan migrate --package=vendor/package
```

> 注意: 如果在运行迁移的时候收到一个 "class not found" 的错误, 请尝试运行 `composer dump-autoload` 命令.

在生产环境中强制使用数据迁移

有些迁移操作具有破坏性,会导致你丢失数据库中原有数据.为了防止你运行这样的命令造成不必要的破坏,这些命令运行的时候会询问你是否确定要这样做.如果你想运行这样的命令而不出现提示,可以使用 `--force` 选项:

```
php artisan migrate --force
```

回滚数据迁移(即使回滚,原有数据也被破坏了,只能回滚表结构,所以别拿这个功能当救命稻草)

回滚最后一次迁移

```
php artisan migrate:rollback
```

回滚所有迁移

```
php artisan migrate:reset
```

回滚所有迁移并重新运行数据迁移

```
php artisan migrate:refresh

php artisan migrate:refresh --seed
```

数据库填充

Laravel 可以非常简单的使用数据填充类(seed classes)帮你生成一些测试数据放到数据库中去.所有的数据填充类(seed classes)都存放在 `app/database/seeds` 路径下.数据填充类(seed classes)你可以随便命名,但最好遵循一些合理的约定,例如 `UserTableSeeder` 等. `DatabaseSeeder`是一个已经生成好的默认类(它将默认被执行,你也可以把这个当作例子).在这个类(DatabaseSeeder)中,你可以使用 `call` 方法来运行其他数据填充类,这样你就能控制数据填充的顺序了.

数据库填充类的例子

```
class DatabaseSeeder extends Seeder {

    public function run()
    {
        $this->call('UserTableSeeder');

        $this->command->info('User table seeded!');
    }

}

class UserTableSeeder extends Seeder {

    public function run()
    {
        DB::table('users')->delete();

        User::create(array('email' => 'foo@bar.com'));
    }

}
```

使用 Artisan 命令行的 `db:seed` 命令填充数据库：

```
php artisan db:seed
```

默认情况下 `db:seed` 命令运行的是 `DatabaseSeeder` 类, 这个类中可以(像上面的例子中那样)调用其他seed类. 但是你也可以使用 `--class` 选项来单独运行指定数据库填充类：

```
php artisan db:seed --class=UserTableSeeder
```

您也可以使用 `migrate:refresh` 命令,这将回滚并重新运行所有数据库迁移:

```
php artisan migrate:refresh --seed
```

Redis

简介

Redis 是一个开源的先进键值存储工具(其实是Nosql数据库的一种). 它常用来进行一些数据结构的存储,这些数据结构包括 strings(字符串), hashes(哈希表), lists(双向链表), sets(无序集合), and sorted sets(有序集合).

> 注意: 如果你的Redis PHP扩展是使用PECL安装的,那你需要将 `app/config/app.php` 文件中的Redis别名(alias)改成别的名字.

配置

Redis的配置信息存储在 **app/config/database.php** 文件内. 在这个文件里你可以看见一个 **redis** 数组,它就是你应用中使用Redis的相关配置:

```
'redis' => array(

    'cluster' => true,

    'default' => array('host' => '127.0.0.1', 'port' => 6379),

),
```

这个 `defaule`(默认) 的服务器配置可以在做开发的时候使用. 你也可以任意的按照你的需要和使用环境来进行配置. 你可以简单的给每个Redis服务器起一个名字并指定服务器(host)和端口(port).

上面那个 `cluster`(集群) 选项是告诉Laravel框架Redis的客户端要访问多个节点,允许你在内存中建立一个记录这些节点的节点池. 然而,请注意客户端不会处理故障转移,也就是说缓存的数据可能会从另一个服务器上读取(这个不是特别理解,可能是说一旦一个节点上的服务器挂了,有可能配置了集群这个选项后就能在集群中其他服务器节点上读取到你要的缓存数据了).

如果你的 Redis服务器端使用了用户验证,你需要增加一个 `password` 键值对选项到配置的数组中来配置密码.

使用方法

你可以用 `Redis::connection` 方法来获取一个Redis连接实例:

```
$redis = Redis::connection();
```

这样做可以给你一个默认的Redis服务器连接实例.如果你需要其他服务器连接实例,那么你就要在连接函数中声明你具体需要使用的那个连接配置名称(这个名称就是前面你在配置里除了 default 外自定义的其他连接名称):

```
$redis = Redis::connection('other');
```

一旦你有了一个Redis的客户端连接实例,你就可以发送你的 Redis commands(Redis操作命令) 来操作Redis数据库了.Laravel使用魔术方法来发送命令到Redis服务器(即你可以按照实际需要发送任何Redis能处理的命令过去):

```
$redis->set('name', 'Taylor');

$name = $redis->get('name');

$values = $redis->lrange('names', 5, 10);
```

注意命令对应的参数可以简单的放在魔术方法调用的函数括号里面发送.当然,你可能不喜欢使用魔术方法来调用那些命令,你也可以通过 `command` 方法向服务器发送命令:

```
$values = $redis->command('lrange', array(5, 10));
```

当你只是想简单执行一下default(默认)连接的命令时,你可以使用 `Redis` 类的静态魔术方法:

```
Redis::set('name', 'Taylor');

$name = Redis::get('name');

$values = Redis::lrange('names', 5, 10);
```

> 注意: Redis cache(缓存) 和 session 驱动程序已经包含在Laravel框架内.

流水线(Pipelining)

流水线模式(Pipelining)是当你需要一次向服务器发送多个命令的时候使用的方法.你需要使用 `pipeline` 命令来开头:

流水线模式可以一次向**Redis**服务器发送多条指令

```
Redis::pipeline(function($pipe)
{
    for ($i = 0; $i < 1000; $i++)
    {
```

```
        $pipe->set("key:$i", $i);
    }
});
```

Artisan 命令行工具

Artisan CLI

简介

Artisan是Laravel中自带的命令行工具的名称。它提供了一些开发过程中有用的命令用。它是基于强大的Symfony Console 组件开发的。

用法

列出所有可用命令

执行 `list` 命令可以列出所有可用的Artisan命令：

```
php artisan list
```

查看某个命令的帮助文档

每个命令都包含有 "help" 信息，用以提供此命令所允许的参数和选项。只需在命令前边添加 `help` 即可：

```
php artisan help migrate
```

指定配置环境

通过指定 `--env` 选项，你可以指定命令执行时的配置环境：

```
php artisan migrate --env=local
```

显示当前的**Laravel**版本

你可以使用 `--version` 选项查看当前安装的Laravel的版本：

```
php artisan --version
```

译者:王赛（Bootstrap中文网）

Artisan 开发

简介

除了 Artisan 本身提供的命令之外，您也可以建立与您的应用程序相关的命令，这些自建命令将会存放在 `app/commands` 目录底下；然而，您可以任意选择存放位置，只要您的命令能够被 composer.json 自动载入。

自定义命令

生成类

要创建一个新的命令，您可以使用 command:make 这个 Artisan 命令，这将产生一个命令基本文件协助您开始编码：

生成一个新的命令类

```
php artisan command:make FooCommand
```

默认情况下，生成的命令将被储存在 `app/commands` 目录；然而，您可以指定自定义路径或命名空间：

```
php artisan command:make FooCommand --path=app/classes --namespace=Classes
```

在创建命令时，加上 `--command` 这个选项，将可以指定这个命令的名称：

```
php artisan command:make AssignUsers --command=users:assign
```

撰写自定义命令

当自定义命令生成后，您需再填写命令的 `名称` 与 `描述`，这部份将会显示在命令行表清单的画面上。

当您的自定义命令被执行时，将会调用 `fire` 方法，您可以在此加入任何的逻辑判断。

参数与选项

`getArguments` 与 `getOptions` 方法是用来接收要传入您的自定义命令的地方，这两个方法都会回传

一组命令数组，并由数组清单所组成。

当定义 `arguments` 时，该数组对应的值表示如下：

```
array($name, $mode, $description, $defaultValue)
```

参数 `mode` 可以是下列其中一项：`InputArgument::REQUIRED` 或 `InputArgument::OPTIONAL`.

当定义 `options` 时，该数组对应的值表示如下：

```
array($name, $shortcut, $mode, $description, $defaultValue)
```

对选项而言，参数 `mode` 可以是下列其中一项：`InputOption::VALUE_REQUIRED`, `InputOption::VALUE_OPTIONAL`, `InputOption::VALUE_IS_ARRAY`, `InputOption::VALUE_NONE`.

该 `VALUE_IS_ARRAY` 模式表示调用命令时可以传入多个值：

```
php artisan foo --option=bar --option=baz
```

该 `VALUE_NONE` 模式表示将选项当作是"开关"

```
php artisan foo --option
```

取得输入

当您的命令执行时，您需要让您的应用程序可以获取到这些参数和选项的值，要做到这一点，您可以使用 `argument` 和 `option` 方法：

取得自定义命令的输入参数

```
$value = $this->argument('name');
```

取得所有自定义命令的输入参数

```
$arguments = $this->argument();
```

取得自定义命令的输入选项

```
$value = $this->option('name');
```

取得所有自定义命令的输入选项

```
$options = $this->option();
```

产生输出

显示信息到终端上，您可以使用 `info`, `comment`, `question` 和`error`方法，每一种方法将会对应到一个 ANSI 颜色。

显示信息到终端

```
$this->info('Display this on the screen');
```

显示错误信息到终端

```
$this->error('Something went wrong!');
```

询问式输入

您也可以使用 `ask` 和 `confirm` 方法来提示用户进行输入：

提示用户进行输入

```
$name = $this->ask('What is your name?');
```

提示用户进行加密输入

```
$password = $this->secret('What is the password?');
```

提示用户进行确认

```
if ($this->confirm('Do you wish to continue? [yes|no]'))
{
    //
}
```

您也可以指定一个默认值给 `confirm` 方法，可以是 `true` 或 `false`：

```
$this->confirm($question, true);
```

注册命令

注册一个 **Artisan** 命令

当您的自定义命令完成后，您需要向 Artisan 注册才能使用，通常是在 `app/start/artisan.php` ，在此文件内，您可以使用 `Artisan::add` 方法注册该命令：

```
Artisan::add(new CustomCommand);
```

在 **IoC Container** 内注册命令

如果您的自定义命令是在应用程序 IoC container 内注册，您需要使用 `Artisan::resolve` 方法让 Artisan 可以使用：

```
Artisan::resolve('binding.name');
```

在 **Service Provider** 内注册命令

如果您需要从 service provider 注册命令，您应该在 provider 的 `boot` 方法内调用 `commands` 方法，传入 IoC container 绑定此命令：

```
public function boot()
{
    $this->commands('command.binding');
}
```

调用其它命令

有时候您可能希望在您的命令内部调用其它命令，此时您可以使用 `call` 方法：

```
$this->call('command:name', array('argument' => 'foo', '--option' => 'bar'));
```

中文翻译贡献者名单

- 王赛

王赛

- 邮箱:wangsai@bootcss.com
- 网址: Bootstrap中文网 & Laravel中文网
- Github: https://github.com/wangsai/

- 何文祥

何文祥

- 邮箱:erhuabushuo@gmail.com
- 网址: 二话不说
- Github: https://github.com/erhuabushuo/

- 许晨光

许晨光

- 邮箱:gukemanbu@126.com
- Github: https://github.com/gukemanbu