Lavavel
V6 框架中文文档

编　　写　　人：　翻译

日　　期：　2024-12-05

版　　本：　V1.0.0

修改历史

日期	版本	作者	修改内容
2024-12-05	1.0.0.01	翻译	初始版本

目录

第一章 序言

1.1. 新版特性

Laravel 6.0 (LTS 版本) 在 Laravel 5.8 基础上继续进行优化。包括引入语义化版本、优化授权响应、支持任务中间件、新增懒集合、优化数据库子查询。

1.1.1. 语义化版本

Laravel 框架包 laravel/framework 现在遵循语义化版本标准。

1.1.2. 兼容 Laravel Vapor

Laravel 6.0 提供了对 laravel Vapor 的兼容，这是一个针对 Laravel 应用的自动扩容无服务器部署平台。

1.1.3. 优化授权响应

在 Laravel 6.0 中，我们可以使用 Gate::inspect 方法和授权响应消息来轻松实现。

1.1.4. 任务中间件

任务中间件允许你封装自定义的队列任务异常业务逻辑，避免在任务自身处理中混入对应样板代码。

1.1.5. 懒集合

在 Laravel6.0 中新引入了一个 LazyColection 类来对 Collection 类进行补充，LazyCollection 底层基于 PHP 的生成器实同，适用于处理大型数据集。

例如，假设您的应用需要处理 GB 级别的日志文件，并使用 Laravel 的集合方法来解析日志，这个时候将整个日志文件一次性读取到内存显示是不合适的，这个时候懒集合类就派上用场了，它可以每次只取文件的一小部分到内存。

```
use App\LogEntry;

use Illuminate\Support\LazyCollection;
```

```
LazyCollection::make(function () {
    $handle = fopen('log.txt', 'r');

    while (($line = fgets($handle)) !== false) {
        yield $line;
    }
})
->chunk(4)
->map(function ($lines) {
return LogEntry::fromLines($lines);
})
->each(function (LogEntry $logEntry) {
    // Process the log entry...
});
```

不过，从 Laravel6.0 开始，查询构建器的 cursor 方法已经被升级为返回 LazyCollection 实例，这样一来，我们就可以像之前一样执行一次数据库查询，但是每次只会加载一个 Eloquent 模型到内存。

1.1.6. Eloquent 子查询优化

Laravel6.0 引入了多个数据库子查询优化和增强支持。

1.1.7. Laravel UI

之前版本 Laravel 提供的典型的前端脚手架代码现在被提取到独立的 Composer 扩展包 laravel/ui 中，这样一来可以让 UI 脚手架代码的开发和维护与主框架分离。

1.2. 升级指南

1.2.1. 重要更新概览

影响较大：

授权资源 & viewAny

字符串 & 数组辅助函数

影响中等：

认证 RegisterController

不再支持 Carbon 1.x

数据库 Capsule::table 方法

队列重试限制

重发邮箱验证路由

Input 门面

预计升级时间：1 个小时。

1.2.2. PHP 7.2

Laravel 6.0 要求 PHP 版本大于等于 7.2。

1.2.3. 更新依赖

在 composer.json 文件中更新 laravel/framework 依赖到^6.0。

当然，不要忘了检查应用所使用的第三方扩展包是否支持 Laravel6.0，如果需要升级的话也要更新。

1.2.4. 授权

授权资源& viewAnyCarbon

1.2.5. Carbon

不再支持 Carbon 1.x

1.2.6. 配置

AWS_REGION 环境变量

1.2.7. 数据库

1.2.8. Eloquent

1.2.9. 邮箱验证

1.2.10. 辅助函数

1.2.11. 本地化

1.2.12. 邮件

1.2.13. 通知

1.2.14. 密码重置

1.2.15. 队列

1.2.16. 请求

1.2.17. 任务调度

1.2.18. 存储

1.2.19. URL 生成

1.2.20. 其他

我们来鼓励您查看 laravel/laravel 代码仓库的更新日志。

第二章 快速入门

2.1. 安装配置

2.1.1. 服务器要求

需要确认自己的环境满足以下要求：

PHP >= 7.2.0

PHP BCMath 扩展

PHP Ctype 扩展

PHP JSON 扩展

PHP Mbstring 扩展

PHP OpenSSL 扩展

PHP PDO 扩展

PHP Tokenlaer 扩展

PHP XML 扩展

2.2. 目录结构

2.2.1. 简介

框架默认的目录结构试图为不管是大型应用还是小型应用提供一个良好的起点。当然，您也可以按照自己的喜好重新组织应用的目录结构，只要 Composer 可以自动加载入它们要即可。

许多初学者都会困惑框架为什么没有提供 models 目录，我可以负责任的告诉大家，这是故意的。不过对于国内开发者，尤其是 PHP 开发者来说，models 目录用于存放与数据交互的模型类应该没有什么异议，而业务逻辑应该放到 services 这种目录之下。所以推荐大家在生成模型类的时候指定生成到 app/Models 目录下。

```
php artisan make:model Models/Test
```

2.2.2. 根目录

应用目录

App 目录包含了应的核心代码。

启动目录

Bootstrap 目录包含了少许文件.app.php 用于框架的启动和自动载入配置，还有一个 cache 文件夹，里面包含了框架为提升性能所生成的文件，如路由和服务缓存文件。

配置目录

Config 目录包含了应用所有的配置文件。

数据库目录

Database 目录包含了数据库迁移文件及填充文件。

对外公开目录

Public 目录包含了应用入口文件 index.php 和前端资源文件。

资源目录

Resources 目录包含了应用视图文件和未编译的原生前端资源文件，以及本地化语言文件。

路由目录

Routes 目录包含了应用定义的所有路由。

文件存储目录

Storage 目录包含了编译后的 blade 模板、基于文件的 Session、文件缓存，以及其他由框架生成的文件，该目录细分为 app、frameworkd 和 logs 子目录。

测试目录

Tests 目录包含自动化测试文件，其中默认已经提供了一个开箱即用的 PHPUnit 救命。

Vendor 目录

Vendor 目录包含了应用所有通过 Composer 加载的依赖。

2.2.3. 应用目录

App 目录个包含多个子目录，如 Console、Http、Providers 等。Console 和 Http 目录提供了进入应用核心的 API,HTTP 协议和 CLI 是和应用进行交互的两种机制，

但实际上并不包含应用逻辑。换句话说，它们只是两个向应用发送命令的方式。

2.3. 部署应用到服务器

当您准备部署应用到生产环境时，有一些重要的事情可以确保应用尽可能高效地运行。

第三章 底层原理

3.1. 一次请求的生命周期

3.1.1. 简介

当我们使用现实世界中的任何工具时，如果理解了该工具的工作原理，那么用起来就会得心应手，应用开发也是如此。

如果您不能马上理解所有这些条款，不要失去信心！先试着掌握一些基本的东西，你的知识水平将会随着对文档的探索而不断提升。

3.1.2. 生命周期概览

第一件事

请求入口是 public/index.php 文件，这里是加载框架其他部分的起点。

载入 Composer 生成的自动加载设置，然后从 bootstrap/app.php 脚本获得 Laravel 应用实例，Laravel 的第一个动作晃是创建服务容器实例。

HTTP/Console 内核

接下来，请求被发送到 HTTP 内核或 Console 内核（分别用于处理 Web 请求和 Artisan 命令），这取决于进入应用的请求类型。这两个内核是所有请求都要经过的中央处理器，现在，就让我们聚焦在位于 app/Http/Kernel.php 的 HTTP 的内核。

HTTP 内核继承自 Illuminate\Foundation\Http\Kernel 类，该类应义了一个 bootstrappers 数组，这个数组中的类在请求被执行前运行，这些 bootstrappers 配置了错误处理、日志、检测应用环境以及其它在请求被处理前需要执行的任务。

HTTP 内核还定义了一系列所有请求在处理前需要经过的 HTTP 中间件，这些中间件处理 HTTP 会话的读写、判断应用是否处于维护模式、验证 CSRF 令牌等等。

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

内核启动过程中最重要的动作之一就是为应用载入服务提供者，应用的所有服务提供者都被配置在 config/app.php 配置文件的 providers 数组中。首先，所有提供者的 register 方法被调用，然后，所有提供被注册之后，boot 方法被调用。

服务提供者负责启动框架的所有各种各样的组件，比如数据库、队列、验证器，以及路由组件等，正是因为他们启动并配置了框架提供的所有特性，所以服务提供者是整个框架启动过程中最重要的部分。

一旦应用被启动并且所有的服务提供者被注册，Request 将会被交给路由器进行分发，路由器将会分发请求到路由或控制器，同时运行所有路由指定的中间件。

3.1.3. 聚焦服务提供者

应用实例被创建后，服务提供者被注册，请求被交给启动后的应用进行处理，整个过程就是这么简单。

对应用如何通过服务提供者构建和启动有一个牢固的掌握非常有价值，当然，应用默认的服务提供者存放在 app/Providers 目录下。

默认情况下，AppServicePrvider 是空的，这里是添加自定义启动和服务容器绑定的最佳位置，当然，对于大型应用，您可能希望创建多个服务提供者，每一个都有着更加细粒度的启动。

3.2. 服务容器

3.2.1. 简介

服务容器是一个用于管理类依赖和执行依赖注入的强大工具。依赖注入听上

去很花哨，其实质是通过构造函数或者某些情况下通过 setter 方法将类依赖注入到类中。

3.2.2. 绑定

3.2.2.1. 绑定基础

简单的绑定

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

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

绑定一个实例

Singleton 方法绑定一个只会解析一次的类或接口到容器，然后接下来对容器的调用将会返回一个对象实例。

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

绑定实例

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

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

绑定原始值

你可能有一个接收注入类的类，同时需要注入一个原生的数值比如整型，可

以结合上下文轻松注入这个类需要的任何值。

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

3.2.2.2. 绑定接口到实现

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

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

这段代码告诉窗口当一个类需要 EventPusher 的实现时会注入RedisEventPusher，现在我们可以在构造器或者任何其他通过服务容器注入依赖的地方进行 EventPusher 接口的依赖注入。

3.2.2.3. 上下文绑定

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

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


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

```
    ->needs(Filesystem::class)
    ->give(function () {
        return Storage::disk('local');
    });

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

3.2.2.4. 标签

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

```
$this->app->bind('SpeedReport', function () {
    //
});
$this->app->bind('MemoryReport', function () {
    //
});
$this->app->tag(['SpeedReport', 'MemoryReport'], 'reports');
```

这些服务被打上标签后，可以通过 tagged 方法来轻松解析它们。

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

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

3.2.2.5. 扩展绑定

extend 方法允许对解析服务进行修改。例如，当服务被解析后，可以运行额外代码装饰或配置该服务。extend 方法接收一个闭包来返回修改后的服务。

```
$this->app->extend(Service::class, function($service)
{
    return new DecoratedService($service);
});
```

3.2.3. 解析

Make 方法

有很多方式可以从容器中解析对象，首先，你可以使用 make 方法。

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

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

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

某些类的依赖不能通过窗口来解析，你可以通过关联数组方式将其传递到 makeWith 方法来注入。

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

自动注入

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

3.2.4. 容器事件

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

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

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

3.3. 服务提供者

3.3.1. 简介

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

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

3.3.2. 编写服务提供者

所有的服务提供者都继承自 Illuminate\Support\ServiceProvider 类。大部分服务提供者都包含两个方法：register 和 boot。在 register 方法中，您唯一要做的事就是绑定服务到服务容器，不要尝试在该方法中注册事件监听器、路由或者任何其他功能。

3.3.2.1. Register 方法

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

```
<?php

namespace App\Providers;


use Riak\Connection;

use Illuminate\Support\ServiceProvider;


class RiakServiceProvider extends ServiceProvider{

/**

* 在容器中注册绑定.

*

* @return void

*/

public function register()

{

$this->app->singleton(Connection::class, funct

ion ($app) {

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

});

}

}
```

该服务只定义了一个 register 方法，并使用改方法在服务容器中定义了一个

Riak\Connection 的实现。

Bindings 和 singletons 属性

如果你的服务提供者注册了很多简单的绑定，你可能希望使用 bindings 和 singletons 属性来替代手动注册每个容器绑定以简化代码。

3.3.2.2. boot 方法

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

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

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

```
}
```

Boot 方法的依赖注入

我们可以在 boot 方法中对依赖进行类型提示，服务容器会自动注入你所需要的依赖。

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

public function boot(ResponseFactory $response){

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

//

});

}
```

3.3.3. 注册服务提供者

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

要注册你自己的服务提供者，只需要将其追加到该数组中即可。

```
'providers' => [

    // 其它服务提供者

    App\Providers\ComposerServiceProvider::class,

],
```

3.3.4. 延迟加载服务提供者

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

Laravel 编译并保存所有延迟服务提供者提供的服务及服务提供者的类名。然

后，只有当你尝试解析其中某个服务时框架才会加载其服务提供者。

3.4. 门面(Facades)

3.4.1. 简介

门面为应用服务容器中的绑定类提供了一个静态接口，Laravel 内置了很多门面，你可能在不知道的情况下正在使用它们。Laravel 的门面作为服务容器中底层类的静态代理，相比于传统静态方法，在维护时能够提供更加易于测试、更加灵活、简明优雅的语法。

框架所有门面都定义在 Illuminate\Support\Facades 命名空间下，所以我们可以轻松访问到门面。

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

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

    return Cache::get('key');

});
```

在整个框架中，很多例子使用了门面来演示框架的各种功能特性。

3.4.2. 何时使用门面

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

但是，使用门面也有需要注意的地方，一个最主要的危险就是类范围变化。由于门面如此好用并且不需要注入，在单个类中使用过多门面会让类很容易变得越来越大，使用依赖注入则会让此类问题缓解。因此，使用门面的时候要尤其注意类的大小，以便控制其有限职责。

3.4.2.1. 门面 vs.依赖注入

依赖注入的最大优点是可以替换注入类的实现。这在测试时很有用。因为你可以注入一个模拟并且可在存根上断言不同的方法。

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

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

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

    return Cache::get('key');

});
```

3.4.2.2. 门面 vs.辅助函数

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

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

return view('profile');
```

门面和辅助函数并不存在实质性的差别，使用辅助函数的时候，可以像测试相应门面那样测试它们。

3.4.3. 门面工作原理

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

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

3.4.4. 实时门面

使用实时门面，可以将应用中的任意类当做门面来使用。

3.4.5. 门面类列表

序号	门面	类	处理方式
1	App	Illuminate\Foundation\Application	app
2	Artisan	Illuminate\Contracts\Console\Kernel	artisan
3	Auth	Illuminate\Auth\Auth\Manager	auth
4	Auth(实例)	Illuminate\Contracts\Guard	Auth.driver
5	Blade	Illuminate\View\Compilers\BladeCompiler	Blade.compile
6	Broadcast	Illuminate\Contracts\Broadcasting\Factory	
7	Broadcast(实例)	Illuinate\Contracts\Broadcasting\Broadcaster	
8	Bus	Illuminate\Contracts\Bus\Dispatcher	
9	Cache	Illuminate\Cache\CacheManager	cache
10	Cache(实例)	Illuminate\Cache\Repository	cache.store
11	Config	Illuminate\Config\Repository	config

12	Cookie	Illuminate\Cookie\CookieJar	cookie
13	Crypt	Illuminate\Encryption\Encrypter	encrypter
14	DB	Illuminate\Database\DatabaseManager	db

3.5. 契约（Contracts）

3.5.1. 简介

契约是指框架提供的一系列定义的核心服务的接口。例如，Illuminate\Contracts\Queue\Queue 契约定义了队列任务需要实现的方法，Illuminate\Contracts\Mail\Mailer 契约定义了发送邮件所需要实现的方法。

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

3.5.1.1. 契约 Vs. 门面

门面为服务提供了便捷方式，不再需要从服务容器中类型提示和契约解析即可直接通过静态门面调用。

不同于门面不需要再构造器中进行类型提示，契约允许您在类中定义显式的依赖。有些开发者喜欢门面带来的便捷，也有些开发者倾向于使用契约，他们喜欢定义明确的依赖。

3.5.2. 何时使用契约

正如上面所讨论的，大多数情况下使用契约还是门面取决于个人或团队的喜好。只要保持类的职责单一，你会发现使用契约和门面并没有什么实质性的差别。

但是，对契约您可能还有些疑问。例如，为什么要全部使用接口？使用接口是不是更复杂？

3.5.3. 如何使用契约

要实现一个契约，需要在解析类的构造函数中类型提示这个契约接口。

3.5.4. 契约列表

下面是契约列表，以及其对应的“门面”：

契约	对应门面
Illuminate\Contracts\Auth\Access\Authorizable	
Illuminate\Contracts\Auth\Access\Gate	Gate
Illuminate\Contracts\Auth\Authenticatable	
Illuminate\Contracts\Auth\CanResetPassword	
Illuminate\Contracts\Auth\Factory	Auth
Illuminate\Contracts\Auth\Guard	Auth::guard()
Illuminate\Contracts\Auth\PasswordBroker	Password::broker()
Illuminate\Contracts\Auth\PasswordBrokerFactory	Password
Illuminate\Contracts\Auth\StatefulGuard	
Illuminate\Contracts\Auth\SupportsBasicAuth	

第四章 基础组件

4.1. 路由

4.1.1. 路由入门

最基本的路由只接收一个 URL 和一个闭包。

```
Route::get('hello', function () {
return 'Hello, Welcome to LaravelAcademy.org';
});
```

我们可以注册路由来响应任何 HTTP 请求动作。

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

4.1.1.1. 路由重定向

如果您需要定义一个重定向到其他的 URI 的路由，可以使用 Route::redirect 方法，该方法非常方便。

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

您还可以使用 Route::permanentRedirect 方法来返回 301 状态码。

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

4.1.1.2. 路由视图

如果您的路由需要返回一个视图，可以使用 Route::view 方法。

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

4.1.2. 路由参数

4.1.2.1. 必选参数

有时我们需要在路由中获取 URI 请求参数。

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

4.1.2.2. 可选参数

有必选参数就有可选参数，可以通过在参数名后加一表 ？ 标记来实现。

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

4.1.2.3. 正则约束

可以通过路由实例上的 where 方法来约束路由参数的格式。

```
Route::get('user/{name}', function ($name) {
// $name 必须是字母且不能为空
})->where('name', '[A-Za-z]+');
Route::get('user/{id}', function ($id) {
    // $id 必须是数字
})->where('id', '[0-9]+');
Route::get('user/{id}/{name}', function ($id, $name)
{
```

```
    // 同时指定 id 和 name 的数据格式
})->where(['id' => '[0-9]+', 'name' => '[a-z]+']);
```

使用正则还有一个好处就是避免了 user/{id}和 user/{name}的混淆。

4.1.2.4. 命名路由

命名中为生成 URL 或重定向提供了方便,在路由定义之后使用 name 方法链的方式来定义该路由的名称。

```
Route::get('user/profile', function () {
// 通过路由名称生成 URL
return 'my url: ' . route('profile');
})->name('profile');
```

还可以为控制器动作指定路由名称。

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

4.1.3. 路由分组

路由分组的目的是让我们在多个路由中共享相同的路由属性，比如中间件和命名空间等,这样的话我们定义了大量的路由时就不必为每一个路由单独定义属性。共享属性以数组的形式作为第一个参数被传递给 Route::group 方法。

嵌套的分组会尝试智能地将属性合并到父分组中，中间件和 where 条件会直接被合并，而路由命名、命名空间、以及路由前缀会被附加到父组件对应属性之后。

4.1.3.1. 中间件

要给某个路由分组中定义的所有路由分配中间件,可以在定义分组之前使用 middleware 方法。

```
Route::middleware(['first', 'second'])->group(functio
```

```
n () {
Route::get('/', function () {
// Uses first & second Middleware
});
Route::get('user/profile', function () {
// Uses first & second Middleware
});
});
```

4.1.3.2. 命名空间

路由分组另一个通用例子就是使用 namespace 方法分配同一个 PHP 命名空间给该分组下的多个控制器。

```
Route::namespace('Admin')->group(function () {
// Controllers Within The "App\Http\Controllers\Ad
min" Namespace
});
```

4.1.3.3. 子域名路由

路由分组还可以被用于处理子域名路由，子域名可以像 URI 一样被分配给路由参数，从而允许补获子域名的部分用于路由或者控制器，子域名可以在定义分组之前调用 domain 方法来指定。

```
Route::domain('{account}.blog.dev')->group(function ()
{
    Route::get('user/{id}', function ($account, $id) {
        return 'This is ' . $account . ' page of User ' . $id;
    });
```

```
});
```

4.1.3.4. 路由前缀

prefix 方法可以用来为分组中每个路由添加一个指定 URI 前缀。

例如，您可以为分组中所有路由 URI 添加 admin 前缀。

```
Route::prefix('admin')->group(function () {
    Route::get('users', function () {
        // Matches The "/admin/users" URL
    });
});
```

这样，我们就可以通过 http:.//blog.test/admin/users 访问路由了。

4.1.3.5. 路由名称前缀

name 方法可通过传入字符串为分组中的每个路由名称设置前缀。

例如，您可能想要在所有分组路由的名称前添加 admin 前缀。

4.1.4. 路由模型绑定

注入模块 ID 到路由或控制器动作时，通常需要查询数据库才能获取相应的模型数据。

4.1.4.1. 隐式绑定

Laravel 会自动解析定义在路由或控制器动作中的 Eloquent 模型类型声明。

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

4.1.4.2. 显式绑定

要注册显式绑定，可以使用路由器的 model 方法来为给定参数指定绑定类。

```
public function boot()
{
    parent::boot();
    Route::model('user_model', App\User::class);
}
```

接下来，在 routes/api.php 中定义一个包含{user}参数的路由。

```
$router->get('profile/{user_model}', function(App\Use
r $user) {
    dd($user);
});
```

由于我们已经绑定{user_model}参数到 App\User 模型，User 实例会被注入以该路由。

4.1.5. 兜底路由

使用 Route::fallback 方法可以定义一个当所有其他路由都未能匹配请求 URL 时所执行的路由。

4.1.6. 频率限制

框架自带了一个中间件用于限制对应用路由的访问频率。Throttle 中间件接收两个参数用于判断给定时间内（单位：分钟）的最大请求次数。

```
Route::middleware('auth:api', 'throttle:60,1')->group
(function () {
    Route::get('/user', function () {
        //
    });
});
```

超出访问次数后，会返回 429 状态码并提示 Too many requests.

4.1.7. 表单方法伪造

HTML 表单不支持 PUT、PATCH 或者 DELETE 请求方法，因此，需要添加一个隐藏的_method 字段。

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

4.1.8. 访问当前路由

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

```
// 获取当前路由实例
$route = Route::current();
// 获取当前路由名称
$name = Route::currentRouteName();
// 获取当前路由 action 属性
$action = Route::currentRouteAction();
```

4.2. 中间件

4.2.1. 简介

中间件为过滤进入应用的 HTTP 请求提供了一套便利的机制。

框架自带了一些中间件，包括认证、CSRF 保护中间件等等。所有的中间件都位于 app/Http/Middleware 目录下。

4.2.2. 定义中间件

要创建一个中间件，可以通过 Artisan 命令 make:middleware

```
php artisan make:middleware CheckToken
```

这个命令会在 app/Http/Middleware 目录下创建一个新的中间件。

4.2.2.1. 请求之前/之后的中间件

一个中间件是请求前还是请求后执行取决于中间件本身。

4.2.3. 注册中间件

中间件分为三类，分别是全局中间件、中间件组和指定路由中间件。

4.2.3.1. 全局中间件

如果您想要定义的中间件在每一个 HTTP 请求时都被执行，只需要将相应的中间件类添加到 app/Http/Kernel.php 的数组属性$middleware 中即可。

4.2.3.2. 中间件组

有时候您可能想到通过指定一个键名的方式将相关中间件分到同一个组里面，这样可以更方便地将其分配到路由中。

4.2.3.3. 指定路由中间件

如果您想要分配中间件到指定路由，首先应该在 app/Http/Kernelphp 文件中分配该中间件一个 key.

4.2.4. 中间件参数

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

4.2.5. 终端中间件

终端中间件，可以理解为一个善后的后台处理中间件。有时候中间件可能需要在 HTTP 响应发送到浏览器之后做一些工作。比如，Laravel 内置的 session 中间件会在响应发送到浏览器之后将 Session 数据写到存储器中，为了实现这个功

能，需要定义一个终止中间件关添加 terminate 方法到这个中间件。

4.3. CSRF 保护

4.3.1. 简介

跨站请求伪造（CSRF）是一种通过伪装授权用户的请求来攻击授信网站的恶意漏洞。

4.3.2. 排除指定 URL 不做 CSRF 安全校验

有时候我们需要从 CSRF 保护中间件中排除一些 URL。

4.4. 控制器

4.4.1. 简介

控制器用于将相关的 HTTP 请求封装到一个类中进行处理，这些控制器类存放在 app/Http/Controllers 目录下。

4.4.2. 控制器入门

4.4.2.1. 定义控制器

所有的控制器应该继承自框架自带的控制器基类 App\Http\Controllers\Controller，我们为该控制器添加一个 show 方法。

```
<?php

namespace App\Http\Controllers;

use App\User;

use Illuminate\Http\Request;

class UserController extends Controller

{

    /**

    * 为指定用户显示详情

    *
```

```
    * @param int $id
    * @return Response
    * @author LaravelAcademy.org
    */
    public function show($id)
    {
        return view('user.profile', ['user' => User::findOrFail($id)]);
    }
}
```

我们可以像这样定义指向该控制器动作的路由：

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

现在，如果有一个请求匹配上面的路由 URI，UserController 的 show 方法就会被执行。

4.4.2.2. 命名空间

您应用注意到我们在定义控制器路由的时候没有指定完整的控制器命名空间，而只是定义了 App\Http\Controllers 之后的部分。这是因为默认情况下，RouteServiceProvider 将会在一个指定了控制器所在命名空间的路由分组中载入路由文件，故而我们只需要指定后后面相对命名空间即可。

4.4.2.3. 单一动作控制器

如果您想要定义一个只处理一个动作的控制器，可以在这个控制器中定义 __invoke 方法。

```
<?php
namespace App\Http\Controllers;
use App\User;
use App\Http\Controllers\Controller;
```

```
class ShowProfile extends Controller
{
    /**
    * 展示给定用户的个人主页
    *
    * @param int $id
    * @return Response
    */
    public function __invoke($id)
    {
        return view('user.profile', ['user' => User::findOrFail($id)]);
    }
}
```

当您为这个单动作控制器注册路由的时候，不需要指定方法。

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

这背后的原理是__invoke()方法会被自动调用。

4.4.3. 控制器中间件

中间件可以像这样分配给控制器路由。

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

4.4.4. 资源控制器

资源控制器可以让我们很便捷地构建基于资源的 RESTful 控制器，例如，您可能想要在应用中创建一个控制器，用于处理关于文章存储的 HTTP 请求。

4.4.5. 依赖注入

4.4.5.1. 构造函数注入

框架使用服务容器解析所有的控制器，因此，可以在控制器的构造函数中注入任何依赖，这些依赖会被自动解析并注入到控制器实例中。

4.4.5.2. 方法注入

除了构造函数注入之外，还可以在控制器的动作方法中进行依赖注入。

4.4.6. 路由缓存

如果您的应用完全基于控制器路由，可以使用路由缓存，使用路由缓存将会会极大降低注册所有应用路由所花费的时间开销。

4.5. HTTP 请求

4.5.1. 访问请求实例

在控制器中获取当前 HTTP 请求实例，需要在构造函数或方法中对 Illuminate\Http\Request 类时行依赖注入，这样当前请求实例会被服务容器自动注入。

4.5.2. 请求字符串处理

默认情况下，在 App\Http\Kernel 的全局中间件堆栈中引入了 TrimStrings 和 ConvertEmptyStringsToNull 中间件。这些中间件会自动对请求中的字符串字段进行处理，前者将字符串两端的空格清除，后者将空字符串转化为 null。

4.5.3. 获取请求输入

您可以使用 all 方法以数组格式获取所有输入值。

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

4.5.3.1. 上一次请求输入

框架允许您在两次请求之间保存上一次输入数据，这个特性在检测校验数据

失败后需要重新填充表单数据时很有用。

4.5.3.2. Cookie

为了安全起见，框架创建的 Cookie 都经过加密并使用一个认证码进行签名。

4.5.4. 文件上传

4.5.4.1. 获取上传的文件

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

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

$file = $request->photo;
```

您可以使用 hasFile 方法判断文件在请求中是否存在。

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

4.5.4.2. 保存上传的文件

要保存上传的文件，需要使用您所配置的某个文件系统，对应配置位于 config/filesystems.php。

框架默认使用 local 配置存放上传文件，即本地文件系统，默认根目录是 storage/app,public 也是本地文件系统，只不过存放在这里的文件可以被公开访问，其对应的根目录是 storage/app.public。

4.5.5. 配置信任代理

如果您的应用运行在一个会中断 SSL 证书的负载均衡后，有时候应用不会生

成 HTTPS 链接。

4.6. HTTP 响应

4.6.1. 创建响应

所有路由和控制器处理完业务逻辑之后都会返回一个发送到用户浏览器的响应，框架提供了多种不同的方式来返回口响应，最基本的响应是从路由或控制器返回一个简单的字符串，框架会自动将这个字符串转化为一个完整的 HTTP 响应。

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

通常我们并不只是从路由动作简单返回字符中和数组，大多数情况下，都会返回一个完整的 Illuminate\Http\Response 实例或视图。返回一个完整的 Response 实例允许您自定义响应的 HTTP 状态码和头信息。

4.6.2. 重定向

重定向响应是 Illuminate\Http\RedirectResponse 类的实例，包含了必要的头信息将用户重定向到另一个 URL，有很多方法来生成 RedirectReponse 实例，最简单的方法就是使用全局辅助函数 redirect。

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

4.6.3. 其他响应类型

我们还可以通过辅助函数 reponse 很方便地生成其他类型的响应实例。

4.6.4. 响应宏

如果您想要定义一个自定义的可以在多个路由和控制器中复用的响应，可以

使用 Resonse 门面上的 macro 方法。

4.7. 视图

4.7.1. 创建视图

视图包含应用的 HTML 代码，并将应用的控制器逻辑和表现逻辑进行分离。视图文件存放在 resources/views 目录。

4.7.2. 传递数据到视图

可能简单通过数组方式将数据传递到视图。

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

4.7.3. 视图 Composer

视图 Composer 是当视图被渲染时的回调函数或类方法。

4.8. URL 生成

4.8.1. 简介

框架提供了很多个辅助函数来帮助我们在应用中生成 URL。这些函数主要用于在视图模板和 API 响应中构建链接，或者生成重定向响应。

4.8.2. 快速入门

4.8.2.1. 生成 URL

Url 辅助函数可用于为应用生成任意 URL，并且生成的 URL 会自动使用当前请求的 scheme 和 host 属性。

```
$post = App\Post::find(1);

echo url("/posts/{$post->id}");
```

4.8.2.2. 访问当前 URL

如果没有传递路径信息给 url 辅助函数，则会返回一个

Illuminate\Routing\UrlGenerator 实例，从页允许您访问当前 UR 的信息。

```
use Illuminate\Support\Facades\URL;

echo URL::current();
```

4.8.3. 命名路由 URL

4.8.4. 控制器动作 URL

action 辅助函数用于为控制器动作生成 URL，和路由中定义的一样，您不需要传递完整的控制器命名空间。

4.8.5. 参数默认值

我们要吧在当前请求中使用 URL:defaults 方法为这个参数定义一个默认值。

4.9. Session

4.9.1. 简介

由于 HTTP 协议本身是无状态的，上一个请求与下一个请求无任何关联，为此我们引入 Session 来存储用户请求信息以解决特定场景下无状态导致的问题（比如登录、购物）。

4.9.2. 使用 Session

4.9.2.1. 获取数据

主要有两种方式处理 Session 数据，全局的辅助函数 session，或者通过 Request 实例（启动过程中会将 Session 数据设置到请求实例的 session 属性中）。

4.9.2.2. 存储数据

要在 Session 中存储数据，通常可以通过 put 方法或 session 辅助函数。

```
//通过调用请求实例的 put 方法

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

// 通过全局辅助函数 session
```

```
session(['key' => 'value']);
```

4.9.2.3. 一次性数据

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

```
$request->session()->flash('status', '登录成功!');
```

4.9.2.4. 删除数据

Forget 方法从 Session 中移除指定数据，如果您要从 Session 中移除所有数据，可以使用 flush 方法。

```
// Forget a single key...
$request->session()->forget('key');
// Forget multiple keys...
$request->session()->forget(['key1', 'key2']);
$request->session()->flush();
```

4.9.2.5. 重新生成 SessionID

重新生成 SessionID 经常用于阻止恶意用记对应用进行 session fixaction 攻击。

如果您使用内置的 LoginController 的话，框架会在认证期间自动重新生成 session ID，如果您需要手动重新生成 sessionID，可以使用 regenerate 方法。

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

4.9.3. 添加自定义 Session 驱动

4.9.3.1. 实现驱动

自定义 Session 驱动需要实现 SessionHandlerInterface 接口，改接口包含少许我们需要实现的方法，比如一个基于 MongoDB 的 Session 驱动实现如下。

```
<?php

namespace App\Extensions;

class MongoSessionHandler implements SessionHandlerInterface
{

    public function open($savePath, $sessionName) {}

    public function close() {}

    public function read($sessionId) {}

    public function write($sessionId, $data) {}

    public function destroy($sessionId) {}

    public function gc($lifetime) {}
}
```

4.9.3.2. 注册驱动

Session 驱动被实现后，需要将其注册到框架，要添加额外驱动到后端，可以使用 Session 门面上的 extend 方法。

4.10. 表单验证

4.10.1. 简介

框架提供了多种方法来验证请求输入数据。默认情况下，控制器类使用 ValidatesRequest trait，该 trait 提供了便捷方法通过各种功能强大的验证规则来验证输入的 HTTP 请求。

4.10.2. 快速入门

要掌握框架强大的验证特性，让我们先看一个完整的验证表单并返回错误信息给用户的示例。

4.10.2.1.定义路由

首先，我们假定在 routes/web.php 文件中包含如下路由。

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

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

4.10.2.2.创建控制器

接下来，让我们看一个处理这些路由的简单控制器示例。

4.10.2.3.编写验证逻辑

4.10.2.4.显示验证错误信息

4.10.2.5.可选字段注意事项

默认情况下，框架自带中间件。

4.10.3. 表单请求

4.10.3.1.创建表单请求

4.10.3.2.授权表单请求

4.10.3.3.自定义错误消息

4.10.4. 自定义验证属性

如果您想要将验证消息中的:attribute 部分替换为自定义的属性名，可以通过重写 attributes 方法来指定自定义的名称。该方法会返回属性名及对应自定义名称键值对数组。

```
/**
* Get custom attributes for validator errors.
*
* @return array
*/
public function attributes()
{
    return [
        'email' => 'email address',
    ];
}
```

4.10.5. 手动创建验证器

4.10.6. 处理错误信息

4.10.7. 验证规则大全

4.10.8. 添加条件规则

4.10.9. 验证数据输入

4.10.10. 自定义验证规则

4.10.11. 隐式扩展

默认情况下，被验证的属性如果没有提供或者验证规则为 required 而值为空，

那么正常的验证规则，包括自定义扩展将不会执行。

4.11. 异常处理

4.11.1. 简介

框架默认已经为我们配置好了错误和异常处理，我们在 App\Exceptions\Handler 类中触发异常并将响应返回给用户。

4.11.2. 配置

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

对本地开发而言，您应该设计环境变更 APP_DEBUG 值为 true。在生产环境，该值应该被设置为 false。

4.11.3. 异常处理器

所有异常都由类 App\Exceptions\Handler 处理，该类包含两个方法：report 和 render。

4.11.3.1.Report 方法

Report 方法用于记录异常并将其发送给外部服务。

4.11.3.2.Render 方法

Render 方法负责将给定异常转化为发送给浏览器的 HTTP 响应，默认情况下，异常被传递给为您生成响应的基类。

4.11.3.3.可报告异常

除了在处理器的 report 和 render 方法中进行异常类型检查外，还可以在自定义异常中直接定义 report 和 render 方法。

4.11.4. HTTP 异常

有些异常来自于服务器的 HTTP 错误码，例如，这可能是一个“页面未找到”错误。

4.12. 日志

4.12.1. 简介

为了帮助您了解更多关于应用中所发生的事情，框架提供了强大的日志服务来记录日志信息到文件、系统错误日志、甚至是 Slack 以便通知整个团队。

在日志引擎之下，框架集成了 Monolog 日志库以便提供各种功能强大的日志处理器，从而允许您来通过它们定制自己应用的日志处理。

4.12.2. 配置

应用日志系统的所有配置都存放在配置文件 config/logging.php 中。

4.12.2.1.构建日志推栈

示例配置。

```
'channels' => [
'stack' => [
'driver' => 'stack',
'channels' => ['syslog', 'slack'],
],
'syslog' => [
'driver' => 'syslog',
'level' => 'debug',
],
'slack' => [
'driver' => 'slack',
'url' => env('LOG_SLACK_WEBHOOK_URL'),
'username' => 'Laravel Log',
'emoji' => ':boom:',
```

```
'level' => 'critical',

],

],
```

4.12.3. 写入日志信息

您可以使用 Log 门面记录日志信息。

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

Log::alert($error);

Log::critical($error);

Log::error($error);

Log::warning($error);

Log::notice($error);

Log::info($error);

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

4.12.4. 高级 Monolog 通道自定义

4.12.5. 创建 Monolog 处理器通道

第五章 前端开发

5.1. Blade 模板引擎

Blade 是由框架提供的非常简单但功能强大的模板引擎。

5.2. 本地化

Laravel 的本地化特性允许您在应用中轻松实现多语言支持。

5.3. 前端快速入门

5.3.1.1. 简介

框架并不强制您使用什么 JavaScript 框架或者 CSS 预处理器，不过也确实提供了对很多应用而言都很有用的 Bootstrap 和 Vue 的一些基本脚手架。

5.3.1.2. 编写 CSS

5.3.1.3. 编写 JavaScript

5.3.1.4. 添加预设命令

预设命令是 macroable 的，允许您在运行时添加额外方法到 UiComand 类。

5.4. 前端使用进阶

5.4.1.1. 简介

Laravel Mix 提供了一套流式 API，使用一些通用的 CSS 和 JavaScript 预处理器为应用定义 Webpack 构建步骤。

第六章 安全系列

6.1. 登录认证

6.1.1. 简介

实现登录认证非常简单。

6.1.1.1. 数据库考量

默认情况下，框架在 app 目录下包含了一个 Eloquent 模型 App\User。

6.1.2. 快速入门

6.1.2.1. 路由

基于框架提供的 laravel/ui 扩展包，我们可以通过运行如下命令快速生成用

户认证需要的所有路由和视图。

6.1.2.2. 视图

6.1.2.3. 认证

6.1.2.4. 获取登录用户

6.1.2.5. 路由保护

6.1.2.6. 登录失败次数限制

如果您使用了自带的 LoginController 类，就已经启用了内置的 Illuminate\Foundateion\Auth\TrottlesLogins。

6.1.3. 手动认证用户

6.1.4. 基于 HTTP 的基本认证

6.1.5. 退出

6.1.6. 添加自定义 Guard 驱动

6.1.7. 添加自定义用户提供者

6.1.8. 事件

框架支持在认证过程中触发多种事件，您可以在自己的 EventServiceProvider 中监听这些事件。

6.2. API 认证

6.2.1. 简介

默认情况下，框架通过为应用中的每个用户分配一个随机的令牌这种方式提供了一个非常简单的 API 认证解决方案。

6.3. 授权

6.3.1. 简介

除了提供开箱即用的认证服务之外，框架还提供了一个简单的方式来管理授权逻辑以便控制对资源的访问权限。和认证一样，主要有两种方式：Gate 和 Policy。

6.4. 邮箱验证

6.4.1. 简介

很多 Web 应用都要求用户注册之后验证邮箱地址才能登录，为些，框架也提供了便捷方法来发送和验证邮箱验证请求，但是这个功能是可选的，您可以启用也可以不启用。

6.5. 加密

6.5.1. 简介

框架的加密器使用 OpenSSL 来提供 AES-256 和 AES-128 加密。强烈建议使用自带的加密设置，不要尝试推出自己的加密算法。

6.6. 重置密码

6.6.1. 简介

大多数 Web 应用都提供了为用户重置密码的功能。

第七章 进阶系列

7.1. Artisan 控制台

7.1.1. 简介

Artisan 是框架自带的命令行接口。

想要查看所有可用的命令，可使用 list 命令：

```
php artisan list
```

7.1.2. 编写命令

除了 Artisan 提供的系统 命令之外，还可以编写自己的命令。

7.1.2.1. 生成命令

要创建一个新命令，您可以使用 Artisan 命令 make:command，该命令会在 app/Console/Commands 目录下创建一个新的命令类。

```
php artisan make:command SendEmails
```

7.1.2.2. 命令结构

命令生成以后，需要填写该类的 signature 和 description 属性，这两个属性在调用 list 显示命令的时候会被用到。

7.1.2.3. 闭包命令

基于闭包的命令和闭包路由之于控制器一样，为以类的方式定义控制台命令提供了可选方案。

7.1.3. 定义期望输入

编写控制台命令的时候，通常通过参数和选项收集用户输入，框架使这项操作变得很方便。

7.1.4. 命令 I/O

7.1.5. 注册命令

7.1.6. 通过代码调用命令

有时候您可能希望在 CLI 之外执行 Artisan 命令，比如，您可能希望在路由或控制器中触发 Artisan 命令。

7.2. 广播

7.2.1. 简介

有很多现代 Web 应用中，Web 套接字（WebSockets）被用于实例实时更新的用户接口。当一些数据在服务器上被更新，通常一条消息通过 Websccket 连接被发送给客户端处理。这为我们提供了一个更强大的、更有效的选择来持续拉取应用的更新。

为帮助您构建这样的应用，框架让通过 WebSocket 连接广播事件变得简单。

7.2.1.1. 配置

应用的所有事件广播配置选项都存放在 config/broadcasting.php 配置文件中。开箱支持多种广播驱动：Pusher、Redis 以及一个服务于本地开发和调试的 log 驱动。此外，还提供一个 null 驱动用于完全禁止事件广播。

7.2.1.2. 驱动预备知识

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

7.2.2. 概念概览

事件广播允许您使用基于驱动的 WebSocket 将服务器端事件广播到客户端 JavaScript 应用。

7.2.3. 定义广播事件

7.2.3.1. 广播名称

默认情况下，框架会使用事件的类名来广播事件。

7.2.3.2. 广播数据

如果某个事件被广播，其所有的 public 属性都会按照事件负载自动序列化和广播，从而允许你从 JavaScript 中访问所有 public 数据。

7.2.3.3. 广播队列

默认情况下，每个广播事件都会被推送到配置文件 queue.php 中指定的默认队列连接对应的默认队列中，您可以通过在事件类上定义一个 broadcastQueue 属性来自定义广播使用的队列。该属性需要指定广播时您想要使用的队列名称。

```
/**
* 事件被推送的队列名称.
*
* @var string
```

```
*/

public $broadcastQueue = 'your-queue-name';
```

如果您想要使用 sync 队列而不是默认的队列驱动来广播事件，可以实现 ShouldBroadcastNow 接口来取代 ShouldBroadcast：

```
<?php

use Illuminate\Contracts\Broadcasting\ShouldBroadcastNow;

class ShippingStatusUpdated implements ShouldBroadcastNow

{

    //

}
```

7.2.3.4. 广播条件

有时候您想要在指定条件为 true 的前提下才广播事件。

7.2.4. 授权频道

私有频道要求您授权当前认证用户可以监听的频道。

7.2.4.1. 定义授权路由

7.2.4.2. 定义授权回调

7.2.4.3. 定义频道类

7.2.5. 广播事件

7.2.6. 接收广播

7.2.7. 存在频道

7.2.8. 客户端事件

7.2.9. 通知

通过配对事件广播和通知，JavaScript 应用可以在事件发生时无需刷新当前页面接收新的通知。

7.3. 缓存

7.3.1. 配置

框架为不同的缓存系统提供了统一的 API。缓存配置位于 config/cache.php。在该文件中你可以指定在应用中默认使用哪个缓存驱动。开箱支持主流的缓存后端如 Memcached 和 Redis 等。

缓存配置文件还包含其他文档化的选项，确认仔细阅读这些选项。

7.3.2. 缓存使用

7.3.2.1. 获取缓存实例

Illuminate\Contracts\Cache\Factory 和 Illumicate\Contracts\Cache\Repository 契约提供了访问缓存服务的方法。

Factory 契约提供了所有访问应用定义的缓存驱动的方法。

Repository 契约通常是应用中 cache 配置文件中指定的默认缓存驱动的一个实现。

7.3.2.2. 从缓存中获取数据

Cache 门面的 get 方法用于从缓存中获取缓存项，如果缓存项不存在，则返回 null。如果需要的话您可以传递第二个参数到 get 方法指定不存在时的自定义默认值。

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

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

您甚至可以传递一个闭包作为默认值，如果缓存项不存在的话闭包的结果将会被返回。

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

Has 方法用于判断缓存项是否存在，如果值为 null 或 false 该方法会返回 false。

7.3.2.3. 在缓存中存储数据

您可以使用 Cache 门面上的 put 方法在缓存中存储数据。

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

如果没有传递缓存时间到 put 方法，则缓存存永久有效。

7.3.2.4. 从缓存中移除数据

您可以使用 Cache 门面上的 forget 方法从缓存中移除项数据。

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

还可以通过设置缓存有效期为 0 或负数来移除缓存项。

7.3.2.5. 原子锁

原子锁允许你对分布式锁进行操作而不必担心竞争条件。

7.3.3. 缓存标签

缓存标签目前不支持 file 或 database 缓存驱动。

7.3.4. 添加自定义缓存驱动

7.3.4.1. 编写驱动

要创建自定义的缓存驱动，首先需要实现 Illuminate\Contracts\Cache\Store 契约，所以，我们的 MongoDB 缓存实现看起来会像这样子。

```
<?php

namespace App\Extensions;

use Illuminate\Contracts\Cache\Store;

class MongoStore implements Store
{

    public function get($key) {}

    public function many(array $keys);

    public function put($key, $value, $seconds) {}

    public function putMany(array $values, $seconds);

    public function increment($key, $value = 1) {}

    public function decrement($key, $value = 1) {}

    public function forever($key, $value) {}

    public function forget($key) {}

    public function flush() {}

    public function getPrefix() {}

}
```

我们只需要使用一个 MongoDB 连接来实现其中的每个方法，实现完成后，我们就可以完成自定义驱动注册。

```
Cache::extend('mongo', function($app) {

    return Cache::repository(new MongoStore);

});
```

注：如果您在担心将自定义缓存驱动代码放到哪，可以在 app 目录下创建一个 Extensions 命名空间。

7.3.4.2. 注册驱动

要通过框架注册自定义的缓存驱动，可以使用 Cache 门面上的 extend 方法。

```
<?php
namespace App\Providers;
use App\Extensions\MongoStore;
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\ServiceProvider;

class CacheServiceProvider extends ServiceProvider
{
    /**
    * Perform post-registration booting of services.
    *
    * @return void
    * @translator laravelacademy.org
    */
    public function boot()
    {
        Cache::extend('mongo', function($app) {
            return Cache::repository(new MongoStore);
        });
    }
    /**
    * Register bindings in the container.
    *
```

```
    * @return void
    */
    public function register()
    {
        //
    }
}
```

传递给 extend 方法的第一个参数是驱动名称。

7.3.5. 缓存事件

要在每次缓存操作时执行代码，您可以监听缓存触发的事件，通常，您可以将这些缓存处理程序代码到 EventServiceProvider 中。

```
/**
* The event listener mappings for the application.
*
* @var array
*/
protected $listen = [
    'Illuminate\Cache\Events\CacheHit' => [
    'App\Listeners\LogCacheHit',
],
'Illuminate\Cache\Events\CacheMissed' => [
    'App\Listeners\LogCacheMissed',
],
'Illuminate\Cache\Events\KeyForgotten' => [
```

```
    'App\Listeners\LogKeyForgotten',
],
'Illuminate\Cache\Events\KeyWritten' => [
    'App\Listeners\LogKeyWritten',
],
];
```

7.4. 集合

7.4.1. 简介

Illuminate\Support\Collection 类为处理数组数据提供了流式、方便的封装。例如，查看下面的源码，我们使用辅助函数 collect 创建一个新的集合实例，为每一个元素运行 strtoupper 函数，然后移除所有空元素。

```
$collection = collect(['taylor', 'abigail', null])->map(function ($name)
{
    return strtoupper($name);
    })->reject(function ($name) {
    return empty($name);
});
```

正如您所看到的，Collection 类允许您使用方法链对底层数组执行匹配和移除操作。

7.4.1.1. 创建集合

```
    $collection = collect([1, 2, 3]);
```

7.4.1.2. 扩展集合

```
    use Illuminate\Support\Str;
```

```
Collection::macro('toUpper', function () {
return $this->map(function ($value) {
return Str::upper($value);
});
});
$collection = collect(['first', 'second']);
$upper = $collection->toUpper();
// ['FIRST', 'SECOND']
```

7.4.2. 集合方法

所有这些方法都可以以方法链的方式流式操作底层数组。

7.4.3. 方法列表

All 方法返回集合表示的底层数组。

Avg 方法返回所有集合项的平均值。

7.4.4. 高阶消息传递

集合还支持“高阶消息传递”，也就是在集合上执行通用的功能，支持高阶消息传递的方法包括。

7.4.5. 懒集合

为了继续完善功能已经很强大的 Collection 类，LazyCollection 类使用了 PHP 的生成器，从而可以通过极低的内存处理极大的数据集。

7.5. 事件

7.5.1. 简介

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

7.5.2. 注册事件/监听器

7.5.3. 定义事件

7.5.4. 定义监听器

7.5.5. 事件监听器队列

7.5.6. 分发事件

7.5.7. 事件订阅者

7.5.7.1. 编写事件订阅者

事件订阅者是指那些在类本身中订阅多个事件的类，通过事件订阅者您可以在单个类中定义多个事件处理器。

7.5.7.2. 注册事件订阅者

编写好订阅者后，就可以通过事件分发器对订阅者进行注册，您可以使用 EventServiceProvider 提供的$subcribe 属性来注册订阅者。

7.6. 文件存储

7.6.1. 简介

框架基于 Frank de Jonge 开发的 PHP 包 Flysystem 提供了强大的文件系统 抽象层。Laravel 集成 Flysystem 以便使用不同驱动简化对文件系统的操作。此外，这些存储选项之间切换非常简单，因为对不同系统而言，API 是一致的。

7.6.2. 配置

文件系统配置文件位于 config/filesystems.php。在该文件中可以配置所有“”“磁盘”，每个磁盘描述了特定的存储驱动和存储路径。

7.6.3. 获取硬盘实例

我们可以使用 Storage 门面和上面配置的任意磁盘进行交互。

7.6.4. 获取文件

7.6.5. 存储文件

7.6.6. 删除文件

7.6.7. 目录

7.6.8. 自定义文件系统

框架的 Flysystem 集成提供了多外“驱动”，不过 Flysystem 的功能并不止步于此，还为许多其他存储系统提供了适配器。

7.7. 辅助函数

7.7.1. 简介

框架自带了一系列 PHP 辅助函数，很多被框架自身使用，如果您觉得方便的话也可以在代码中使用它们。

7.7.2. 方法列表

Arr::add 方法添加给定键值对到数组。

7.8. 邮件

7.8.1. 简介

框架基于 SwiftMailer 库提供了一套干净、清爽的邮件 API。

7.8.2. 生成可邮寄类

在框架中，应用发送的每一封邮件都可以表示为“可邮寄”类，这些类都存放在 app/Mail 目录。

7.8.3. 编写可邮寄类

所有的可邮寄类配置都在 build 方法中完成，在这个方法中，您可以调用多个方法，例如 from、subject、view 和 attach 来配置邮件的内容和发送。

7.8.3.1. 配置发件人

首先，我们来看一下邮件发件人的配置。

7.8.3.2. 配置视图

您可以在可邮寄类的 build 方法中使用 virw 方法来指定渲染邮件内容时使用哪个视图模板，由于每封邮件通常使用 Blade 模板来渲染内容，所以您可以在构建邮件 HTML 时使用 Blade 模板引擎提供的所有功能。

7.8.3.3. 视图数据

通常，我们需要传递一些数据到渲染邮件的 HTML 视图以供使用。有两种方式将数据传递到视图，第一种是可邮寄类的公共（public）属性在视图中自动生效。

7.8.3.4. 附件

要添加附件以邮件，可以在可邮寄类的 build 方法中使用 attach 方法。

7.8.3.5. 内联附件

要嵌入内联图片，在邮件视图中使用$message 变量上的 embed 方法即可。

7.8.3.6. 自定义 SwiftMailer 消息

Mailable 基类上的 withSwiftMessage 方法允许您注册一个在发送消息之前可以被原生 SwiftMailer 消息实例调用的回调。

7.8.4. Markdonw 可邮寄类

Maikdown 邮件消息允许您在可邮寄类中利用预置模板和邮件通知组件。因为这些消息以 Markdown 格式编写，框架还可以为它们渲染出高颜值、响应式的 HTML 模板，同时自动生成纯文本的副本。

7.8.5. 发送邮件

要发送一条信息，可以使用 Mail 门面上的 to 方法。To 方法接收邮箱地址、用户实例或用户集合作为参数。如果传递的是对象或对象集合，在设置邮件收件人的时候邮件会自动使用它们的 email 和 name 属性。

7.8.6. 渲染可邮寄类

7.8.7. 本地化可邮寄类

7.8.8. 邮件&本地开发

7.8.9. 事件

框架会在发送邮件消息前触发两个事件，MessageSending 事件在消息发送前触发，MessageSent 事件在消息发送后触发。

7.9. 通知

7.9.1. 简介

除了支持发送邮件外，框架还支持通过多种传播通道发送通知，这些通道包括邮件、短信以及 Slack 等。

7.9.2. 创建通知

7.9.3. 发送通知

7.9.4. 邮件通知

7.9.5. Markdown 邮件通知

7.9.6. 数据库通知

7.9.7. 广播通知

7.9.8. 短信

7.9.9. Slack 通知

7.10. 扩展包开发

7.10.1. 简介

扩展包是添加额外功能到框架的主要方式。

7.10.2. 包自动发现

7.10.3. 服务提供者

服务提供者是扩展包和框架之间的连接纽带。服务提供者负责绑定对象到框架的服务容器并告知从哪里加载包资源，如视图、配置和本地化文件。

7.10.4. 资源

7.10.4.1.配置

通常，需要发布的扩展包配置文件到应用根目录下的 config 目录。

7.10.4.2.路由

如果扩展包包含路由，可以使用 loadRoutesFrom 方法加载它们，这个方法会自动判定应用的路由是否被缓存，如果路由已经被缓存将不会加载路由文件。

```
/**
* Perform post-registration booting of services.
*
* @return void
*/
public function boot()
{
$this->loadRoutesFrom(__DIR__.'/routes.php');
}
```

7.10.4.3.迁移

如果您的扩展包包含数据库迁移，可以使用 LoadMigrationsFrom 方法告知框架如何加载它们。

7.10.4.4.翻译

如果您的扩展包包含翻译文件，您可以使用 LoadTranslationsFrom 方法告诉框架如何加载它们。

7.10.4.5.视图

如果您想要发布扩展包翻译到应用的 resources/lang/verdor 目录，可以使用服务提供者提供的 publishes 方法。

7.10.5. 命令

7.10.6. 发布前端资源

7.10.7. 发布文件组

7.11. 队列

7.11.1. 简介

队列为不同的后台队列服务提供了统一的 API。队列的目的是将耗时的任务延时处理，比如发送邮件，从而大幅度缩短 Web 请求和响应的时间。

队列配置文件存在 config/queue.php。

7.11.2. 创建任务

7.11.2.1.生成任务类

通常，所有的任务类都保存在 app/Jobs 目录。

7.11.2.2.任务类结构

7.11.2.3.任务中间件

7.11.3. 分发任务

7.11.4. 队列闭包

7.11.5. 运行队列

7.11.6. 配置

7.11.7. 处理失败

7.12. 任务调度

7.12.1. 简介

Cron 是 Unix、Solaris、Linux 下的一个十分有用的工具，通过 Cron 脚本能使计划任务定期地在系统后台自动运行。Crontab 则是用来记录在特定时间运行的 Cron 的一个脚本文件，Crontab 文件的每一行均遵守特定的格式。

任务调度定义在 app/Console/Kernel.php 文件的 schedule 方法中。

7.12.2. 定义调度

7.12.2.1.调度 Artisan 命令

可以调度 Artisan 命令和操作系统 命令。

```
$schedule->command('emails:send --force')->daily();

$schedule->command(EmailsCommand::class, ['--force'])->daily();
```

7.12.2.2.调度队列任务

Job 方法可用于调度一个队列任务。

```
$schedule->job(new Heartbeat)->everyFiveMinutes();

// Dispatch the job to the "heartbeats" queue...

$schedule->job(new Heartbeat, 'heartbeats')->everyFiveMinutes();
```

7.12.2.3.调度 Shell 命令

exec 方法可用于调用操作系统 命令。

```
$schedule->exec('node /home/forge/script.js')->daily();
```

7.12.2.4.调度常用选项

7.12.2.5.时区

7.12.2.6.避免任务重叠

7.12.2.7.在单台服务器上运行任务

7.12.2.8.后台任务

7.12.2.9.维护模式

7.12.3. 任务输出

调度器为处理调度任务输出提供了多个方便的方法。

```
$schedule->command('emails:send')

    ->daily()

    ->sendOutputTo($filePath);
```

7.12.4. 任务钩子

使用 before 和 after 方法，您可以指定在调度任务完成之前和之后要执行的代码。

第八章 数据库操作

8.1. 快速入门

8.1.1. 简介

框架让连接不同数据库以及对数据库进行增删改查操作变得非常简单，无论使用原生 SQL、还是查询构建器，还是 Eloquent ORM。

目前，框架支持四种类型的数据库系统。

MySQL、Postgres、SQLite、SQL Server。

8.1.1.1. 配置

应用的数据库配置位于 config/database.php。

默认情况下，使用 MySQL 作为数据库引擎，并且示例配置已经为 Laravel Homestead 环境做好了配置。

8.1.1.2. 读写分离

有时候您希望使用一个数据库连接做查询，另一个数据库连接做插入、更新和删除，框架中实现这种读写分离非常简单。

```
'mysql' => [
    'read' => [
    'host' => '192.168.1.1'
    ],
    'write' => [
    'host' => '192.168.1.2'
    ],
    'sticky' => true,
    'driver' => 'mysql',
    'database' => 'database',
```

```
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8mb4',
    'collation' => 'utf8mb4_unicode_ci',
    'prefix' => '',
],
```

8.1.1.3. 使用不同数据库连接

使用多个数据库连接的时候，可以通过 DB 门面上的 connection 方法访问不同连接。传递给 connecton 方法的 name 对应配置文件 config/database.php 中设置的某个连接。

```
$users = DB::connection('read')->select(...);
```

甚至还可以指定数据库和连接名，使用：：分隔。

8.1.2. 运行原生 SQL 查询

配置好数据库连接后，就可以使用 DB 门面来运行查询。DB 门面为每种操作提供了相应方法。select, update, insert, delete 和 statement。

8.1.3. 数据库事务

想要在一个数据库事务中运行一连串操作，可以使用 DB 门面的 transaction 方法，使用 transaction 方法时不需要手动回滚或提交，如果事务闭包中抛出异常，事务将会自动回滚；如果闭包执行成功，事件将会自动提交。

```
DB::transaction(function () {
    DB::table('users')->update(['votes' => 1]);
    DB::table('posts')->delete();
});
```

8.1.3.1. 处理库死锁

数据库死锁指的有两个或两个以上数据库操作相互依赖，一方需要等待另一方退出才能获取资源，但是没有一方提前退出，就会造成死锁，数据库事务容易造成的一个副作用就是死锁。为此 transaction 方法接收一个可选参数作为第二个参数，用于定义死锁发生时事务的最大重试次数。

```
DB::transaction(function () {
    DB::table('users')->update(['votes' => 1]);
    DB::table('posts')->delete();
}, 5);
```

8.1.3.2. 手动处理事务

如果您想要手动开启事务从而对回滚和提交有更好的控制，可以使用 DB 门面的 beginTransaction 方法。

```
    DB::beginTransaction();
```

您可以通过 rollBack 方法回滚事务。

```
    DB::rollBack();
```

最后，您可以通过 commit 方法提交事务。

```
    DB::commit();
```

8.2. 查询构建器

8.2.1. 简介

数据库查询构建器提供了一个方便的流接口用于创建和执行数据库查询。查询构建器可以用于执行应用中绝大部分数据库操作，并且能够在框架支持的所有数据库系统上工作。

8.2.2. 获取结果集

我们可以从 DB 门面的 table 方法开始，table 方法为给定表返回一个流式查询构建器实例，该实例允许你在查询上链接多个约束条件并返回最终查询结果。

8.2.3. 查询（Select）

我们并不总是想要获取数据表的所有有列，使用 select 方法，您可以为查询指定自定义的 select 子句。

```
$users = DB::table('users')->select('name', 'email as

user_email')->get();
```

8.2.4. 原生表达式

有时候您希望在查询中使用原生表达式，这些表达式将会以字符串的形式注入到查询中，所以要格外小心避免 SQL 注入。创建原生表达式，可以使用 DB::raw 方法。

```
$users = DB::table('users')

    ->select(DB::raw('count(*) as user_count, status'))

    ->where('status', '<>', 1)

    ->groupBy('status')

    ->get();
```

8.2.4.1. 原生方法

除了使用 DB::raw 外，您还可以使用以下方法来插入原生表达式到查询的不同部分。

8.2.4.1.1. sekectRaw

8.2.5. 连接（Join）

查询构建器还可以用于编写连接语句，关于 SQL 的几种连接类型，通过下图可以一目了然。

8.2.5.1. 内连接

要实现一个简单的“内连接”，您可以使用查询构建器实例上的 join 方法，传递给 join 方法的第一个参数是您需要连接到的表名，剩余的其他参数则是为连接指定的列约束，当然，正如您所看到的，您可以在单个查询中连接多张表。

```
$users = DB::table('users')
    ->join('contacts', 'users.id', '=', 'contacts.user_id')
    ->join('orders', 'users.id', '=', 'orders.user_id')
    ->select('users.*', 'contacts.phone', 'orders.price')
    ->get();
```

8.2.5.2. 左连接/右连接

如果您想要执行左连接或右连接，而不是内连接，可以使用 leftJoin 或 rightJoin 方法。这些方法和 join 方法的用法一样。

```
$users = DB::table('users')
    ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
    ->get();
$users = DB::table('users')
    ->rightJoin('posts', 'users.id', '=', 'posts.user_id')
    ->get();
```

8.2.5.3. 交叉连接

要执行交叉连接可以使用 crossJoin 方法，传递您想要交叉连接的表名到该方法即可。交叉连连接在第一张表和被连接表之间生成一个笛卡尔积。

```
$users = DB::table('sizes')
   ->crossJoin('colours')
   ->get();
```

8.2.5.4. 高级连接语句

您还可以指定更多的高级连接子句，传递一个闭包到 join 方法作为第二个参数，该闭包将会接收一个 JoinClause 对象用于指定 join 子句约束。

```
DB::table('users')
   ->join('contacts', function ($join) {
      $join->on('users.id', '=', 'contacts.user_id')
      ->orOn(...);
   })
   ->get();
```

8.2.5.5. 子查询连接

您可以使用 joinSub、leftJoinSub 和 rigthJoinSub 方法将查询和一个子查询进行连接，每个方法都接收三个参数——子查询、表别名和定义关联字段的闭包。

```
$latestPosts = DB::table('posts')
      ->select('user_id', DB::raw('MAX(created_at) as
last_post_created_at'))
      ->where('is_published', true)
```

```
        ->groupBy('user_id');
$users = DB::table('users')
        ->joinSub($latestPosts, 'latest_posts', function($join) {
            $join->on('users.id', '=', 'latest_posts.user_id');
        })->get();
```

8.2.6. 联合（Union）

查询构建器还提供了联合两个查询的快捷方式，比如，您可以先创建一个查询，然后使用 union 方法将其和第二个查询进行联合。

```
$first = DB::table('users')
    ->whereNull('first_name');
$users = DB::table('users')
    ->whereNull('last_name')
    ->union($first)
    ->get();
```

注：unionAll 方法也是有效的，并且和 union 使用方式相同。

8.2.7. Where 子句

8.2.7.1. 简单 Where 子句

使用查询构建器上的 where 方法可以添加 where 子句到查询中，调用 where 最基本的方式需要传递三个参数，第一个参数是列名，第二个参数是任意一个数据库系统支持的操作符，第三个参数是该列要比较的值。

```
$users = DB::table('users')->where('votes', '=', 100)
->get();
```

8.2.7.2. Or 语句

您可以通过方法链将多个 where 约束链接到一起，也可以添加 or 子句到查询，orWhere 方法和 where 方法接收参数一样。

```
$users = DB::table('users')
    ->where('votes', '>', 100)
    ->orWhere('name', 'John')
    ->get();
```

8.2.7.3. 更多 where 子句

8.2.7.4. 参数分组

8.2.7.5. Where exists 子句

8.2.8. 排序、分组

8.2.8.1. orderBy

8.2.9. 条件子句

有时候您可能想要某些条件为 true 的时候才将条件子句应用到查询。

8.3. 分页

8.3.1. 简介

在其他框架中，分页可能是件非常痛苦的事，但此框架让这件事变得简单。

8.3.2. 基本使用

8.3.3. 显示分页视图

8.3.4. 自定义分页视图

8.3.5. 分页器实例方法

每个分页器实例都可以通过以下方法提供更多分页信息。

8.4. 迁移

8.4.1. 简介

所谓迁移就是数据库的版本控制，这种机制允许团队简单轻松的编辑并共享应用的数据库表结构。

8.4.2. 生成迁移

8.4.3. 迁移结构

8.4.4. 运行迁移

8.4.5. 数据表

8.4.6. 数据表

8.4.7. 索引

8.5. 数据填充

8.5.1. 简介

填充类提供了一个简单方法来填充测试数据到数据库。

8.5.2. 编写填充器

8.5.3. 运行填充器

8.6. Redis

8.6.1. 简介

Redis 经常被当作数据结构服务器，因为其支持字符串、Hash、列表、集合和有序集合等数据结构。

8.6.2. 与 Redis 交互

8.6.3. 发布/订阅

第九章 Eloquent ORM

9.1. 快速入门

9.1.1. 简介

框架内置的 Eloquent ORM 提供了一个美观、简单的与数据库打交道的 ActiveRecord 实现，每张数据表都对应一个与该表进行交互的模型（Model），通过模型类，您可以对数据表进行查询、插入、更新、删除等操作。

9.1.2. 定义模型

我们从创建一个 Eloquent 模型开始，模型通常位于 app 目录下。所有 Eloquent 模型都继承自 Illuminate\Database\Eloquent 类。

9.1.2.1. Eloquent 模型约定

现在，让我们来看一个 Flight 模型的例子，我们将用该类获取和存取数据表 flights 中的信息。

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model

{
```

```
//
}
```

注意我们并没有告诉 Eloquent 我们的 Flight 模型使用哪张表，默认规则是小写的模型类名复数格式作为与其对应的表名。所以，在本例中，Eloquent 认为 Flight 模型存储记录在 flights 表中。您也可以在模型中定义 table 属性来指定自定义的类名。

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Flight extends Model
{
    /**
    * 关联到模型的数据表
    *
    * @var string
    */
    protected $table = 'my_flights';
}
```

Eloquent 默认每张表的主键名为 id，您可以在模型类中定义一个$primaryKey 属性来覆盖该约定。

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Flight extends Model
```

```
{
    /**
    * The primary key associated with the table.
    *
     * @var string
    */
    protected $primaryKey = 'flight_id';
}
```

此外，Eloquent 默认主键字段是自增的整形数据，这意味着主键将会被自动转化为 int 类型，如果您想要使用非自增或非数字类型主键，必须在对应模型中设置$incrementing 属性为 false。

```
<?php
class Flight extends Model
{
    /**
    * Indicates if the IDs are auto-incrementing.
    *
    * @var bool
    */
    public $incrementing = false;
}
```

如果主键不是整形，还要设置$keyType 属性值为 string.

```
<?php
class Flight extends Model
```

```
{
    /**
    * The "type" of the auto-incrementing ID.
    *
    * @var string
    */
    protected $keyType = 'string';
}
```

默认情况下，Eloquent 期望 created_at 和 updated_at 已经存在于数据表中，如果您不想要这些框架自动管理的数据列，在模型类中设置$timestamps 属性为 false。

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Flight extends Model
{
    /**
    * 表明模型是否应该被打上时间戳
    *
    * @var bool
    */
    public $timestamps = false;
}
```

如果您需要自定义用于存储时间戳的字段名称，可以在模型中设置

CREATED_AT 和 UPDATED_AT 常量。

```
<?php

class Flight extends Model

{

    const CREATED_AT = 'creation_date';

    const UPDATED_AT = 'last_update';

}
```

9.1.2.2. 默认属性值

如果您想要定义某些属性的默认值，可以在模型上定义 $attributes 属性：

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model

{

    /**

    * The model's default values for attributes.

    *

    * @var array

    */

    protected $attributes = [

    'delayed' => false,

    ];

}
```

9.1.3. 获取模型

创建完模型及其关联的数据表后，就可以从数据库中获取数据了。将 Eloquent 模型看作功能强大的查询构建器，您可以使用它来流畅地查询与其关联的数据表。例如：

```
<?php

use App\Flight;

$flights = App\Flight::all();

foreach ($flights as $flight) {

    echo $flight->name;

}
```

Eloquent 的 all 方法返回模型表的所有结果，由于每一个 Eloquent 模型都是一个查询构建器，您还可以添加约束条件到查询，然后使用 get 方法获取对应结果。

```
$flights = App\Flight::where('active', 1)

    ->orderBy('name', 'desc')

    ->take(10)

    ->get();
```

9.1.3.1. 集合

对 Eloquent 中获取多个结果的方法（比如 all 和 get）而言，其返回值是 Illuminate\Database\Eloquent\Collection 的一个实全，Collection 类提供了多外有用的函数来处理 Eloquent 结果集。

```
$flights = $flights->reject(function ($flight) {

    return $flight->cancelled;

});
```

当然，您也可以像数组一样循环遍历该集合。

9.1.3.2. 组块结果集

如果您需要处理数据量很大的 Eloquent 结果集，可以全用 chunk 方法。Chuck 方法会获取一个指定数量的 Eloquent 模型组块，并将其填充到给定闭包进行处理。使用 chuck 方法在处理大量数据集合时能够有效减少内存消耗。

```
Flight::chunk(200, function ($flights) {
    foreach ($flights as $flight) {
    //
    }
});
```

传递给该方法的第一个参数是你想要获取的“组块”数目，闭包作为第二个参数被传入用于处理每个从数据库获取的组块数据。

9.1.3.3. 高级子查询

Eloquent 还提供了高级子查询支持，该特征允许你从关联数据表中拉取信息到单个查询。例如，假设我们有一个航班目的地表 destinations 和一个飞向这些目的地的航班表 flights，flights 包含用于记录航班到达目的地时间的 arrived_at 字段。

```
use App\Destination;
use App\Flight;
return Destination::addSelect(['last_flight' => Fligh
t::select('name')
->whereColumn('destination_id', 'destinations.id
')
->orderBy('arrived_at', 'desc')
->limit(1)
```

```
])->get();
```

9.1.4. 获取单个模型/聚合结果

当然，除了从给定表中获取所有记录之外，还可以使用 find 和 first 获取单个记录。这些方法返回单个模型实例而不是模型集合。

```
// 通过主键获取模型...
$flight = App\Flight::find(1);
// 获取匹配查询条件的第一个模型...
$flight = App\Flight::where('active', 1)->first();
```

还可以通用传递主键数组还调用 find 方法。

```
$flights = App\Flight::find([1, 2, 3]);
```

9.1.4.1. 获取聚合结果

当然，您还可以使用查询构建器提供的聚合方法，例如 count、sum、max，以及其他查询构建器提供的聚合函数。这些方法返回计算后的结果而不是整个模型实例。

```
$count = App\Flight::where('active', 1)->count();
$max = App\Flight::where('active', 1)->max('price');
```

9.1.5. 插入/更新模型

9.1.5.1. 插入

想要在数据库中插入新的记录，只需创建一个新的模型实例，设置模型的属性，然后调用 save 方法。

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

```
use App\Flight;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;
class FlightController extends Controller{
    /**
    * 创建一个新的航班实例
    *
    * @param Request $request
    * @return Response
    */
    public function store(Request $request)
    {
        // 验证请求...
        $flight = new Flight;
        $flight->name = $request->name;
        $flight->save();
    }
}
```

在这个例子中,我们只是简单分配 HTTP 请求中的 name 参数值给 App\Flight 模型实例的 name 属性，当我们调用 save 方法时，一条记发将会被插入数据库。

9.1.5.2. 更新

Save 方法还可以用于更新数据库中已存在的模型。要更新一个模型，应该先获取它，设置你想要更新的属性，然后调用 save 方法。

```
$flight = App\Flight::find(1);
```

```
$flight->name = 'New Flight Name';

$flight->save();
```

9.1.5.3. 批量赋值

9.1.5.4. 其他创建方法

还有其它两种可以用来创建模型的方法：firstOrCreate 和 firstOrNew。

9.1.6. 删除模型

要删除一个模型，调用模型实例上的 delete 方法。

```
$flight = App\Flight::find(1);

$flight->delete();
```

如果您知道模型的主键，可以调用 destroy 方法直接删除而不需要获取它。

9.1.7. 查询作用域

9.1.7.1. 全局作用域

全局作用域允许我们为给定模型的所有查询添加条件约束。

9.1.7.2. 本地作用域

本地作用域允许我们定义通用的约束集合以便在应用中复用。

9.1.8. 比较模型

有时候您可能需要确定两个模型是否是一样的。

```
if ($post->is($anotherPost)) {

//

}
```

9.1.9. 事件

Eloquent 模型可以触发事件，允许您在模型生命周期中的多个时间点调用如下这

些方法：retrieved, creating, created, updating, updated, saving, saved, deleting, deleted, restoring, restored。

9.2. 关联关系

9.2.1. 简介

数据表经常要与其它表做关联，比如一篇文章可能有很多评论，或者一个订单会被关联到下单用户，Eloquent 让组织和处理这些关联关系变得简单，并且支持多种不同类型的关联关系。

一对一

一对多

多对多

9.2.2. 定义关联关系

9.2.3. 多态关系

9.2.4. 关联查询

9.2.5. 渴求式加载

9.2.6. 插入&更新关联模型

9.2.7. 触发父模型时间戳更新

9.3. 集合

9.3.1. 简介

Eloquent 返回的包含多条记录的结果集都是 Illuminate\Database\Eloquent\Collectoin 对象的实例。

9.3.2. 可用方法

9.3.3. 自定义集合

9.4. 访问器和修改器

9.4.1. 简介

9.4.2. 访问器&修改器

9.4.3. 日期修改器

9.4.4. 属性转换

9.5. API 资源类

9.5.1. 简介

构建 API 时，在 Eloquent 模型和最终返回给应用用户的 JSON 响应之间可能需要一个转化层。

9.5.2. 生成资源类

9.5.3. 核心概念

9.5.4. 编写资源类

9.5.5. 资源响应

9.6. 序列化

9.6.1. 简介

当构建 JSON API 时，经常需要转化模型和关联关系为数组或 JSON。

9.6.2. 序列化模型&集合

9.6.3. 在 JSON 中隐藏属性

9.6.4. 追加值到 JSON

9.6.5. 日期序列化

第十章 测试系列

10.1. 快速入门

10.1.1. 简介

内置的 PHPUnit 对测试提供支持是开箱即用的。

10.2. HTTP 测试

10.3. 控制台测试

10.4. 浏览器测试

10.5. 数据库测试

10.6. 模拟

第十一章 附录：官方扩展包

11.1. 订阅支付解决方案

11.2. 远程操作解决方案

11.3. 队列系统解决方案

11.4. API 认证解决方案

11.5. 全文搜索解决方案

11.6. 第三方登录解决方案

11.7. 模拟本地开发调试解决方案

第十二章 补充说明

暂无。

第十三章 示例 AAA

13.1. 【示例：111】

子系统总体功能介绍。

13.1.1.【示例 1-标准版：222】

【模块的大体描述】

实现对 XXX 的基本管理。

13.1.1.1.【333】

【模块的大体描述】

XXX 的基本管理。

13.1.1.1.1. 【444】

这里需要对【XXX】这个功能做主要的功能概述。

13.1.1.1.2. 【555】

列表涉及到的基本功能（新增、编辑、删除、启用停用）、查询条件也在这里说明。

13.1.1.1.2.1. 【666】

字段说明为必须，需要描述。