1、简介

Laravel 应用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好的起点，当然，你可以自己按照喜好重新组织应用目录结构，Laravel 对类在何处被加载没有任何限制——只要 Composer 可以自动载入它们即可。

手册：http://laravelacademy.org/laravel-docs-5\_4

安装：

· PHP版本 >= 5.6.4

· PHP扩展：OpenSSL

· PHP扩展：PDO

· PHP扩展：Mbstring

可将libeay32.dll和ssleay32.dll文件复制并替换到apache\bin目录下

2、目录结构

新安装的 Laravel 应用包含许多文件夹：

app

目录包含了应用的核心代码；

bootstrap

目录包含了少许文件用于框架的启动和自动载入配置，还有一个cache文件夹用于包含框架生成的启动文件以提高性能；

config

目录包含了应用所有的配置文件；

database

目录包含了数据迁移及填充文件，如果你喜欢的话还可以将其作为 SQLite 数据库存放目录；

public

目录包含了前端控制器和资源文件（图片、JavaScript、CSS等）；

resources

目录包含了视图文件及原生资源文件（LESS、SASS、CoffeeScript），以及本地化文件；

storage

目录包含了编译过的Blade模板、基于文件的session、文件缓存，以及其它由框架生成的文件，该文件夹被细分为成app、framework和logs子目录，app目录用于存放应用要使用的文件，framework目录用于存放框架生成的文件和缓存，最后，logs目录包含应用的日志文件；

tests

目录包含自动化测试，其中已经提供了一个开箱即用的PHPUnit示例；

vendor

目录包含Composer依赖；

3、App目录

应用的核心代码位于app目录下，默认情况下，该目录位于命名空间 App 下， 并且被 Composer 通过 PSR-4自动载入标准 自动加载。你可以通过Artisan命令app:name来修改该命名空间。

app目录下包含多个子目录，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布命令的方式。Console目录包含了所有的Artisan命令，Http目录包含了控制器、中间件和请求等。

Jobs

目录是放置队列任务的地方，应用中的任务可以被队列化，也可以在当前请求生命周期内同步执行。

Events

目录是放置事件类的地方，事件可以用于通知应用其它部分给定的动作已经发生，并提供灵活的解耦的处理。

Listeners

目录包含事件的处理器类，处理器接收一个事件并提供对该事件发生后的响应逻辑，比如，UserRegistered事件可以被SendWelcomeEmail监听器处理。

Exceptions

目录包含应用的异常处理器，同时还是处理应用抛出的任何异常的好地方。

注意：app目录中的很多类都可以通过Artisan命令生成，要查看所有有效的命令，可以在终端中运行php artisan list make命令。

基本上就这么多了，可以开干！

控制器编写

路由

Route::get($uri, $callback|控制器名@方法名);

Route::post($uri, $callback|控制器名@方法名);

有时候还需要注册路由响应多个 HTTP 请求——这可以通过 match 方法来实现。或者，可以使用 any 方法注册一个路由来响应所有 HTTP 请求：

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

//

});

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

//

});

路由参数

必选参数

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

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

return 'User '.$id;

});

可以按需要在路由中定义多个路由参数：

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

//

});

路由参数总是通过花括号进行包裹，这些参数在路由被执行时会被传递到路由的闭包。路由参数不能包含 - 字符，需要的话可以使用 \_ 替代。

可选参数

有时候可能需要指定可选的路由参数，这可以通过在参数名后加一个 ? 标记来实现，这种情况下需要给相应的变量指定默认值：

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

return $name;

});

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

return $name;

});

正则约束

可以使用路由实例上的 where 方法来约束路由参数的格式。where 方法接收参数名和一个正则表达式来定义该参数如何被约束：

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

//

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

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

//

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

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

//

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

路由群组

路由群组允许我们在多个路由中共享路由属性，比如中间件和命名空间等，这样的话我们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。

中间件

要给路由群组中定义的所有路由分配中间件，可以在群组属性数组中使用 middleware。中间件将会按照数组中定义的顺序依次执行：

Route::group(['middleware' => 'auth'], function () {

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

// 使用 Auth 中间件

});

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

// 使用 Auth 中间件

});

});

命名空间

另一个通用的例子是路由群组分配同一个 PHP 命名空间给其下的多个控制器，可以在分组属性数组中使用 namespace 来指定群组中所有控制器的公共命名空间：

Route::group(['namespace' => 'Admin'], function(){

// 控制器在 "App\Http\Controllers\Admin" 命名空间下

});

默认情况下，RouteServiceProvider 引入你的路由文件并指定其下所有控制器类所在的默认命名空间App\Http\Controllers，因此，我们在定义的时候只需要指定命名空间 App\Http\Controllers 之后的部分即可。

子域名路由

路由群组还可以被用于子域名路由通配符，子域名可以像 URI 一样被分配给路由参数，从而允许捕获子域名的部分用于路由或者控制器，子域名可以通过群组属性数组中的 domain 来指定：

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

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

//

});

});

路由前缀

群组属性 prefix 可以用来为群组中每个路由添加一个给定 URI 前缀，例如，你可以为所有路由 URI 添加 admin 前缀 ：

Route::group(['prefix' => 'admin'], function () {

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

// 匹配 "/admin/users" URL

});

});

控制器编写：

下面是一个基本控制器类的例子。所有的 Laravel 控制器应该继承自 Laravel 自带的控制器基类 Controller，控制器基类提供了一些很方便的方法如 middleware ，用于添加中间件到控制器动作：

<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

class NewsController extends Controller{

function detail($id){

//echo "文章详细页，$id";

$arr=['id'=>1,'title'=>'t','content'=>'c'];

return view("news.detail",['arr'=>$arr]); }

}

单动作控制器

如果你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

{

/\*\*

\* Show the profile for the given user.

\*

\* @param int $id

\* @return Response

\*/

public function \_\_invoke($id)

{

return view('user.profile', ['user' => User::findOrFail($id)]);

}

}

当你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');

中间件的定义：

要创建一个新的中间件，可以通过 Artisan 命令 make:middleware：

php artisan make:middleware CheckAge

这个命令会在 app/Http/Middleware 目录下创建一个新的中间件类 CheckAge，在这个中间件中，我们只允许提供的 age 大于 200 的请求访问路由，否则，我们将用户重定向到 home：

<?php

namespace App\Http\Middleware;

use Closure;

class CheckAge

{

/\*\*

\* 返回请求过滤器

\*

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

\* @param \Closure $next

\* @return mixed

\*/

public function handle($request, Closure $next)

{

if ($request->input('age') <= 200) {

return redirect('home');

}

return $next($request);

}

}

正如你所看到的，如果 age <= 200，中间件会返回一个 HTTP 重定向到客户端；否则，请求会被传递下去。将请求往下传递可以通过调用回调函数 $next 并传入 $request。

理解中间件的最好方式就是将中间件看做 HTTP 请求到达目标动作之前必须经过的“层”，每一层都会检查请求并且可以完全拒绝它。

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');

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

class UserController extends Controller

{

/\*\*

\* 实例化一个新的 UserController 实例

\*

\* @return void

\*/

public function \_\_construct()

{

$this->middleware('auth');

$this->middleware('log')->only('index');

$this->middleware('subscribed')->except('store');

}

}

在控制器中还可以使用闭包注册中间件，这为我们定义只在某个控制器使用的中间件提供了方便，无需定义完整的中间件类：

$this->middleware(function ($request, $next) {

// ...

return $next($request);

});

注：你还可以将中间件分配给多个控制器动作，不过，这意味着你的控制器会变得越来越臃肿，这种情况下，需要考虑将控制器分割成多个更小的控制器。

提示跳转：

函数 redirect(“路由”)

前后台分离：

在App/Http/Controllers/Admin/XxxController.php

namespace App\Http\Controllers\Admin;

use App\Http\Controllers\Controller;

class NewsController extends Controller{

function \_\_construct(){

$this->middleware('checklogin');

}

function add(){

echo "添加文章";

}

function oper(){

echo "管理文章";

}

function save(){

echo "保存数据";

}

}

使用中间件控制用户访问权限：

使用分组路由和前缀路由实现地址规则：

Route::group(['prefix'=> 'admin','namespace' => 'Admin'], function(){

// 控制器在 "App\Http\Controllers\Admin" 命名空间下

Route::get('login/{message?}','UserController@login');

Route::get('news/add','NewsController@add');

Route::get('news/oper','NewsController@oper');

Route::post('news/save','NewsController@save');

Route::get('product/add','ProductController@add');

Route::get('product/oper','ProductController@oper');

Route::post('product/save','ProductController@save');

});

Blade模板：

简介

Blade 是 Laravel 提供的一个非常简单但很强大的模板引擎，不同于其他流行的 PHP 模板引擎，Blade 在视图中并不约束你使用 PHP 原生代码。所有的 Blade 视图都会被编译成原生 PHP 代码并缓存起来直到被修改，这意味着对应用的性能而言 Blade 基本上是零开销。Blade 视图文件使用 .blade.php 文件扩展并存放在 resources/views 目录下。

数据显示

可以通过两个花括号包裹变量来显示传递到视图的数据，比如，如果给出如下路由：

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

return view('welcome', ['name' => 'Samantha']);

});

那么可以通过如下方式显示 name 变量的内容：

Hello, {{ $name }}.

当然，不限制显示到视图中的变量内容，你还可以输出任何 PHP 函数的结果，实际上，可以将任何 PHP 代码放到 Blade 模板语句中：

The current UNIX timestamp is {{ time() }}.

注：Blade 的 {{}} 语句已经经过 PHP 的 htmlentities 函数处理以避免 XSS 攻击。

输出存在的数据

有时候你想要输出一个变量，但是不确定该变量是否被设置，我们可以通过如下 PHP 代码：

{{ isset($name) ? $name : 'Default' }}

除了使用三元运算符，Blade 还提供了更简单的方式：

{{ $name or 'Default' }}

在本例中，如果 $name 变量存在，其值将会显示，否则将会显示 Default。

显示原生数据

默认情况下，Blade 的 {{ }} 语句已经通过 PHP 的 htmlentities 函数处理以避免 XSS 攻击，如果你不想要数据被处理，可以使用如下语法：

Hello, {!! $name !!}.

流程控制

除了模板继承和数据显示之外，Blade 还为常用的 PHP 流程控制提供了便利操作，例如条件语句和循环，这些快捷操作提供了一个干净、简单的方式来处理 PHP 的流程控制，同时保持和 PHP 相应语句的相似性。

If 语句

可以使用 @if , @elseif , @else 和 @endif 来构造 if 语句，这些指令的功能和 PHP 相同：

@if (count($records) === 1)

I have one record!

@elseif (count($records) > 1)

I have multiple records!

@else

I don't have any records!

@endif

为方便起见，Blade 还提供了 @unless 指令：

@unless (Auth::check())

You are not signed in.

@endunless

循环

除了条件语句，Blade 还提供了简单指令处理 PHP 支持的循环结构，同样，这些指令的功能和 PHP 一样：

@for ($i = 0; $i < 10; $i++)

The current value is {{ $i }}

@endfor

@foreach ($users as $user)

<p>This is user {{ $user->id }}</p>

@endforeach

@forelse ($users as $user)

<li>{{ $user->name }}</li>

@empty

<p>No users</p>

@endforelse

@while (true)

<p>I'm looping forever.</p>

@endwhile

注：在循环的时候可以使用 $loop 变量获取循环信息，例如是否是循环的第一个或最后一个迭代。

使用循环的时候还可以结束循环或跳出当前迭代：

@foreach ($users as $user)

@if ($user->type == 1)

@continue

@endif

<li>{{ $user->name }}</li>

@if ($user->number == 5)

@break

@endif

@endforeach

$loop变量

在循环的时候，可以在循环体中使用 $loop 变量，该变量提供了一些有用的信息，比如当前循环索引，以及当前循环是不是第一个或最后一个迭代：

@foreach ($users as $user)

@if ($loop->first)

This is the first iteration.

@endif

@if ($loop->last)

This is the last iteration.

@endif

<p>This is user {{ $user->id }}</p>

@endforeach

注释

Blade 还允许你在视图中定义注释，然而，不同于 HTML 注释，Blade 注释并不会包含到 HTML 中被返回：

{{-- This comment will not be present in the rendered HTML --}}

PHP

在一些场景中，嵌入 PHP 代码到视图中很有用，你可以使用 @php 指令在模板中执行一段原生 PHP 代码：

@php

//

@endphp

尽管 Blade 提供了这个特性，如果过于频繁地使用它意味着你在视图模板中嵌入了过多的业务逻辑，需要注意。

6、包含子视图

Blade 的 @include 指令允许你很轻松地在一个视图中包含另一个 Blade 视图，所有父级视图中变量在被包含的子视图中依然有效：

<div>

@include('shared.errors')

<form>

<!-- Form Contents -->

</form>

</div>

尽管被包含的视图可以继承所有父视图中的数据，你还可以传递额外参数到被包含的视图：

@include('view.name', ['some' => 'data'])

数据库操作：

链接数据库修改配置文件

config/database.php

'mysql' => [

'driver' => 'mysql',

'host' => env('DB\_HOST', '127.0.0.1'),

'port' => env('DB\_PORT', '3306'),

'database' => env('DB\_DATABASE', 'forge'),

'username' => env('DB\_USERNAME', 'forge'),

'password' => env('DB\_PASSWORD', ''),

'charset' => 'utf8mb4',

'collation' => 'utf8mb4\_unicode\_ci',

'prefix' => '',

'strict' => true,

'engine' => null,

],

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。

运行 Select 查询

运行一个最基本的查询，可以使用 DB 门面的select 方法：

<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\DB;

use App\Http\Controllers\Controller;

class UserController extends Controller

{

/\*\*

\* 展示应用的用户列表.

\*

\* @return Response

\*/

public function index()

{

$users = DB::select('select \* from users where active = ?', [1]);

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

}

}

传递给 select 方法的第一个参数是原生的SQL语句，第二个参数需要绑定到查询的参数绑定，通常，这些都是 where 子句约束中的值。参数绑定可以避免 SQL 注入攻击。

select 方法以数组的形式返回结果集，数组中的每一个结果都是一个 PHP StdClass 对象，从而允许你像下面这样访问结果值：

foreach ($users as $user) {

echo $user->name;

}

使用命名绑定

除了使用 ? 占位符来代表参数绑定外，还可以使用命名绑定来执行查询：

$results = DB::select('select \* from users where id = :id', ['id' => 1]);

运行插入语句

使用 DB 门面的 insert 方法执行插入语句。和 select 一样，该方法将原生 SQL 语句作为第一个参数，将绑定作为第二个参数：

DB::insert('insert into users (id, name) values (?, ?)', [1, 'Dayle']);

运行更新语句

update 方法用于更新数据库中已存在的记录，该方法返回受更新语句影响的行数：

$affected = DB::update('update users set votes = 100 where name = ?', ['John']);

运行删除语句

delete 方法用于删除数据库中已存在的记录，和 update 一样，该语句返回被删除的行数：

$deleted = DB::delete('delete from users');

运行一个通用语句

有些数据库语句不返回任何值，对于这种类型的操作，可以使用 DB 门面的 statement 方法：

DB::statement('drop table users');

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，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，事务将会自动提交。使用transaction 方法时不需要手动回滚或提交：

DB::transaction(function () {

DB::table('users')->update(['votes' => 1]);

DB::table('posts')->delete();

});

处理死锁

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。如果尝试次数超出指定值，会抛出异常：

DB::transaction(function () {

DB::table('users')->update(['votes' => 1]);

DB::table('posts')->delete();

}, 5);

手动使用事务

如果你想要手动开始事务从而对回滚和提交有一个完整的控制，可以使用 DB 门面的beginTransaction 方法：

DB::beginTransaction();

你可以通过 rollBack 方法回滚事务：

DB::rollBack();

最后，你可以通过 commit 方法提交事务：

DB::commit();

获取结果集

从一张表中取出所有行

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最终返回查询结果。在本例中，我们使用 get 方法获取表中所有记录：

<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\DB;

use App\Http\Controllers\Controller;

class UserController extends Controller{

/\*\*

\* 显示用户列表

\*

\* @return Response

\*/

public function index()

{

$users = DB::table('users')->get();

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

}

}

get 方法返回包含结果集的 Illuminate\Support\Collection，其中每一个结果都是 PHP 的 StdClass 对象实例。你可以像访问对象的属性一样访问字段的值：

foreach ($users as $user) {

echo $user->name;

}

从一张表中获取一行/一列

如果你只是想要从数据表中获取一行数据，可以使用 first 方法，该方法将会返回单个 StdClass 对象：

$user = DB::table('users')->where('name', 'John')->first();

echo $user->name;

如果你不需要完整的一行，可以使用 value 方法从结果中获取单个值，该方法会直接返回指定列的值：

$email = DB::table('users')->where('name', 'John')->value('email');

获取数据列值列表

如果想要获取包含单个列值的数组，可以使用pluck 方法，在本例中，我们获取角色标题数组：

$titles = DB::table('roles')->pluck('title');

foreach ($titles as $title) {

echo $title;

}

还可以在返回数组中为列值指定自定义键（该自定义键必须是该表的其它字段列名，否则会报错）：

$roles = DB::table('roles')->pluck('title', 'name');

foreach ($roles as $name => $title) {

echo $title;

}

组块结果集

如果你需要处理成千上百条数据库记录，可以考虑使用 chunk 方法，该方法一次获取结果集的一小块，然后传递每一小块数据到闭包函数进行处理，该方法在编写处理大量数据库记录的 Artisan 命令的时候非常有用。比如，我们可以将处理全部 users 表数据分割成一次处理 100 条记录的小组块：

DB::table('users')->orderBy('id')->chunk(100, function($users) {

foreach ($users as $user) {

//

}

});

你可以通过从闭包函数中返回 false 来终止组块的运行：

DB::table('users')->orderBy('id')->chunk(100, function($users) {

// 处理结果集...

return false;

});

聚合函数

查询构建器还提供了多个聚合方法，如count, max, min, avg 和 sum，你可以在构造查询之后调用这些方法：

$users = DB::table('users')->count();

$price = DB::table('orders')->max('price');

当然，你可以联合其它查询子句和聚合函数来构建查询：

$price = DB::table('orders')

->where('finalized', 1)

->avg('price');

3、查询（Select）

指定查询子句

当然，我们并不总是想要获取数据表的所有列，使用 select 方法，你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();

distinct 方法允许你强制查询返回不重复的结果集：

$users = DB::table('users')->distinct()->get();

如果你已经有了一个查询构建器实例并且希望添加一个查询列到已存在的 select 子句，可以使用 addSelect 方法：

$query = DB::table('users')->select('name');

$users = $query->addSelect('age')->get();

4、原生表达式

有时候你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。想要创建一个原生表达式，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();

5、连接（Join）

内连接（等值连接）

查询构建器还可以用于编写基本的 SQL “内连接”，你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你需要连接到的表名，剩余的其它参数则是为连接指定的列约束，当然，正如你所看到的，你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();

左连接

如果你是想要执行“左连接”而不是“内连接”，可以使用 leftJoin 方法。该方法和 join 方法的用法一样：

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

->leftJoin('posts', 'users.id', '=', 'posts.user\_id')

->get();

交叉连接

要执行“交叉连接”可以使用 crossJoin 方法，传递你想要交叉连接的表名到该方法即可。交叉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();

高级连接语句

你还可以指定更多的高级连接子句，传递一个闭包到 join 方法作为该方法的第二个参数，该闭包将会返回允许你指定 join 子句约束的 JoinClause 对象：

DB::table('users')

->join('contacts', function ($join) {

$join->on('users.id', '=', 'contacts.user\_id')->orOn(...);

})

->get();

如果你想要在连接中使用“where”风格的子句，可以在查询中使用 where 和orWhere 方法。这些方法将会将列和值进行比较而不是列和列进行比较：

DB::table('users')

->join('contacts', function ($join) {

$join->on('users.id', '=', 'contacts.user\_id')

->where('contacts.user\_id', '>', 5);

})

->get();

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，你可以先创建一个查询，然后使用 union 方法将其和第二个查询进行联合：

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

->whereNull('first\_name');

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

->whereNull('last\_name')

->union($first)

->get();

注：unionAll 方法也是有效的，并且和 union 有同样的使用方式。

7、Where子句

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 最基本的方法需要传递三个参数，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。

例如，下面是一个验证“votes”列的值是否等于 100 的查询：

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

为了方便，如果你只是简单比较列值和给定数值是否相等，可以将数值直接作为where方法的第二个参数：

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

当然，你还可以使用其它操作符来编写where 子句：

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

->where('votes', '>=', 100)

->get();

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

->where('votes', '<>', 100)

->get();

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

->where('name', 'like', 'T%')

->get();

还可以传递条件数组到 where 函数：

$users = DB::table('users')->where([

['status', '=', '1'],

['subscribed', '<>', '1'],

])->get();

or语句

你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();

更多Where子句

whereBetween

whereBetween 方法验证列值是否在给定值之间：

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

->whereBetween('votes', [1, 100])->get();

whereNotBetween

whereNotBetween 方法验证列值不在给定值之间：

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

->whereNotBetween('votes', [1, 100])

->get();

whereIn/whereNotIn

whereIn 方法验证给定列的值是否在给定数组中：

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

->whereIn('id', [1, 2, 3])

->get();

whereNotIn 方法验证给定列的值不在给定数组中：

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

->whereNotIn('id', [1, 2, 3])

->get();

whereNull/whereNotNull

whereNull 方法验证给定列的值为NULL：

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

->whereNull('updated\_at')

->get();

whereNotNull 方法验证给定列的值不是 NULL：

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

->whereNotNull('updated\_at')

->get();

whereDate / whereMonth / whereDay / whereYear

whereDate 方法用于比较字段值和日期：

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

->whereDate('created\_at', '2016-10-10')

->get();

whereMonth 方法用于比较字段值和一年中的指定月份：

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

->whereMonth('created\_at', '10')

->get();

whereDay 方法用于比较字段值和一月中的制定天：

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

->whereDay('created\_at', '10')

->get();

whereYear 方法用于比较字段值和指定年：

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

->whereYear('created\_at', '2016')

->get();

whereColumn

whereColumn 方法用于验证两个字段是否相等：

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

->whereColumn('first\_name', 'last\_name')

->get();

还可以传递一个比较运算符到该方法：

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

->whereColumn('updated\_at', '>', 'created\_at')

->get();

还可以传递多条件数组到 whereColumn 方法，这些条件通过 and 操作符进行连接：

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

->whereColumn([

['first\_name', '=', 'last\_name'],

['updated\_at', '>', 'created\_at']

])->get();

参数分组

有时候你需要创建更加高级的 where 子句，比如“where exists”或者嵌套的参数分组。Laravel 查询构建器也可以处理这些。作为开始，让我们看一个在括号中进行分组约束的例子：

DB::table('users')

->where('name', '=', 'John')

->orWhere(function ($query) {

$query->where('votes', '>', 100)

->where('title', '<>', 'Admin');

})

->get();

正如你所看到的，传递闭包到 orWhere方法构造查询构建器来开始一个约束分组，该闭包将会获取一个用于设置括号中包含的约束的查询构建器实例。上述语句等价于下面的 SQL：

select \* from users where name = 'John' or (votes > 100 and title <> 'Admin')

where exists子句

whereExists 方法允许你编写 where exists SQL子句，whereExists 方法接收一个闭包参数，该闭包获取一个查询构建器实例从而允许你定义放置在“exists”子句中的查询：

DB::table('users')

->whereExists(function ($query) {

$query->select(DB::raw(1))

->from('orders')

->whereRaw('orders.user\_id = users.id');

})

->get();

上述查询等价于下面的 SQL 语句：

select \* from users

where exists (

select 1 from orders where orders.user\_id = users.id

)

JSON Where子句

Laravel 还支持在提供 JSON 字段类型的数据库（目前是 MySQL 5.7 和 Postgres）上使用操作符 -> 查询 JSON 字段类型：

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

->where('options->language', 'en')

->get();

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

->where('preferences->dining->meal', 'salad')

->get();

8、排序、分组、限定

orderBy

orderBy 方法允许你通过给定字段对结果集进行排序，orderBy 的第一个参数应该是你希望排序的字段，第二个参数控制着排序的方向——asc或desc：

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

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

->get();

latest / oldest

latest 和 oldest 方法允许你通过日期对结果进行排序，默认情况下，结果集通过 created\_at 字段进行排序，或者，你可以你想要排序的字段作为字段名传入：

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

->latest()

->first();

inRandomOrder

inRandomOrder 方法可用于对查询结果集进行随机排序，比如，你可以用该方法获取一个随机用户：

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

->inRandomOrder()

->first();

groupBy / having / havingRaw

groupBy 和 having 方法用于对结果集进行分组，having 方法和 where 方法的用法类似：

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

->groupBy('account\_id')

->having('account\_id', '>', 100)

->get();

havingRaw 方法可用于设置原生字符串作为 having 子句的值，例如，我们可以这样找到所有售价大于 $2,500 的部分：

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

->select('department', DB::raw('SUM(price) as total\_sales'))

->groupBy('department')

->havingRaw('SUM(price) > 2500')

->get();

skip / take

想要限定查询返回的结果集的数目，或者在查询中跳过给定数目的结果，可以使用skip 和 take 方法：

$users = DB::table('users')->skip(10)->take(5)->get();

作为替代方法，还可以使用 limit 和offset 方法：

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

->offset(10)

->limit(5)

->get();

9、条件子句

有时候你可能想要某些条件为 true 的时候才将条件子句应用到查询。例如，你可能只想给定值在请求中存在的情况下才应用where 语句，这可以通过 when 方法实现：

$role = $request->input('role');

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

->when($role, function ($query) use ($role) {

return $query->where('role\_id', $role);

})

->get();

when 方法只有在第一个参数为 true 的时候才执行给定闭包，如果第一个参数为 false，则闭包不执行。

你可以传递另一个闭包作为 when 方法的第三个参数，该闭包会在第一个参数为false 的情况下执行。为了演示这个特性如何使用，我们来配置一个查询的默认排序：

$sortBy = null;

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

->when($sortBy, function ($query) use ($sortBy) {

return $query->orderBy($sortBy);

}, function ($query) {

return $query->orderBy('name');

})

->get();

10、插入（Insert）

查询构建器还提供了 insert 方法用于插入记录到数据表。insert 方法接收数组形式的字段名和字段值进行插入操作：

DB::table('users')->insert(

['email' => 'john@example.com', 'votes' => 0]

);

你甚至可以一次性通过传入多个数组来插入多条记录，每个数组代表要插入数据表的记录：

DB::table('users')->insert([

['email' => 'taylor@example.com', 'votes' => 0],

['email' => 'dayle@example.com', 'votes' => 0]

]);

自增ID

如果数据表有自增ID，使用 insertGetId 方法来插入记录并返回ID值：

$id = DB::table('users')->insertGetId(

['email' => 'john@example.com', 'votes' => 0]

);

注：当使用 PostgresSQL 时insertGetId 方法默认自增列被命名为 id，如果你想要从其他“序列”获取ID，可以将序列名作为第二个参数传递到insertGetId 方法。

11、更新（Update）

当然，除了插入记录到数据库，查询构建器还可以通过使用 update 方法更新已有记录。update 方法和 insert 方法一样，接收字段名和字段值的键值对数组包含要更新的列，你可以通过 where 子句来对 update 查询进行约束：

DB::table('users')

->where('id', 1)

->update(['votes' => 1]);

更新JSON字段

更新 JSON 字段的时候，需要使用 -> 语法访问 JSON 对象上相应的值，该操作只能用于支持 JSON 字段类型的数据库：

DB::table('users')

->where('id', 1)

->update(['options->enabled' => true]);

增加/减少

查询构建器还为增减给定字段名对应数值提供方便。相较于编写 update 语句，这是一条捷径，提供了更好的体验和测试接口。

这两个方法都至少接收一个参数：需要修改的列。第二个参数是可选的，用于控制列值增加/减少的数目。

DB::table('users')->increment('votes');

DB::table('users')->increment('votes', 5);

DB::table('users')->decrement('votes');

DB::table('users')->decrement('votes', 5);

在操作过程中你还可以指定额外的列进行更新：

DB::table('users')->increment('votes', 1, ['name' => 'John']);

12、删除（Delete）

当然，查询构建器还可以通过 delete 方法从表中删除记录，你可以在调用 delete 方法前通过添加 where 子句来添加约束条件：

DB::table('users')->delete();

DB::table('users')->where('votes', '>', 100)->delete();

如果你希望清除整张表，也就是删除所有列并将自增ID置为0，可以使用 truncate 方法：

DB::table('users')->truncate();

分页：

简单分页

如果你只需要在分页视图中简单的显示“下一页”和“上一页”链接，可以使用 simplePaginate 方法来执行一个更加高效的查询。在渲染包含大数据集的视图且不需要显示每个页码时这一功能非常有用：

$users = DB::table('users')->simplePaginate(15);

基于Eloquent结果集进行分页

你还可以对 Eloquent 查询结果进行分页，在本例中，我们对 User 模型进行分页，每页显示 15 条记录。正如你所看到的，该语法和基于查询构建器的分页差不多：

$users = App\User::paginate(15);

当然，你可以在设置其它约束条件之后调用paginate，比如 where 子句：

$users = User::where('votes', '>', 100)->paginate(15);

在对 Eloquent 模型进行分页时你也可以使用simplePaginate 方法：

$users = User::where('votes', '>', 100)->simplePaginate(15);

手动创建分页器

有时候你可能想要通过传递数组数据来手动创建分页实例，你可以基于自己的需求通过创建 Illuminate\Pagination\Paginator 或 Illuminate\Pagination\LengthAwarePaginator 实例来实现。

Paginator 类不需要知道结果集中数据项的总数；不过，正因如此，该类也没有提供获取最后一页索引的方法。LengthAwarePaginator 接收参数和 Paginator 几乎一样，只是，它要求传入结果集的总数。

换句话说，Paginator 对应 simplePaginate 方法，而LengthAwarePaginator 对应 paginate 方法。

注：当手动创建分页器实例的时候，应该手动对传递到分页器的结果集进行“切片”，如果你不确定怎么做，查看 PHP 函数 array\_slice。

3、显示分页结果

当调用 paginate 方法时，你将获取Illuminate\Pagination\LengthAwarePaginator 实例，而调用方法simplePaginate 时，将会获取 Illuminate\Pagination\Paginator 实例。这些对象提供相关方法描述这些结果集，除了这些辅助函数外，分页器实例本身就是迭代器，可以像数组一样对其进行循环调用。所以，获取到结果后，可以按如下方式使用 Blade 显示这些结果并渲染页面链接：

<div class="container">

@foreach ($users as $user)

{{ $user->name }}

@endforeach

</div>

{{ $users->links() }}

验证码

<?php

namespace App\Http\Controllers\Extension;

use Illuminate\Http\Request;

use App\Http\Controllers\Controller;

use Gregwar\Captcha\CaptchaBuilder;

use Illuminate\Support\Facades\Session;

class CaptchaController extends Controller

{

/\*\*

\* Display a listing of the resource.

\*

\* @return \Illuminate\Http\Response

\*/

public function captcha($random)

{

header('Content-type: image/jpeg');

header("Cache-Control: no-cache, must-revalidate");

$builder = new CaptchaBuilder();

$builder->setBackgroundColor(255, 255, 255)->build(120,28);

session()->put('captcha', $builder->getPhrase());

$builder->output();

}

/\*\*

\* Show the form for creating a new resource.

\*

\* @return \Illuminate\Http\Response

\*/

public function index()

{

return view('extension/captcha/index');

}

public function test()

{

return view('extension/captcha/test');

}

/\*\*

\* Store a newly created resource in storage.

\*

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

\* @return \Illuminate\Http\Response

\*/

public function post(Request $request)

{

if ($request->captcha == session()->get('captcha')) {

return '注册成功';

} else {

return '验证码错误';

}

}

}

显示验证码：

<img src="{{url('extension/captcha/captcha') . '/' . mt\_rand()}}" alt="" class="captcha" title="看不表，换一张">

验证：

public function post(Request $request)

{

if ($request->captcha == session()->get('captcha')) {

return '注册成功';

} else {

return '验证码错误';

}

}

文件上传：

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Controllers\Controller;

class NewsController extends Controller{

function \_\_construct(){

//$this->middleware('checklogin');

}

function detail($id){

//echo "文章详细页，$id";

$arr=['id'=>1,'title'=>'t','content'=>'c'];

return view("news.detail",['arr'=>$arr]);

}

function upload(Request $request){

if($request->isMethod('post')){

$files=$request->file('upload');

if($files){

foreach($files as $ob){

if($ob->isValid()){//文件上传成功

echo $ob->getClientOriginalName(),"<br/>";

echo $ob->getClientOriginalExtension(),"<br/>";

echo $ob->getClientMimeType(),"<br/>";

echo $ob->getClientSize(),"<br/>";

//$fileName="Y-m-d-H-i-s-".uniqid().".".$ob->getClientOriginalExtension();

$re=$ob->store('','uploads');

var\_dump($re);

}else{

echo $ob->getError();

}

}

}

}else{

return view("upload");

}

}

}