# Laravel笔记

## 安装

#### composer安装

composer create-project --prefer-dist laravel/laravel blog "5.2.\*"

## 配置

#### 使用config()获取你的设置值

设置值可以用[点]语法来获取：文件名.选项名称；

$value = config(‘app.timezone’);// show “UTC”

若要在运行期间修改设置，需要给config()传一个数组；

config([‘app.timezone’ => ‘Amercia/Chicago’]);

#### 维护模式

当你的应用处于维护模式时，所有传递至应用程序的请求都会显示出一个自定义视图。

启用维护模式：php artisan down

关闭维护模式：php artisan up

维护模式的默认模板在：resources/views/errors/503.blade.php

#### 数据库迁移

Laravel 的数据库迁移提供了一个简单的方式，使用流畅、一目了然的 PHP 代码来定义数据表的结构与修改。你无需再告诉团队成员要手动增加字段至他们本机的数据库中，他们只需要运行你提交到版本控制的迁移，即可应用数据库修改。

使用Artisan命令：

php artisan make:migration create\_tasks\_table --create=tasks//创建数据库迁移文件，里面等于是建表语句，默认放在database/migrations 目录中。

php artisan migrate //运行迁移，等于执行上面的迁移文件

在创建模型时自动创建迁移文件

php artisan make:model Test –m

#### 数据填充

单个填充

php artisan make:seeder TestTableSeeder

database/seeds目录下TestTableSeeder的run 方法添加：

DB::table(‘test’)->insert([

//数据

]);

执行插入：

php artisan db:seed --class=TestTableSeeder

批量填充：

在DatabaseSeeder.php的run 方法里引入你创建的Seeder类：$this->call(…);

执行：php artisan db:seed

#### Eloquent 模型

每个 Eloqunet 模型会直接对应一张数据表。

使用make:model命令来生成tasks数据表的模型Task（是个空模型，放在app目录下）：

php artisan make:model Task //默认创建的Task模型对应tasks数据表

## 路由

#### 响应多个HTTP动作的路由

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

//

});

//匹配所有请求方式

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

//

});

#### 路由参数

* + 1. 必选参数

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

//

});

路由参数都放在花括号内，参数不能含‘-’；

* + 1. 可选参数

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

return $name;

});

在参数后面加‘？’表示可选；

* + 1. 正则限制参数

//使用where（）方法，前面是参数名，后面是正则；若同时限制多个参数，采用关联数组方式。

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]+']);

* + 1. 全局限制

若希望路由参数一直遵循正则表达式，在RouteServiceProvider 的 boot 方法里定义这些模式：

public function boot(Router $router)

{

$router->pattern('id', '[0-9]+');

parent::boot($router);

}

模式一旦被定义，便会自动应用到所有使用该参数名称的路由上

* + 1. 命名路由

------//为这个路由指定一个名字；

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

//

}]);

------//也可以为控制器的方法指定路由名字：

Route::get('user/profile', [

'as' => 'profile', 'uses' => 'UserController@showProfile'

]);

------//还可以在路由定义后链式调用name()方法

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

一旦你在指定的路由中分配了名称，则可通过 route 函数来使用路由名称生成 URLs 或重定向：

$url = route('profile');

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

若路由中带参数，可以用 route(‘profile’,[‘id’=>12]); 直接将参数加入到url中；

-------//在路由群组的属性数组中指定 as 关键字，会为群组中所有路由设置相同的前缀

Route::group(['as' => 'admin\_'], function () {

Route::get('dashboard', ['as' => 'dashboard', function () {

// 路由名称为「admin\_dashboard」

return route(‘admin\_dashboard’);

}]);

});

* + 1. 路由前缀

为路由指定的URI加上前缀

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

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

// 符合「/admin/users」URL

});

});

* + 1. 绑定模型

|  |
| --- |
| //路由参数中的{test}默认对应tests表的主键‘id’  Route::any('test/{test}','TestController@test2');  在TestController控制器的将$test实例传入，即可在方法体里直接使用该实例  function test2(App\Models\Test $test){  dd($test);  } |

#### CSRF 保护

（1）

Laravel 会自动生成一个 CSRF token 给每个用户的 Session。该 token 用来验证登录用户和发起请求者是否是同一人。可以使用 csrf\_field 辅助函数来生成一个包含 CSRF token 的 \_token 隐藏表单字段：

<?php echo csrf\_field(); ?>

csrf\_field 辅助函数会生成以下的 HTML：

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

在Blade模板中使用 {{csrf\_field()}}

不需要手动验证请求的csrf token, VerifyCsrfToken HTTP 中间件 将自动验证请求与 session 中的 token 是否相符。

（2）

并非所有请求都需要避免CSRF攻击，比如第三方请求。此时需要在：app/Http/Middleware/VerifyCsrfToken.php中间件中将需要排除的url添加到$except数组中即可

（3）

如果使用Ajax提交POST表单，又该如何处理呢？我们可以将Token设置在meta中：

<meta name="csrf-token" content="{{ csrf\_token() }}">

然后在全局Ajax中使用这种方式设置X-CSRF-Token请求头并提交：

$.ajaxSetup({

headers: {

'X-CSRF-TOKEN': $('meta[name="csrf-token"]').attr('content')

}

});

Laravel的VerifyCsrfToken中间件会检查X-CSRF-TOKEN请求头，如果该值和Session中CSRF值相等则验证通过，否则不通过。

(4)原理：首先在laravel开启session时，会生成一个token放在session中，（ Illuminate\Session\Store.php第90行start方法）

然后看VerifyCsrfToken.php的handle方法，首先是isReading方法判断请求方式，如果请求方法是HEAD、GET、OPTIONS其中一种，则不做CSRF验证；

然后，再通过shouldPassThrough方法判断请求路由是否在$excpet属性数组中进行了排除，如果做了排除也不做验证；

最后通过tokensMatch方法判断请求参数中的CSRF TOKEN值和Session中的Token值是否相等，如果相等则通过验证，否则抛出TokenMismatchException异常。

注：tokensMatch方法首先从Request中获取\_token参数值，如果请求中不包含该参数则获取X-CSRF-TOKEN请求头的值，如果该请求头也不存在则获取X-XSRF-TOKEN请求头的值，需要注意的是X-XSRF-TOKEN请求头的值需要调用Encrypter的decrypt方法进行解密。

#### 路由分组

* + 1. 中间件

生成中间件的命令：

php artisan make:middleware TestMiddleware

而后把中间件添加到/app/Http/Kernal.php文件的$routeMiddleware属性：

protected $routeMiddleware = [

'test' => \App\Http\Middleware\TestMiddleware::class,

];之后即可在路由中使用中间件。

若要让中间件在全局有效，每次请求都会被调用，则将中间件追加到$middleware属性数组（在路由中不用middleware 指明了）

使用：

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

或Route::get('/', ['middleware' => ['first', 'second'], function () {}]);

或Route::get('/', function () {})->middleware(['first', 'second']);

控制器的构造方法中定义：

|  |
| --- |
| public function \_\_construct(){  $this->middleware('auth');  } |

自定义中间件类只需要定义一个handle方法即可，将主要的业务逻辑定义在该方法中。

若想要在请求处理前执行业务逻辑，则在$next闭包执行前执行业务逻辑。

public function handle($request, Closure $next)

{

// 执行业务逻辑操作

return $next($request);

}

若想要在请求处理后执行中间件业务逻辑，则在$next闭包执行后操作

public function handle($request, Closure $next)

{

$response = $next($request);

// 执行动作

return $response;

}

中间件参数：

public function handle($request, Closure $next, $gender)

{

if($request->input('age')>=18 && $gender==$request->input('gender')){

return $next($request);

}else{

return redirect()->route('refuse');

}

}

路由配置：

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

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

//使用Test中间件

});

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

//使用Test中间件

});

});

访问[www.blog.com/se?age=19&gender=male](http://www.blog.com/se?age=19&gender=male)时会通过

* + 1. 命名空间

默认情况下，routes.php中的定义的控制器位于App\Http\Controllers命名空间下，所以如果要指定命名空间，只需指定App\Http\Controllers之后的部分即可：

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

{

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

});

## 缓存

#### 配置

缓存配置位于config/cache.php.可以设置使用哪种缓存驱动。

#### 基础知识

数据库缓存驱动

需要一张表：

Schema::create('cache', function($table) {

$table->string('key')->unique();

$table->text('value');

$table->integer('expiration');

});

* + 1. 缓存使用
       1. 获取缓存实例

use Cache;//使用cache门面

* + - 1. 访问多个缓存存储

$value = Cache::store(‘file’)->get(‘foo’);

Cache::store(‘redis’)->put(‘bar’,’baz’,10);

* + - 1. 从缓存中获取数据

$value = Cache::get('key');//若缓存项不存在，返回null

$value = Cache::get('key', 'default');//若有需要，缓存项不存在时，返回第二个参数

$value = Cache::get(‘key’,function(){//缓存项不存在，返回闭包的的结果

return DB::table(…)->get();

})

Cache::has('key')；//检查缓存项是否存在，返回true or false;

Cache::increment('key', $amount);//将缓存项数值增加$amount;

Cache::decrement(‘key’,$amount);//将缓存项数值减少$amount;

获取或更新：若缓存项存在，则获取，若不存在，给他存上一个值，并返回；

例如：从缓存获取用户信息，如果不存在的话，从数据库获取他们并将其添加到缓存中；

$value = Cache::remember('users', $minutes, function() {

return DB::table('users')->get();});

获取并删除：  
 $value = Cache::pull('key');//若不存在，返回null

* + - 1. 存储缓存项到缓存

使用put方法存储缓存项；需要制定缓存时间，分钟数

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

add方法只在缓存项不存在时候添加，添加成功返回true;失败返回false;

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

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

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

* + - 1. 从缓存中移除数据

Cache::forget('key');

* + - 1. 添加自定义的缓存驱动

|  |
| --- |
| <?php  namespace App\Providers;  use Cache;  use App\Extensions\MongoStore;//调用缓存驱动  use Illuminate\Support\ServiceProvider;  class CacheServiceProvider extends ServiceProvider{  public function boot()  {  Cache::extend('mongo', function($app) {  return Cache::repository(new MongoStore);  });  }  public function register()  {  //  }  } |

extend()第一个参数是驱动名，对应config/cache.php中的driver选项。第二个参数是返回Illuminate\Cache\Repository实例的闭包。该闭包中被传入一个$app实例，也就是服务容器的一个实例。

调用Cache::extend可以在默认App\Providers\AppServiceProvider中的boot方法中完成，或者你也可以创建自己的服务提供者来存放该扩展（像上面例子）——只是不要忘了在配置文件config/app.php中注册该提供者。

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缓存实现看起来像这样子：

|  |
| --- |
| <?php  namespace App\Extensions;//存放位置，自由  class MongoStore implements \Illuminate\Contracts\Cache\Store{  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() {}  public function getPrefix() {}  } |

最后在config/cache.php中添加你的配置即可。

* + - 1. 缓存标签

缓存标签允许你给相关的缓存项打上同一个标签，然后可以输出被分配同一个标签的所有缓存值。

Cache::tags(['people', 'artists'])->put('John', $john, $minutes);//带标签的存储

$john = Cache::tags(['people', 'artists'])->get('John');//带标签查询

Cache::tags(['people', 'authors'])->flush();//将带有这两种标签的缓存项移除

## HTTP请求实例

#### 获取request请求

路由：Route::controller('request','RequestController'); //已废弃

在app/Http/Controllers下创建一个控制器RequestController.php：

class RequestController extends Controller

{

public function getBasetest(Request $request)

{

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

echo $input;

}

}

直接访问：<http://www.blog.com/request/basetest?test=laravelacademy>

访问控制器的方法时，前面的get|post不要带，后面的若是大驼峰写法，在两个英文单词之间用‘-’隔开；

#### 获取请求URL及请求方法

* + 1. 获取请求URL

$request->path();//获取host后面的部分，即$\_SERVER[‘REQUEST\_URI’]

$request->url();//获取完整url路径

$request->is(‘request/\*’);//判断request是否匹配request/\*

* + 1. 获取请求方法(GET | POST | PUT…)

$request->method();

$request->isMethod(‘GET | POST | PUT… ’);//判断请求方式是不是括号中的

$request->ajax();//判断是否是ajax请求

$request->has(‘act’)//判断请求是否有act信息(http://www.ydnew.com/page?act=add)

#### 获取请求数据

* + 1. 当前请求输入

$request->input(‘参数名’[，默认值])；//参数名为空，返回默认值

$url = ‘http://www.blog.com/test[0][name]=john’

另：input(‘test.0.name’),支持数组点语法

$request->all();//获取所有输入的参数值，返回一个关联数组

$request->only(‘参数1’，‘参数2’)；//获取部分参数值

$request->except(‘参数1’，’参数2’)//排除部分参数值

或使用帮助函数request()

request（）返回当前请求实例,返回一个对象

request()->all() 返回请求传递过来的数组；

request(‘title’) 或者获取一个输入项

request([‘title’,’content’]) 获取提交过来的两个字段的值

* + 1. 上一次输入请求

若要获取上一次请求的输入，需在上一次请求的时候使用flash（）方法将请求数据暂时保存在session中，在当前请求中使用old()方法获取session中的数据，获取后会将session中保存的请求数据销毁。

|  |
| --- |
| public function getLastRequest(Request $request){  $request->flash();  Return redirect(‘request/current-request’)->withInput();//数据保持，默认把输入的数据全部返回了，在视图中用{{old(‘name’)}}来获取上次输入  }  public function getCurrentRequest(Request $request){  $lastRequestData = $request->old();  echo '<pre>';  print\_r($lastRequestData);  } |

#### 获取COOKIE数据

$request->cookie([‘参数名’])；//接收参数名，返回对应cookie值，若无参数，返回全部cookie值

新增COOKIE：

调用Response实例上的withCookie方法：

|  |
| --- |
| $response = new Response();  //第一个参数是cookie名，第二个参数是cookie值，第三个参数是有效期（分钟）  $response->withCookie(cookie('website','LaravelAcademy.org',1));  //如果想要cookie长期有效使用如下方法  //$response->withCookie(cookie()->forever('name', 'value')); |

## 文件上传

|  |
| --- |
| public function getFileupload()  {  $postUrl = '/request/fileupload';  $csrf\_field = csrf\_field();  $html = <<<CREATE  <form action="$postUrl" method="POST" enctype="multipart/form-data">  $csrf\_field  <input type="file" name="file"><br/><br/>  <input type="submit" value="提交"/>  </form>  CREATE;  return $html;  } |

处理上传文件：

|  |
| --- |
| public function postFileupload(Request $request){  //判断请求中是否包含name=file的上传文件  if(!$request->hasFile('file')){  exit('上传文件为空！');  }  $file = $request->file('file');//获取文件对象  //判断文件上传过程中是否出错  if(!$file->isValid()){  exit('文件上传出错！');  }  $destPath = public\_path('images');//获取public目录下images文件夹的目录  if(!file\_exists($destPath))  mkdir($destPath,0755,true);  $filename = $file->getClientOriginalName();//获得原文件名  $result = $file->move($destPath,$filename);//将文件移动到指定位置  if(empty($result)){  exit('保存文件失败！');  }  exit('文件上传成功！');  } |

文件系统配置在config/filesystems.php；默认使用local本地驱动，所有支持的驱动及驱动详细配置定义在disks配置项中，其中的root项定义了文件存储路径：storage/app。所以默认使用Storage门面的根目录‘/’就是storage/app;

使用Storage门面：

|  |
| --- |
| public function postFileupload(Request $request){  //判断请求中是否包含name=file的上传文件  if(!$request->hasFile('file')){  exit('上传文件为空！');  }  $file = $request->file('file'); //临时文件对象  //判断文件上传过程中是否出错  if(!$file->isValid()){  exit('文件上传出错！');  }  //$file->getClientOriginalExtension() 获取源文件的扩展名；  $newFileName = md5(time().rand(0,10000)).'.'.$file->getClientOriginalExtension();    $bytes = Storage:: disk('local')->put(//指明了config/filesystem.php中使用的驱动，默认使用local  $newFileName,//新文件名，不带路径，存放目录在配置文件disks->local->root里  file\_get\_contents($file->getRealPath())//文件内容  );  if(!Storage::exists($newFileName)){  exit('保存文件失败！');  }  header("Content-Type: ".Storage::mimeType($newFileName));  echo Storage::get($newFileName);  } |

内部相关函数：

$request->hasFile($key) 判断name=$key的input是否有文件

$request->file($key); 获取临时文件对象，里面有属性：originalName，mimeType，size，error，pathName，fileName

$file->isValid() 判断文件上传是否出错

$file->getClientOriginalExtension() 获取源文件的扩展名

Storage::disk(‘local’)->put($newFilename,$data)

$newFilename :新文件名，默认存放目录在配置文件disks->local->root设置的里

$file->getRealPath() 获取临时文件的绝对路径；

file\_get\_contents($file->getRealPath())//获取文件内容

或者File::get($file) 获取文件内容；

Storage::copy(‘test/xxx.txt’,’file/xxx.txt’); //复制文件

Storage::move(‘test/xxx.txt’,’images/xxx.txt’)//移动文件

Storage::delete(‘test/xxx.txt’);删除文件

Storage::files($filePath);//获取$filePath下面的所有文件；（不会返回子目录下的文件）

Storage::allFiles($filePath);// 获取嵌套子目录下的文件

Storage::directories($filePath);//获取子目录

Storage::allDirectories ($filePath);// 获取嵌套的子目录

Storage::makeDirectory($dirPath);//创建目录

Storage::deleteDirectory($dirPath);//递归删除子目录下的所有文件和文件夹；

关于5.3以后的版本的文件上传

|  |
| --- |
| public function update(Request $request)  {  $path = $request->file('avatar')->store('avatars');  或$path = Storage::putFile('avatars', $request->file('avatar'));  return $path;  } |

注意：store（）的参数是目录名，我们并没有指定上传之后的文件名，默认，store方法会基于文件内容自动生成文件名，这通过对文件内容进行MD5实现。store方法会返回文件路径以便在数据库中保存文件路径和文件名(如果你接收的上传文件尺寸很大，你可能需要手动指定文件名，因为计算大文件的MD5哈希值很耗内存。)。

指定文件名：

|  |
| --- |
| //方法接收路径、文件名以及磁盘（可选）作为参数：  $path = $request->file('avatar')->storeAs(  路径 , 文件名  );  或$path = Storage::putFileAs(  路径, $request->file('avatar'), 文件名  ); |

## Response篇

#### 基本响应

Response是继承自Symfony\Component\HttpFoundation\Response的 Illuminate\Http\Response类的一个实例；

|  |
| --- |
| Route::get('testResponse',function(){  $content = 'Hello LaravelAcademy！';  $status = 200;  $value = 'text/html;charset=utf-8';  return (new Response($content,$status))->header('Content-Type',$value);  }); |

为了方便使用，可使用全局帮助函数response来替代生成response对象实例；

|  |
| --- |
| Route::get('testResponse',function(){  $content = 'Hello LaravelAcademy！';  $status = 500;  $value = 'text/html;charset=utf-8';  return response($content,$status)->header('Content-Type',$value);  });//不用生成response实例了，推荐使用。 |

注意：Illuminate\Http\Response 类中还使用了ResponseTrait，该trait提供了header , withCookie , content , status方法。

header ()方法用于设置响应头信息；

返回调用它的response对象，支持方法链

withCookie ()方法用于添加cookie；

content ()方法返回当前响应的响应实体内容

status () 方法返回响应状态码

#### 添加Cookie

withCookie()方法可以设置有效期，作用路径，作用域名，实际调用了全局帮助函数cookie（），然后将cookie放到响应头中。

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

->withCookie('site','LaravelAcademy.org',30,'/','laravel.app');

默认情况，该cookie是经过加密的，你也可以在app/Http/Middleware/EncryptCookie.php文件中将对应的cookie名添加到encryptCookies类属性$except中即可；

## ResponseFactory篇

response函数如果不传入参数会返回 Illuminate\Contracts\Routing\ResponseFactory契约的实现——Illuminate\Routing\ResponseFactory,该类提供更多的响应种类；

response（）无参数时，返回ResponseFactory的实例化对象；有参数时，即向浏览器响应的一个方法

#### 视图响应

|  |
| --- |
| Route::get('testResponseView',function(){  $value = 'text/html;charset=utf-8';  return response()->view('hello',['message'=>'Hello LaravelAcademy'])  ->header('Content-Type',$value);  }); |

若不需要定义header头，可以简化使用全局帮助函数view()；

|  |
| --- |
| return view('hello',['message'=>'Hello ']); |

* 1. 返回JSON/JSONP

|  |
| --- |
| Route::get('testResponseJson',function(){  return response()->json(['name'=>'LaravelAcademy','passwd'=>'LaravelAcademy.org']);  }); |

json方法自动设置content-type为application/json,并调用php内置函数json\_encode.

#### 文件下载

使用download方法。

|  |
| --- |
| Route::get('testResponseDownload',function(){  return response()->download(  realpath(base\_path('public/images')).'/laravel-5-1.jpg', //下载文件路径  'Laravel学院.jpg' //另存为名字  );  }); |

## RedirectResponse篇

重定向响应是Illuminate\Http\RedirectResponse类的实例，通常使用全局帮助函数redirect 来生成redirectResponse实例。

redirect函数若接收参数，调用Illuminate\Routing\Redirector类的to方法；无参数，返回redirect对象实例。

若要重定向到上一个位置，使用back方法

return back()->withInput();//数据保持，默认把输入的数据全部返回了，在视图中用{{old(‘name’)}}来获取上次输入

#### 重定向到命名路由

return redirect()->route(‘路由名称’[，路由参数])；

#### 重定向到控制器动作

|  |
| --- |
| Route::resource('post','PostController');  Route::get('testResponseRedirect',function(){  return redirect()->action('PostController@index');//使用action方法  //return redirect()->action(‘PostController@show’,[1])//可以向show方法里传递参数  }); |

#### 带一次性session数据到重定向

|  |
| --- |
| Route::post('user/profile', function () {  // 更新用户属性...  return redirect('dashboard')->with('status', 'Profile updated!');//redirect()里面是路由；  }); |

一次性session数据是使用后立即销毁的session。

在前端页面可以使用Session::get(‘status’)获取，再次刷新就销毁了。

## 视图

#### 在视图间共享数据

全局函数view，若传入参数，返回Illuminate\View\View实例；没有参数，返回Illuminate\View\Factory实例；

要创建所有的视图中都能使用的变量，可以在app/Providers/AppServiceProvider.php中的boot方法里：

|  |
| --- |
| public function boot()  {  //视图间共享数据  view()->share('sitename','Laravel学院');  //视图Composer（组合器）  //hello是视图名称，可传多个，甚至全部视图 ’\*’  view()->composer('hello',function($view){  $view->with('user',array('name'=>'test','avatar'=>'/path/to/test.jpg'));  });  } |

前端页面调用：

<h3>用户信息</h3>

用户名:{{$user['name']}}<br>

用户头像:{{$user['avatar']}}

## 创建Service Provider实例

#### 定义服务类

先定义一个绑定到容器的测试类TestService：

在app/services下创建文件：

|  |
| --- |
| <?php  namespace App\Services;  class TestService  {  public function callMe($controller)  {  dd('Call Me From TestServiceProvider In '.$controller);  }  } |

#### 创建服务提供者并绑定服务

服务提供者，即将服务注册到服务容器的文件，TestServiceProvider。

用Artisan命令创建：

php artisan make:provider TestServiceProvider

有两种方式绑定服务到服务容器

|  |
| --- |
| <?php  namespace App\Providers;  use Illuminate\Support\ServiceProvider;  use App\Services\TestService;  class TestServiceProvider extends ServiceProvider  {  …  public function register()  {  //使用singleton绑定单例  $this->app->singleton('test',function(){  return new TestService();  });  //使用bind绑定实例到接口以便依赖注入  $this->app->bind('App\Contracts\TestContract',function(){  return new TestService();  });  }  }  解析使用make（） |

#### 注册服务提供者

将服务提供者注册到应用中，追加信息到 config/app.php 的providers数组中即可。

|  |
| --- |
| 'providers' => [  //其他服务提供者  App\Providers\TestServiceProvider::class,  ], |

#### 测试

创建控制器：

php artisan make:controller TestController

|  |
| --- |
| <?php  namespace App\Http\Controllers;  use Illuminate\Support\Facades\App;  class TestController extends Controller  {  //依赖注入  public function \_\_construct(TestContract $test){  $this->test = $test;  }  public function index()  {  //解析使用make  $test = App::make('test');  $test->callMe('TestController');  //$this->test->callMe('TestController');  }  } |

定义路由：

Route::resource('test','TestController');

访问[www.blog.com/test](http://www.blog.com/test),会输出："Call Me From TestServiceProvider In TestController"

#### 容器

容器定义在bootstrap/app.php中，即

|  |
| --- |
| $app = new Illuminate\Foundation\Application(  realpath(\_\_DIR\_\_.'/../')  ); |

可以使用$app = app()获取容器；

从容器中获取里面的类的实例：$log = $app->make(‘log’)//此处的‘log’,跟singleton时的字符串一致

之后可以使用$log->info()等方法。

或者不实例化服务，使用依赖注入

public fucntion test(\Illuminate\Contracts\Logging\Log $log){} //类名可以在Application.php中查找

或者使用门面

public function test(){

\Log::info()

}

## Facades门面

门面提供了一个“静态”接口到服务容器中绑定的类。

对应config/app.php里的alias

首先创建一个需要绑定到服务容器的Test类：

|  |
| --- |
| <?php  namespace App\Facades;  class Test  {  public function doSomething()  {  echo 'This is TestClass\'s method doSomething';  }  } |

然后创建一个静态指向Test的门面类TestClass：

|  |
| --- |
| <?php  namespace App\Facades;  use Illuminate\Support\Facades\Facade;  class TestClass extends Facade  {  protected static function getFacadeAccessor()  {  return 'test';  }  } |

之后在服务提供者中绑定Test类到服务容器，

|  |
| --- |
| <?php  namespace App\Providers;  use Illuminate\Support\ServiceProvider;  use App\Services\TestService;  use App\Facades\Test;  class TestServiceProvider extends ServiceProvider  {  public function register()  {  $this->app->singleton('test',function(){  return new Test;  });  }  } |

在config/app.php中将服务提供者加入到providers数组，并在aliases数组中注册门面别名：

|  |
| --- |
| 'aliases' => [  ...//其他门面类别名映射  'TestClass' => App\Facades\TestClass::class,  ], |

最后修改控制器TestController：

|  |
| --- |
| public function index()  {  TestClass::doSomething();//静态调用Test中的方法  } |

## 数据库

若要修改数据库配置信息，去修改.env对应值即可。

#### 使用DB门面

支持原生sql语句

使用前先use DB；

插入数据：

DB::insert("insert into tests values('2','james','j','2')");

或

DB::insert("insert into tests values(?,?,?,?)",['3','kobe','j','2']);//使用占位符

查询语句：返回数组，数组中每个元素都是对象

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

或

$user = DB::select('select \* from users where id = :id', [':id'=>1]);//参数绑定

更新语句：返回受影响行数

$affected = DB::update('update users set name="LaravelAcademy" where name = ?', ['Academy']);

删除语句：返回受影响行数

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

#### 数据库事务

方法一：

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

方法二：

|  |
| --- |
| DB::beginTransaction();  if($somethingIsFailed){  DB::rollback();  return false;  }  DB::commit(); |

#### 使用 查询构建器 进行增删改查

获取查询构建器：$users = DB::table('users');

插入：

|  |
| --- |
| DB::table('users')->insert([  ['id'=>1,'name'=>'Laravel','email'=>'laravel@test.com','password'=>'123'],  ['id'=>2,'name'=>'Academy','email'=>'academy@test.com','password'=>'123']  ]); //插入两条数组； |

若需要返回自增id：

|  |
| --- |
| $lastId = DB::table('users')-> insertGetId ([  ['name'=>'Laravel','email'=>'laravel@test.com','password'=>'123'],//只能传一句sql  ]); |

更新：返回受影响行数

|  |
| --- |
| $affected=DB::table('users')->where('name','Laravel-Academy')->update(['password'=>'123']);  $affected = DB::table('users')->increment(‘age’,5);//自增5，第二参数不填，默认自增1  decrement同理； |

删除：返回受影响行数

|  |
| --- |
| $deleted = DB::table('users')->where('id', '>', 3)->delete();//where条件很特殊  $deleted = DB::table('users')->delete();删除所有记录  $deleted = DB::table('users')->truncate();删除整个表，并清空id; |

查询：

获取所有记录：

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

获取指定的记录（select指定字段，where指定条件）

$user = DB::table('tests')->select('id','name')->where('id','<',5)->get();//返回索引数组，每个数组只含有id . name字段

获取单条记录：

$user = DB::table('tests')->select('id','name')->where('id','<',5)->first();

多个查询条件查询：

//条件只能是等式；

$user = DB::table('tests')->select('id','name')->where([‘name’=>’ju’,’age’=>18])->first();

//whereRaw()第一个参数是条件语句，第二参数是个数组，填充占位符的

$result = DB::table('tests')->whereRaw('id > ? and user\_id < ?',[40,14])->get();

//还可以多个where条件链式连接

DB::table('tests')->where('id','>',40)->where('user\_id' ,'<',14)->get();

获取单列的值：

区别于select：select取出的是二维数组，第一层是索引数组，第二层是关联数组。

而以下都是一维数组。

$user = DB::table('tests')->pluck(‘name’);同lists，但不能跟第二个参数

$user = DB::table('tests')->lists(‘name’);//返回数组，索引数组，下表从0开始，到n

$user = DB::table('tests')->lists(‘name’,’user\_id’)//第二个参数可以为数组设key值，将user\_id作为key，替换原先的索引下标

分段查询：//当数据较多时，不必要将所有数据全部取出

|  |
| --- |
| DB::table('tests')->chunk(2, function ($res) {//2代表每次取几条数据，$res是每次返回的数据  echo "<pre>";  //if($res[0]->id > 42)  // return false;  print\_r($res);  }); |

当需要获取数组形式的数据时：

|  |
| --- |
| $roles = Role::all()->toArray(); |

获取json格式的数据：

|  |
| --- |
| $roles = Role::all()->toJson(); |

#### 使用查询构建器进行高级查询

* + 1. 连接查询 join

内连接：

|  |
| --- |
| $users = DB::table('users')->join('posts','users.id','=','posts.user\_id')->get();  dd($users); |

左连接：

|  |
| --- |
| $users = DB::table('users')->leftJoin('posts','users.id','=','posts.user\_id')->get();  dd($users); |

左连接的结果集指定左表的所有行，若左表的某行在右表没有匹配行，则此行在结果集中右表的所有选择列表列均为空值null。

更复杂的连接：

|  |
| --- |
| DB：：table(‘tests’)->join(‘users’,function($join){  $join->on(‘tests.user\_id’,’=’,’users.id’)->where(‘users.id’,’>’,1);  })->get(); |

* + 1. 联合查询 union

|  |
| --- |
| $users = DB::table('users')->where('id','<',3);  $users = DB::table('users')->where('id','>',2)->union($users)->get(); |

结果是求并集，将id<3的全部取出，再将id>2的全部取出，然后合并成一个集合。

* + 1. where子句

->where('name’,’=’,’laravel’)->get();

三个参数，若中间参数是‘=’，也可以省略：

->where('name’,’laravel’)->get();

多个where之间是and连接，要是用or连接，使用orWhere方法；

->where(‘name’,’laravel’)->orWhere(‘id’,’=’,2)->get();

* + 1. 排序

orderBy方法进行排序

|  |
| --- |
| $users = DB::table('users')->orderBy('id','desc')->get(); |

* + 1. 分组



|  |
| --- |
| $user=DB::table('tests')->select('cate\_id',DB::raw('COUNT(title) as num'))->groupBy('cate\_id')->having(‘cate\_id’,’<’,3)->get(); |

having条件必须搭配groupBy使用；

DB::raw()内可以写原生sql；

* + 1. 分页

使用skip（从第几条开始）、take（获取几条）组合，相当于sql的limit；

|  |
| --- |
| $user = DB::table('tests')->skip(1)->take(3)->get(); |

|  |
| --- |
| $users = App\User::paginate(15);  前端模板显示：  {{ $users->links() }}  links 方法将会渲染结果集中的其他页链接。这些链接中每一个都已经包含了 ?page 查询字符串变量 |

* + 1. 聚合函数

|  |
| --- |
| $num = DB::table('tests')->count();  $result = DB::table('tests')->max('id');//返回ID最大的ID值  $result = DB::table('tests')->avg('user\_id');//返回平均值  $result = DB::table('tests')->sum('user\_id');//返回和； |

## Eloquent ORM

#### ORM

Object-relational mapping:对象关系映射，在关系型数据库和业务实体对象之间做一个映射，就不用操作SQL语句了，只用操作对象的属性和方法即可。

#### 定义模型

* + 1. 创建模型

模型类默认在app目录下，可以设置子目录。Laravel 中所有模型类继承自Illuminate\Database\Eloquent\Model类。

|  |
| --- |
| php artisan make:model Models/Post |

* + 1. 指定表名

不手动指定的话，默认Post模型对应posts表；也可以手动设置模型的属性：

|  |
| --- |
| public $table = 'posts'; |

* + 1. 指定主键

Eloquent默认数据表主键为id，可以设置$primaryKey来自定义主键

|  |
| --- |
| public $primaryKey = 'id'; |

* + 1. 时间戳设置

默认情况，eloquent模型类会自动管理时间戳列create\_at和update\_at,若要取消，设置属性$timestamps=false;

#### 查询数据

* + 1. 获取多个模型

|  |
| --- |
| $posts = Post::all(); |

输出表中的所有记录，每条记录对应一个Post模型实例；

Eloquent模型本身就是一个查询构建器，可以调用所有查询构建器上的方法，第一个方法要是用静态方法调用

|  |
| --- |
| $posts = Post::where('id','<',3)->orderBy('id','desc')->take(1)->get(); |

* + 1. 获取单个模型

|  |
| --- |
| $post = Post::where('id',1)->first(); |

或快捷方法：

|  |
| --- |
| $post = Post::find(2);// find()内部是id值 |

若无记录，返回null；

若要捕获查询结果为空的异常并处理，可以使用findOrFail或者firstOrFail方法，若记录存在，返回第一条记录，否则抛出异常。

|  |
| --- |
| $posts = Test::findOrFail(2);//需要填参数id |
| $posts = Test::firstOrFail();//获取第一条，无需参数 |

* + 1. 聚合函数查询

|  |
| --- |
| $count = Post::where('id','>',0)->count(); |
| $views = Post::where('id','>',0)->max('views'); |

* + 1. chunk（）分组查询

|  |
| --- |
| Test::chunk(2 , function ($res){  var\_dump($res);//$res是模型实例化对象；  }) |

* + 1. 查询部分字段

|  |
| --- |
| 如果使用DB门面写查询构造器，那只需要链式调用select()方法即可：  $users = DB::table('users')->select('name', 'email as user\_email')->get();  使用Eloquent的话，有两种方式：  1. 使用select()  $users = User::select(['name'])->get();  2. 直接将列名数组作为参数传入all()/get()/find()等方法中  1 $users = User::all(['name']);  2 $admin\_users = User::where('role', 'admin')->get(['id', 'name']);  3 $user = User::find($user\_id, ['name']);    在关联查询中使用同理：  $posts = User::find($user\_id)->posts()->select(['title'])->get();  $posts = User::find($user\_id)->posts()->get(['title', 'description']);  注意这里不能使用动态属性（->posts）来调用关联关系，而需要使用关联关系方法（->posts()）。 |

#### 创建模型

* + 1. save方法创建模型

|  |
| --- |
| $test = new Test;  $test->title = 'kate';  $test->user\_id = 4;  $test->cate\_id = 3;  if($test->save()){  echo '文章添加成功';  }else{  echo '文章添加失败';  } |

* + 1. created\_at时间格式

注意：只有在使用模型插入数据时，才会自动记录created\_at和updated\_at，用DB门面和查询构建器都不行！！

在模型类中定义：

|  |
| --- |
| public $timestamps = true;//开启记录时间功能  //获取当前时间  public function freshTimestamp() {  return time();  }  //保存时间为时间戳，而非时间字符串  public function fromDateTime($value) {  return $value;  }  //查询并返回数据时，返回时间戳格式。若不设置，默认返回Y-m-d H:i:s的时间格式  protected function asDateTime($value) {  return $value;  } |

亦可以在控制器中用format（）方法格式化时间。

|  |
| --- |
| $data = POST：：find（1）；  echo $data->updated\_at->format(Y-m-d); |

注意：要实现时间以时间戳形式保存，要先把数据库created\_at和updated\_at的数据类型改为int。

* + 1. Carbon：

laravel默认返回的时间都是Carbon对象，可以使用carbon的所有方法：详情见：

http://carbon.nesbot.com/docs/

例：$comment->created\_at->toFormattedDateString()

* + 1. 使用create方法插入数据

批量赋值：将一个数组发送到模型类用于创建新的模型实例（通常指表单请求数据）

前台表单提交数据，控制器可以：

|  |
| --- |
| $post = Post::create(Input::all());//不像save方法，不用一个一个去赋值，Input::all()即可获取所有表单数据； |

可以设置白名单（$fillable属性）和黑名单($guarded属性)，只能设一个；可以规定哪些字段可以被操作。

Protected $visiable=[‘name’,’sex’]或Protected $hidden=[‘password’]，可以设置能够查询到的字段或者屏蔽掉的不能查询的字段

|  |
| --- |
| $input = [  'title'=>'test 5',  'content'=>'test content',  'cat\_id'=>1,  'views'=>100,  'user\_id'=>2  ];  $post = Post::create($input);  $post->user\_id = 2;  修改被$guarded屏蔽的字段，使用save方法  $post->views = 100;  $post->save();  dd($post); |

先查询记录是否存在，不存在后再插入的操作：

|  |
| --- |
| $input = [  'title'=>'test 3',  'user\_id'=>5,  'cate\_id'=>1  ];  $post = Test::firstOrCreate($input);//先查询数据库，若存在此条数据，直接返回；若此条数据不存在，则插入数据库后再返回。  或 $post = Test::firstOrNew($input);//与firstOrCreate区别在于firstOrCreate不会将数据存入数据库，需要save才行。  $post ->save(); |

#### 更新模型

save方法更新：

首先获取该模型实例，然后修改模型属性，再调用save方法保存即可

|  |
| --- |
| $data = Test::find(2);  $data->title = 'Rose';  if($data->save()){  echo '更新成功';  } |

update方法更新：

与create相对应，也要用到批量赋值；

|  |
| --- |
| $post = Test::find(2);  if($post->update($input)){ //$input是要更新的数据数组  echo '文章修改成功';  dd($post);  }else{  echo '文章修改失败';  } |

#### 模型删除

* + 1. delete删除

|  |
| --- |
| $post = Post::find(5);  $bool = $post->delete() //返回true or false |

* + 1. destroy删除

|  |
| --- |
| $num = Post::destroy(5);//已知要删除的模型id；  $num = Post::destroy([1,2,3,4,5]);//可以传递一个id的数组进去 |

返回：被删除的记录数

* + 1. 用查询构建器删除模型

|  |
| --- |
| $num = Test::where('id','>',9)->delete();//返回被删除的记录数 |

* + 1. 软删除

不直接删除数据，而是给数据表增加一个deleted\_at字段，如果数据被删除，deleted\_at的值为删除时间，没被删除时值为null。

如要是用软删除，要在模型类中引入SoftDeletes：

|  |
| --- |
| <?php  namespace App\Models;  use Illuminate\Database\Eloquent\Model;  use Illuminate\Database\Eloquent\SoftDeletes;  class Test extends Model  {  use SoftDeletes;  protected $fillable = ['title', 'id','user\_id'];  protected $dates = ['deleted\_at']; //给模型类添加deleted\_at属性  } |

用Artisan命令生成迁移文件

|  |
| --- |
| php artisan make:migration alter\_tests\_deleted\_at --table=tests |

编辑生成的迁移文件：

|  |
| --- |
| public function up()  {  Schema::table('posts', function (Blueprint $table) {  $table->softDeletes();  });  } |

运行：php artisan migrate向表中添加deleted\_at字段

控制器中：

|  |
| --- |
| $post = Post::find(6);  $post->delete();  if($post->trashed()){ //$post->trashed()返回true，代表软删除成功  echo '软删除成功！';  dd($post);  }else{  echo '软删除失败！';  } |

然后Post：：all（）;查看数据表，被删除的数据不会显示，但数据表里的记录还在

若要查看包括软删除的记录，使用SoftDelete trait上的withTrashed方法：

|  |
| --- |
| $posts = Post::withTrashed()->get(); |

只查看被软删除的模型，使用onlyTrashed方法：

|  |
| --- |
| $posts = Post::onlyTrashed()->get(); |

恢复软删除：

恢复单个模型：

|  |
| --- |
| $post = Test::withTrashed()->find(7);//先将软删除的模型查到，然后再恢复  $post->restore(); |

恢复多个模型：

|  |
| --- |
| Post::withTrashed()->where('id','>',1)->restore(); |

恢复所有模型：

|  |
| --- |
| Post::withTrashed()->restore();//不加限制条件，恢复所有 |

强制删除：

|  |
| --- |
| $post = Post::find(6);  $post->forceDelete();//已经被软删除的模型也可以使用强制删除 |

#### 查询作用域和模型事件

* + 1. 查询作用域

将一些常用的查询封装到模型方法中，只需要在模型方法前加上scope前缀即可。

|  |
| --- |
| public function scopePopular($query)  {  return $query->where('views','>=',100);  }  //支持传入参数  public function scopeUserId($query,$user\_id = 12){  return $query->where('user\_id','<',$user\_id);  } |

在控制器中使用：

|  |
| --- |
| $tests = Test::popular()->userId(20)->orderBy('views','desc')->get(); |

* + 1. 模型事件 //类似于前置后置方法

当模型被创建、更新或删除的时候触发相应事件;

Eloquent目前支持八种事件类型：creating、created、updating、updated、saving、saved、deleting、deleted。

deleting和deleted很好理解，在删除模型时触发，deleting在删除操作前执行，deleted在删除完成后执行。

当创建模型时，依次执行saving、creating、created和saved，同理在更新模型时依次执行saving、updating、updated和saved。无论是使用批量赋值（create/update）还是直接调用save方法，都会触发对应事件（前提是注册了相应的模型事件）。

可以在任何你喜欢的地方注册模型事件，这里我们选择在服务提供者AppServiceProvider的boot方法中注册：

|  |
| --- |
| Post::saving(function($post){  echo 'saving event is fired<br>';  });  Post::creating(function($post){  echo 'creating event is fired<br>';  });  Post::created(function($post){  echo 'created event is fired<br>';  });  Post::saved(function($post){  echo 'saved event is fired<br>';  }); |

在插入数据时，会触发以上四个方法；

返回：

|  |
| --- |
| saving event is fired  creating event is fired  created event is fired  saved event is fired  <test model event>保存成功！ |

注意：saving/creating/updating/deleting事件返回false；

|  |
| --- |
| Post::creating(function($post){  ehco….;  if(……){  return false;  }  } |

|  |
| --- |
| class UserController extends Controller{  public function test(){  $data = array(  'name'=>'test 2',  'email'=>'test 2',  'password'=>1,  );  $post = User::create($data);  if(!$post->exists){ //exists属性可以判断创建的实例存不存在  echo '添加失败！';exit();  }  echo '&lt;'.$post->name.'&gt;保存成功！';  }  } |

相应的创建/更新/删除操作会退出，不向下执行，返回：

|  |
| --- |
| saving event is fired  creating event is fired  添加文章失败 //只触发前两个 |

#### 模型中的关联关系

模型关联中可以使用orderBy(‘’,’’desc’)等排序功能。

* + 1. 一对一

不管是hasOne还是belongsTo: 主表的id在后面，从表的id在前面

表user和表user\_account对应，相互关联,user表id 等于user\_account表user\_id

|  |
| --- |
| 在User模型中：  public function account()  {  return $this->hasOne('App\Models\UserAccount'，’user\_id’,’ id’);  } |

hasOne（$model,$foreignKey,$localKey）三个参数：

1. 关联的模型类
2. foreignKey: 从表中的字段
3. localKey: 主表中的字段

使用：

控制器中：

|  |
| --- |
| $account = User::find(2)->account;//在user\_account查user\_id=2的记录，后输出调用时不加括号； |

模板中：$user->account->num

但是一般不建议直接在模板中使用关联操作，因为view层不适合进行数据库查询操作，应该尽量将其放到控制器渲染模板之前，所以这样处理：

|  |
| --- |
| $post->load(account);//在渲染模板之前预加载account，  return view('post.show',compact('post')); |

注意：调用accout之后返回的是关联对象！！后面可以直接调用对象的属性

同样，也可以在UserAccount模型中定义：

|  |
| --- |
| public function user(){  return $this->belongsTo('App\Models\User','user\_id','id');  }  注意：若不加后两个参数，则用belongsTo的方法名user，作为四个参数$relation,在没有第二个参数时，会使用$relation.’\_id’作为foreignKey，对应user表的id，若没有相对应的列，又没指名参数，就会报错  因此，当belongsTo的方法名拼上‘\_id’刚好是有对应关系的字段，则可以只写一个参数，否则，需要指明对应的字段 |

belongsTo($model,$foreignKey,$otherKey，$relation)参数：

1. 关联的模型类
2. 从表的字段
3. 主表的字段

控制器中：

|  |
| --- |
| $user = UserAccounts::find(18)->user;//在account表中查id=18的记录，获取user\_id,再在user表查id=user\_id的记录。 |

* + 1. 一对多

在主表中定义使用$this->hasMany(),因为是一对多，返回的是个集合

在从表重定义使用$this->belongsTo();用法同一对一

* + 1. 多对多

例如：users表，rules表，一个用户可以有多个角色，一个角色可以对应多个用户

在User模型中定义：

|  |
| --- |
| public function role()//方法名无所谓  {  return $this->belongsToMany('App\Models\Role');  }  默认设置：users表、roles表通过role\_user（不带s）表来建立关系，且中间表的字段是user\_id和role\_id.若这些都不是默认值，则需要在belongsToMany()参数里设置 |

控制器：

|  |
| --- |
| $user = User::find(1);  $roles = $user->role;  foreach ($roles as $val){  echo $val->name."<br>";  echo $val->pivot->role\_id;//可以使用动态属性pivot获取中间表字段  } |

* + 1. 远层一对多

国家对用户是一对多，用户对文章是一对多，建立国家对文章的关系，即远层一对多。

在国家模型类中定义：

|  |
| --- |
| public function posts(){  return $this->hasManyThrough('App\Models\Post','App\Models\User');  } |

hasManyThrough的第一个参数是远层关联的模型，第二个参数是中间的模型。

若中间层user中对应国家的字段不是country\_id，文章模型中对应用户的字段不是user\_id,可以将这两个参数传入hasManyThough(),作为第三、第四个参数。

即：

|  |
| --- |
| return $this->hasManyThrough('App\Models\Post','App\Models\User'，‘country\_id’,’user\_id); |

* + 1. 关联统计

$posts = App\Post::withCount('comments')->get();

foreach ($posts as $post) {

echo $post->comments\_count;

}

会在$posts对象中增加个comments\_count属性

## Session

#### 使用实例

使用全局帮助函数session();

存放：session(['site'=>'LaravelAcademy.org']);

访问：$site = session('site');

可以操作数组：

session(['site.xxx'=>'LaravelAcademy.org']);

$site = session('site');

使用request实例：

首先要使用kernel.php的中间件组‘web’；//laravel默认已经加载了，可以不写

|  |
| --- |
| //Route::group(['middleware'=>'web'], function () {  // Route::get('session1','TestController@session1');  //}); |

获取所有session：$sessions = $request->session()->all();

存取session：

|  |
| --- |
| $request->session()->put('site', 'http://LaravelAcademy.org');  echo session('site');  结果：http://LaravelAcademy.org |

获取session:

$sitename = $request->session()->get('sitename','Laravel学院');//第二参数是默认值；

推送数据到session数组：

push（）的第二个参数会组成一个索引数组！区别于put（）。

|  |
| --- |
| $request->session()->push('site.xxx', 'http://LaravelAcademy.org');  $request->session()->push('site.xxx', 'Laravel学院');  if($request->session()->has('site')){  $site = $request->session()->get('site');  dd($site);  }  结果：  Array  (  [yyy] => Array  (  [0] => http://LaravelAcademy.org  [1] => yyyyy  )  )  上面若使用put(),也能达到目的：  $request ->session()->put('site.yyy',['a'=>'http://LaravelAcademy.org','b'=>'yyyy'])  区别：push()只能跟两个参数，每使用一次push，就向key对应的value里注入一个数组元素，且是索引数组元素，有点队列的意思；  put()第二个参数可以是字符串或者数组，数组也可以是关联数组。 |

获取并删除：pull()

|  |
| --- |
| $siteid = $request->session()->pull('siteid','LaravelAcademy');//第二参数是默认值；  echo $siteid;  $siteid = $request->session()->get('siteid');  echo $siteid; |

删除指定session：

|  |
| --- |
| $request->session()->forget('site'); |

一次性删除所有session：

|  |
| --- |
| $request->session()->flush(); |

静态类：

Session：：put（’’’key’,’val’）;

Session：：get（’’’key’[,’val’]）;//若key为空，获取默认值val；

Session：：pull（’’’key’[,’val’]）;

Session：：push（’’’user’,‘jack’）;

Session：：all（’’’key’’]）;等等，用法跟上面一样

Session::has()

Session::forget()

Session::flush()

Session::flash()//闪存

#### 一次性session

下一次请求中（仅仅是下一次）有效的Session数据；

|  |
| --- |
| $request->session()->flash('error','failed'); |

若定义过一次性session后，又想持续保存session了，可以用

|  |
| --- |
| $request->session()->reflash();//没有参数 |

也可以单独设置哪些需要持续保存：

|  |
| --- |
| $request->session()->keep(['status','error']);//参数是需要保存的session的key的数组 |

## Blade模板

#### 三元判断

{{isset($a) ? $a : ‘no’}}可以简化为{{$a or ‘no’}}//存在为$a，不存在输出no

#### 原样输出、

@{{ $name }}

#### 模板注释

页面中看不到的注释

{{-- --}}

#### 引入子视图

{{include …}}

#### 流程控制

@if( )

@elseif( )

@else

@endif

@unless( )

@endunless( )

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

@endfor

@foreach($arr as $val)

@endforeach

//forelse比foreach多了个为空的判断

@forelse($arr as $val)

@empty//如果为空

@endforelse

#### 模板中生成链接

<a href=””” {{ url(‘路由名称’) }}“> </a>

<a href=””” {{ action(‘控制器名@方法名’) }} “> </a>

<a href=””” {{ route(‘路由别名’) }} “> </a>

#### 模板中获得当前页面路由

Request::getPathInfo() //获得当前页面路由

例：{{Request::getPathInfo()=='/student/index'?'active' :''}}

#### {{!! $content !!}} 双叹号表示不转义。

默认情况下，{{}}会自动调用 PHP htmlspecialchars 函数来转义数据，以避免 XSS 的攻击

## 全局帮助函数

public\_path（） public下的文件目录

例：public\_path(‘images’);

## Cookie详解

#### 设置cookie

|  |
| --- |
| $cookie1 = Cookie::make('test','test',100);//普通cookie，使用Cookie门面  $cookie2 = Cookie::forever('test2','test2');//永久cookie  $cookie4 = cookie()->forever(‘test4’,’test4’);//用全局帮助函数设置永久cookie  $cookie3 = cookie('test3','test3',200);//全局帮助函数cookie  $cookie5 = cookie('test5','test5',200);//全局帮助函数cookie  return Response::make()->withCookie($cookie3);//make()是返回一个response实例  return response(' ',200)->withCookie ($cookie);//response也是返回一个实例  return response('hello')->cookie('test7','test7',10);//withCookie参数是cookie（‘ ’,‘ ’， ）  推荐：  Cookie::queue('test6','test6',20);//使用cookie队列，会自动添加到响应，可以单独使用 |

#### 获取cookie

|  |
| --- |
| $cookie = Cookie::get();//获得全部的cookie  $cookie2 = Cookie::get('test2');//获得某一个cookie  亦可以使用request：  $request->cookie();//获取所有cookie  $request->cookie(‘test1’);  Request：：cookie（‘test2’）; |

#### 清除cookie

|  |
| --- |
| Cookie::queue(Cookie::forget('test'));//方法一  setcookie('test', '', -1, '/')//方法二  //方法三  $cookie = Cookie::forget('test');  return response（‘sorry’）->withCookie($cookie); |

## Redis

|  |
| --- |
| //存入字符串  Redis::set($key,$val)  //根据键名获取键值  Redis::get($key)  //判断指定键是否存在  Redis::exists($key) |

## 表单验证

#### 控制器验证

基类App\Http\Controllers\Controller里面引入了ValidatesRequests trait，里面有个validate方法。

|  |
| --- |
| $this->validate(*$request*, [  'Student.name'=>'required|min:2|max:5',  'Student.age'=>'required|integer', ],  );  ---------------------------------------------------------------------------------------  若希望首次验证失败后停止其他验证；可以在属性中分配bail规则；  'Student.name'=>'bail | required | min:2 | max:5',  'Student.age'=>'required|integer', //若name通过不了，就不验证age了 |

错误提示信息改为中文方法：将resources/lang/en复制一份，将里面的validation.php替换成中文版本即可。可搜索

验证通过则继续执行以下代码，若验证失败，会重定向到上一个页面，并且走\Illuminate\View\Middleware\ShareErrorsFromSession::class,中间件；这个中间件里，将所有错误信息一次性存放到session中

|  |
| --- |
| $this->view->share(  'errors', *$request*->session()->get('errors') ?: new ViewErrorBag ); |

会返回一个errors到视图中，可以在视图中循环出来错误信息

|  |
| --- |
| @foreach($error->all() as $error)//获取所有错误信息，并循环  {{$error}}  @endforeach |

#### Validator类验证

使用Validator门面手动创建一个验证器实例

|  |
| --- |
| $validator = Validator::make($request->input(),[  'Student.name'=>'required|min:2|max:5',  'Student.age'=>'required|integer',  ],  [  'required'=>':attrbute 为必选项' //定义required具体什么意思，:attrbute为占位符  ],  [  'Student.name'=>'姓名'//定义上面数组的:attribute是什么意思  ]);  if ($validator->fails()){  return redirect()->back()->withErrors($validator);//需要手动返回错误信息，将错误信息一次性存入session，$error变量重定向后自动在视图间共享  //或者withErrors(错误信息)  } |

#### 验证规则

unique:table,字段名，排除验证的字段

## 邮件发送

#### 纯文本

|  |
| --- |
| 在.env配置：    MAIL\_PASSWORD是授权码，并不是邮箱登录密码  测试方法：  public function mail()  {  //raw()第一个参数是邮件的主体内容，此处是文本。  Mail::raw('邮件内容', function ($message) {  $message->from('1027612662@qq.com','举世无双');//发送者  $message->subject('测试');//主题  $message->to('ju910628@163.com');//接收者  });  } |

#### 基于html模板的邮件

建一个HTML页面：mail.blade.php

|  |
| --- |
| <h1>{{$content}}</h1> |

|  |
| --- |
| //send()第一个参数是邮件模板的地址，第二个参数是模板中的变量赋值，第三个参数跟上面方法一样  Mail::send('mail',['content'=>'邮件内容'], function ($message) {  $message->from('1027612662@qq.com','举世无双');  $message->subject('测试');  $message->to('ju910628@163.com');  }); |

1. Redis队列

#### 配置

在config/queue.php中有队列的相关配置，里面的connection=>default对应config/database.php中redis的default配置, queue为默认队列名称；

更改.env的QUEUE\_DRIVER为redis即可。

#### 编写队列任务

首先我们通过如下Artisan命令创建任务类：

|  |
| --- |
| php artisan make:job SendReminderEmail |

生成的类都实现了 Illuminate\Contracts\Queue\ShouldQueue 接口, 告诉Laravel将该任务分配到队列，而不是立即运行。

在app/Jobs下生成SendReminderEmail.php

|  |
| --- |
| <?php  namespace App\Jobs;  use App\User;  use Illuminate\Bus\Queueable;  use Illuminate\Contracts\Mail\Mailer;  use Illuminate\Queue\SerializesModels;  use Illuminate\Queue\InteractsWithQueue;  use Illuminate\Contracts\Queue\ShouldQueue;  //use Illuminate\Support\Facades\Mail;  class SendReminderEmail implements ShouldQueue  {  use InteractsWithQueue, Queueable, SerializesModels;  protected $user;    //因为使用了SerializesModels trait,Eloquent模型会在任务被执行时优雅地序列化和反序列化：在构造方法里接收Eloquent模型，只有模型的主键会被序列化到队列，当任务真正被执行时，队列系统会自动从数据库中获取整个模型实例。  public function \_\_construct(User $user)  {  $this->user = $user;  }    public function handle(Mailer $mailer)  {  //此处填写队列任务逻辑  //这里的Mailer和前一节邮件发送中使用的Mail门面有异曲同工之效，它们最终调用的都是同一个类上的方法，这个类就是Illuminate\Mail\Mailer  $user = $this->user;  $mailer->send('email.reminder',['user'=>$user], function ($message) use($user){  $message->to($user->email)->subject('新功能发布');  });  }  } |

创建视图email/reminder

|  |
| --- |
| 亲爱的{{$user->name}}，您好，Laravel学院新发布了XXX功能，立即去体验下吧：  <a href="http://laravelacademy.org">前往学院</a> |

#### 推送队列任务

可以使用控制器中的DispatchesJobs trait（该trait在控制器基类Controller.php中引入，不需要单独引入）提供的dispatch方法手动分发任务

在控制器中手动推送队列任务：

|  |
| --- |
| //发送提醒邮件  public function sendReminderEmail(Request $request,$id){  $user = App\User::findOrFail($id);  //dispatch（）参数是一个队列任务的实例；  $this->dispatch(new SendReminderEmail($user));  或用全局帮助函数dispatch()  dispatch(new SendReminderEmail($user));  }  延时任务:  $job = (new ProcessPodcast($pocast))  ->delay(90);  dispatch($job); |

路由：Route::get('mail/sendReminderEmail/{id}','MailController@sendReminderEmail');

另：当然你可以在控制器之外的其它地方使用dispatch分发任务，当然在此之前需要在该类中使用 use DispatchesJobs。

#### 运行队列监听器

在浏览器访问路由，此时任务呗推送到redis队列中，还需要运行artisan命令执行队列中的任务，有三种：

1. queue:work 只执行一次队列请求，请求执行完就停止
2. queue:listen 监听队列请求，只要运行着，就能接收请求，除非手动停止
3. queue:work --daemon (推荐) 同 listen 一样， 只要运行着，就能一直接受请求，不一样的地方是在这个运行模式下，当新的请求到来的时候，不重新加载整个框架，而是直接 fire 动作。

注：使用 queue:work --daemon ，当更新代码的时候，需要停止，然后重新启动，这样才能把修改的代码应用上。

## 事件

事件监听器监听到事件发生后会执行一些操作，Laravel使用观察者模式来实现这种监听机制。

#### 注册事件-监听器

首先我们需要在EventServiceProvider.php中注册事件与监听器之间的映射关系：

|  |
| --- |
| protected $listen = [  'App\Events\addAdminEvent' => [  'App\Listeners\addAdminListener',  ],  ]; |

运行artisan命令

|  |
| --- |
| php artisan event:generate |

该命令会在app/Events目录下生成addAdminEvent.php，在app/Listeners目录下生成addAdminListener.php。

#### 定义事件类

该事件类不包含任何特定逻辑，只是一个存放adminUser对象的容器，

|  |
| --- |
| class addAdminEvent  {  use InteractsWithSockets, SerializesModels;  public $adminUser;  //在事件类注入AdminUser实例  public function \_\_construct(AdminUser $adminUser)  {  $this->adminUser = $adminUser;  }    public function broadcastOn()  {  return new PrivateChannel('channel-name');  }  } |

#### 定义监听器类

监听器类中使用handle方法接收事件实例并执行响应该事件的业务逻辑，此处是在添加管理员时给他发个邮件；

|  |
| --- |
| public function handle(addAdminEvent $event)  {  $name = $event->adminUser->user\_name;  Mail::raw('我是'.$name, function ($message) {  $message->from('ju910628@163.com');  $message->subject('测试');  $message->to('1027612662@qq.com');  });  }  //有时监听器会进行一些耗时操作，这时应该结合Laravel的队列系统将监听器进行队列化，前提是已经配置了队列并开启了队列处理器。队列化非常简单，只需监听器实现ShouldQueue接口即可，即：  use Illuminate\Contracts\Queue\ShouldQueue;  class addAdminListener implements ShouldQueue  {  /\*\*  \* 失败重试次数  \* @var int  \*/  public $tries = 1;  ...  } |

#### 触发事件

要触发以上事件，可以使用Event门面提供的fire方法，在AdminController中createUser方法添加管理员之后调用Event::fire(),将事件实例传进去

|  |
| --- |
| //添加管理员  public function createUser(Request $request)  {  $user\_name = $request->input('username');  $admin\_salt = rand(1000,9999);  $password = md5(md5($request->input('password')).$admin\_salt);  $user3 = ['user\_name'=>$user\_name,'password'=>$password,'admin\_salt'=>$admin\_salt];  $adminUser = AdminUser::firstOrCreate($user3);  \Illuminate\Support\Facades\Event::fire(new addAdminEvent($adminUser));  或使用全局帮助函数event（）  } |

在添加管理员之后就会触发发邮件事件

执行流程：控制器先触发addAdminEvent事件，然后监听器监听到事件，执行 handle里面的逻辑代码。