# MVC和Laravel的HTTP层

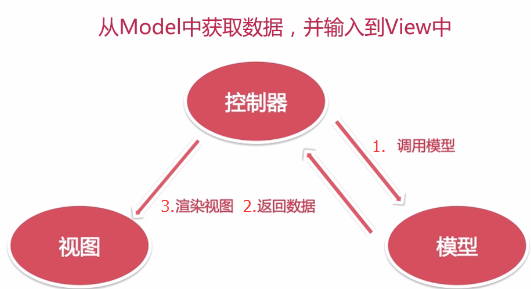
## MVC简介

MVC是模型(model)－视图(view)－控制器(controller)的缩写，是一种UI层的框架(传统的三层架构可分为表现层、中间层和数据层，MVC正是表现层的架构)

model：程序中传递数据的部分，是数据的载体；

view：程序中负责显示数据的部分；

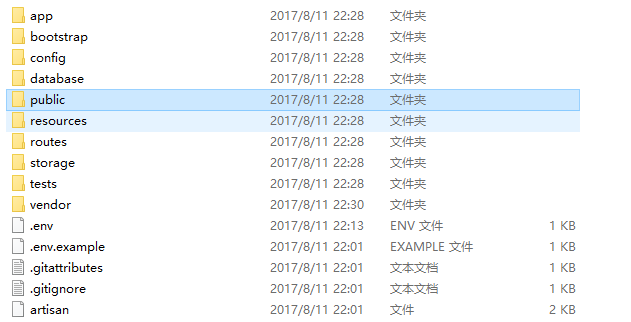
controller：是处理视图层与用户交互的部分，对输入和输出做出反应



## laravel文件结构

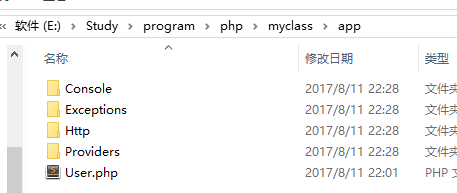
laravel大量使用了php的新特性，因此对php的版本要求较高：

|  |  |
| --- | --- |
| laravel | php |
| 5.3，5.4 | 5.6.4+ |
| 5.1，5.2 | 5.5.9+ |
| 4.2 | 5.4+ |
| 4.1 | 5.3.7+ |



1. **app文件夹**

它包含程序的核心代码，也就是我们自己写的业务逻辑



app文件中的Http文件下有一个app\Http\Controllers文件，我们MVC架构中的controller的代码将写在其中

1. **bootstrap文件夹**

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为提升性能所生成的文件，如路由和服务缓存文件

1. **config文件夹**

config 目录包含了应用所有的配置文件

1. **database文件夹**

database 目录包含了数据迁移及填充文件

1. **public文件夹**

public 目录包含了入口文件 index.php 和前端资源文件（图片、JavaScript、CSS等）

1. **resources文件夹**

resources 目录包含了视图文件及原生资源文件，其中的view文件就是mvc中的view



1. **routes文件夹**

routes 目录包含了应用的所有路由定义。Laravel默认提供了三个路由文件：web.php、api.php和console.php

1. **storage文件夹**

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

1. **tests文件夹**

tests 目录包含自动化测试，每一个测试类都要以 Test 开头

1. **vendor文件夹**

该文件包含所有Composer依赖。

## 1.3 一次请求的生命周期

laravel应用的入口是public文件中的index.php，如果使用的是 Apache 服务器，Laravel中包含的 .htaccess 文件会将所有请求导向这个文件。index.php文件从 bootstrap/app.php 脚本获取 Laravel 应用实例，创建Ioc服务容器实例，也即是Laravel Application对象

然后，framework/start.php 文件配置相关设置并加载各个服务提供器，即向容器注册服务

接着加载app/start下的文件

最后加载Routes，一旦应用被启动并且所有的服务提供者被注册，请求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。

## 1.4 laravel中的路由

larvae中的路由的作用是分析来自客户端的url，根据代码设定的规则将用户的url请求转发给相应程序处理执行，它在url和程序之间建立了映射，它抛弃了抛弃原生php 开发过程中的复杂繁琐的请求路径，避免暴露网站的文件结构，增加了访问的优雅性和安全性

laravel中的请求有get、post、put、patch、delete几种，应用中的所有的路由都定义在位于 routes 目录下的路由文件中，对大多数应用而言，都是从 routes/web.php 文件开始定义路由

### 1.4.1基本路由

基本路由只接收一个 URI 和一个闭包

**语法：**

**Route::post($uri, $callback)**

eg：

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

return 'welcome to use laravel';

});

**注意:1.若为put、patch、delete等其他路由则将post换成响应的put、patch、delete即可**

**2.使用post请求提交表单时，Laravel框架为了防止跨域请求攻击（CSRF）而为用户生成的随机令牌，post请求如果没有验证token，会报错，解决方案是：在form表单中添加一个隐藏域，携带token参数即可**

**eg：**

<**form action="user" method="post"**>  
 **<input type="hidden" name="\_token" value="{{csrf\_token()}}">**  
 用户名：<**input type="text" name="username"**>  
 密码：<**input type="password" name="pwd"**>  
 <**input type="submit" value="提交"**>  
</**form**>

### 多请求路由

多请求路由可以接受多种请求

**方式一：**

**Route::match(请求类型数组,$uri, $callback)**

eg：

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

return 'welcome to use laravel';

});

**方式二：**

any的多请球路由会响应所有类型的路由

**Route::any($uri, $callback)**

eg：

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

return 'welcome to use laravel';

});

### 路由参数

有时候，我们需要从url中截取部分片段，如从url中获取用户id等，

eg：

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

return 'id='.$id.'<br>password='.$password;

});

**注意：**1**.可以使路由参数变为可选，只需要在{}的参数名后面加上?，并且为了防止return语句运行出错，再给回调函数的参数以默认值，即：**

Route::get('/user/{id?}/{password?}',function($id=null,$password=123){

return 'id='.$id.'<br>password='.$password;

});

**2.路由参数还可以通过正则表达式来进行限制，只需要在最后面加上数组->where([被正则表达式验证的参数=>正则表达式]);**

**此时若url给定参数不符合正则表达式，则不匹配该路由，若最终无路由可匹配，则报错**

eg:

Route::get('/user/{id?}/{password?}',function($id=null,$password=123){

return 'id='.$id.'<br>password='.$password;

})->where(**[**'id'**=>**'[A-Za-z]+' , 'password'**=>**'[A-Za-z]+'**]**);

### 路由命名

路由命名可以很方便地获取路由对应的url

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

return Route('profile');

})->name('profile');

**注意:若命名时url使用了路由参数，则应当将路由参数以数组形式作为参数传递给Route函数**

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

return Route('profile',[$id,$password]);

})->name('profile');



### 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，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

**语法：**

**Route::group(共享属性数组,function(){**

**路由1;**

**路由2;**

**…**

**})**

**注意：1.共享属性数组可以为:**

1. **中间件(middleware)**
2. **命名空间(namespace )**
3. **子域名路由(domain)**
4. **路由前缀(prefix)**

**添加路由前缀后，相当于在路由群组中的每一个路由的url前面增加了一个前缀，eg：**

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

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

return 'Im password';// 访问http://localhost:8080/myclass/ID/password

});

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

return 'Im user';

});

});

## 控制器Controller

将所有的请求处理逻辑都放在单个路由文件中显然是不合理的，我们需要使用控制器类组织管理这些行为，控制器文件通常放在app\Http\Controllers文件夹中

### 1.5.1基本控制器

所有的Laravel 控制器应该继承控制器基类Controller

如：

在文件app\Http\Controllers中新建一个php文件：

namespace App\Http\Controllers; //设定命名空间，应当与文件结构保持一致

use App\Http\Controllers\Controller; //使用App\Http\Controllers\Controller类

class myController extends Controller{

function getInfo($id){ //可以从路由url中获取参数

return Route('anotherName',[$id]);

}

}

在路由文件中：

Route::get('user/{id}','myController@getInfo') ->name('anotherName');//不需要写成App\Http\Controllers\myController

或：

Route::get('user/{id}',['uses'=>'myController@getInfo']) ->name('anotherName');//不需要写成App\Http\Controllers\myController

**注意: 1.与控制器绑定的路由也可以使用路由参数，同样也可以添加where限制**

### 1.5.2 请求Request

#### 1.访问Request实例

一般通过依赖注入来获取HTTP请求实例，在控制器的构造函数或方法中对 Illuminate\Http\Request 类进行类型提示，这样当前HTTP请求实例会被服务容器自动注入，eg：

**<?php**

**namespace** App\Http\Controllers;  
**use** Illuminate\Http\Request;  
**class** UserController **extends** Controller  
{**public function** store(Request $request)  
 {  
 $name=$request->input(**'name'**);}  
}

如果还希望控制器的方法获取路由参数，可以将路由参数置于其它依赖之后，eg：

Route::*get*(**'model/{id}'**,**'studentController@orm'**);

**class** studentController **extends** Controller{

**public function** orm(Request $request,$id){

**return '当前路径：'**.$request->path().**'<br>'**.**',参数是'**.$id;  
}

}

#### 2.Request常用方法

**(1) path ()** 返回请求的路径信息, 如果进入的请求路径是 http://domain.com/foo/bar，则 path 方法将会返回 foo/bar

eg：

Route::*get*(**'model/{id}'**,**'studentController@orm'**);

**class** studentController **extends** Controller{

**public function** orm(Request $request,$id){

**return '当前路径：'**.$request->path();  
}

}

**(2) is()** 验证请求的路径信息是否与给定的格式匹配，使用\*通配符省略后面的内容

eg：

Route::*get*(**'model/{id}'**,**'studentController@orm'**);

**public function** orm(Request $request,$id){  
 **if** ($request->is(**'model/\*'**))  
 **return '是model模式'**;  
 **else  
 return '不是model模式'**;  
}

**(3) url()/fullUrl** url 方法将会返回不带查询字符串的URL，而 fullUrl 方法的返回结果则包含查询字符串

eg：

**public function** orm(Request $request,$id){

**return '不带查询字符串的url'**.$request->url().**'<br>带查询字符串的url'**.$request->fullUrl();  
}

**//**不带查询字符串的绝对路径http://localhost:8080/myclass/model/EDG  
带查询字符串的绝对路径http://localhost:8080/myclass/model/EDG?name=3

**(4) isMethod()** 判断HTTP请求的类型，可填入的参数有 ’psot’, ’get’, ’psot’ , ’put,’ patch, ’delete’

eg：

**public function** orm(Request $request,$id){  
 **if**($request->isMethod(**'post'**))  
 **return '是post方法'**;  
 **else if**($request->isMethod(**'get'**))  
 **return '是get方法'**;  
 **else if**($request->isMethod(**'put'**))  
 **return '是put方法'**;  
 **else if**($request->isMethod(**'patch'**))  
 **return '是patch方法'**;  
 **else if**($request->isMethod(**'delete'**))  
 **return '是delete方法'**;  
}}

**(5) method()** 获取当前HTTP请求的访问类型，返回值为访问类型的字符串

**(6) ajax()**  判断当前HTTP请求是否为ajax请求，返回bool值

eg：

**public function** orm(Request $request,$id){  
 **echo** *var\_dump*($request->ajax());  
}

#### 3.获取Request输入

all()方法可以用数组形式获取url中的所有查询字符串，该方法返回url查询字符串数eg：

**public function** orm(Request $request,$id){

**echo** *var\_dump*($request->all());  
}

input()方法可以获取单个输入值，使用该方法还可以将第二个参数作为该查询字符的默认值，eg：

**public function** orm(Request $request,$id){  
 **echo** $request->input(**'name'**);  
 **return** $request->input(**'age'**,**'10'**);  
}

处理表单数组输入时，可以使用”.”来访问数组输入：

$input = $request->input(**'products.0.name'**);  
$names = $request->input(**'products.\*.name'**);

获取JSON输入值

发送JSON请求到应用的时候，只要 Content-Type 请求头被设置为 application/json，都可以通过 input 方法获取 JSON 数据，还可以通过“.”号解析数组：

$name = $request->input(**'user.name'**);

#### 4.文件上传

request请求提供了file方法访问上传的文件， hasFile 方法判断文件在请求中是否存在，isValid方法判断文件是否合法：

eg：

if ($request->hasFile('photo')&& $request->file('photo')->isValid()) {

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

}

laravel提供了store方法来帮助我们保存文件，**第一个参数是目录名**，而不是文件名。默认情况下，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，**第二个参数可选，是指定用于存储的磁盘驱动**，该函数**返回存储路径**

if($request->hasFile('upFile')&&$request->file('upFile')->isValid()){

$path=$request->file('upFile')->store('new/','local');

return '上传成功,存储在'.$path;

}else

return '上传失败';

**注意：**

**1.上传表单方法必须是post方法**

**2.需要在表单中添加csrf令牌**

<input type="hidden" name="\_token" value="{{csrf\_token()}}">

**3.form标签需指定文件编码格式**

enctype="multipart/form-data"

eg：

<form action="{{route('upFile')}}" method="post" enctype="multipart/form-data">

<input type="hidden" name="\_token" value="{{csrf\_token()}}">

<input type="file" name="upFile">

<input type="submit" value="上传">

</form>

### 1.5.3 响应Response

所有路由和控制器都会返回一个被发送到用户浏览器的响应，Laravel 提供了多种不同的方式来返回响应，大多数情况下，都会返回一个完整的 Illuminate\Http\Response 对象实例或视图

#### Response对象

返回一个完整的Response实例可以自定义响应的 HTTP 状态码和头信息。可以通过header()方法添加头信息，也可以通过withHeaders方法来指定头信息数组并添加到响应

eg：

**return** response(**'Hello World'**, 200)

->header(**'Content-Type'**, **'text/plain'**);   
->header(**'X-Header-One'**, **'Header Value'**)  
->header(**'X-Header-Two'**, **'Header Value'**);

**return** response($content)  
 ->withHeaders([  
 **'Content-Type'** => $type,  
 **'X-Header-One'** => **'Header Value'**,  
 **'X-Header-Two'** => **'Header Value'**,  
 ]);

还可以通过Response的make()方法创建一个新的response响应，eg：

**return** response()->make(**'hello!!!myWorld'**,200,[**'Content-Type'**=>**'text/plain'**]);

#### 字符串响应

建议通过创建一个完整的Response实例来返回字符串

eg：

**public function** response(){  
 **return 'hello,world!'**;  
}

#### 视图响应

header()方法可以设置响应头的头信息，view()方法的第二个参数是插入插槽的变量，第三个是响应的状态码，200代表OK

eg：

**public function** jsonp(Request $request){  
 $data = [  
 **'name'** => **'张彪'**,  
 **'sex'** => **'男'** ];  
 $age=**'28'**;  
 $type=**'json'**;  
 **return** response() ->view(**'myView.child1'**, [**'data'**=>$data,**'age'**=>$age], 200)->header(**'Content-Type'**, $type);  
}

#### JSON响应

控制器方法在return一个数组时会自动将数组转化为一个json响应，也可以通过辅助函数response()的**response()->json()**方法手动转化为json响应，eg：

$data = [  
 **'name'** => **'张彪'**,  
 **'sex'** => **'男'**,  
 **'age'**=>**'18'**];  
**return** response()->json($data);

**return**  $data;

**注意：如果想要返回一个jsonp响应，需要在json()方法后添加->setCallback(Input::get('callback'));**

eg：

**return** response()->json($data) ->setCallback($request->input(**'callback'**));

**return** \Illuminate\Support\Facades\Response::*json*($student)->setCallback($request->input(**'callback'**));

注意：\Illuminate的’\’不能丢

#### 重定向响应

重定向响应是Illuminate\Http\RedirectResponse 类的实例，可以通过全局辅助函数**redirect()**和**Redirect门面**来使用重定向实例，重定向的方法一般有两个，route()和action()，**route(路由名称)**是通过路由名称来进行重定向，**action(‘控制器@方法’)**是通过控制器方法，直接使用辅助函数**redirect(‘路由url’)**是通过url重定向，就像3.3，

**public function** response(Request $request)  
{  
 **//return redirect('mysession/session2');  
 //return redirect()->route('myRoute');  
 //return redirect()->action('studentController@session2');  
  
 //return Redirect::route('myRoute');  
 //return Redirect::action('studentController@session2');**}

如果想要将用户重定向到上一个请求的位置，比如，表单提交后，验证不通过，可以通过back()函数回到前一个url，但要注意防止循环重定向出现，eg:

**return** Redirect::*back*();

如果重定向的目标地址需要参数，则可以在**route()和action()**中提供，eg：

Route::*get*(**'session/{id}'**, **'studentController@mysession'**)->name(**'myRoute'**);

Route::*get*(**'response'**,[**'uses'**=>**'studentController@response'**]);

**public function** response()  
{  
 **//return redirect()->route('myRoute',['id'=>'myid']);****return** redirect()->action(**'studentController@session2'**,[**'id'**=>**'myid'**]);

}

**public function** mysession()  
{  
 **return** Response::*make*(**'我是重定向4'**,200,[**'Content-Type'**=>**'text/plain'**]);  
}

实际开发中，常常需要重定向到一个新的 URL 同时将数据存储到一次性 Session 中，我们可以使用**with()**方法，eg：

Route::post(**'user/profile'**, **function** () {  
 **return** redirect(**'redir'**)->with(**'status'**, **'Profile updated!'**);  
});

#### 文件响应

文件响应是直接在用户浏览器显示文件，例如图片或PDF，而不需要下载，该方法接收文件路径作为第一个参数，头信息**数组**作为第二个参数：

eg：

**return** response()->file(base\_path(**'public/img/myHeadImg.jpg'**),[**'Content-Type'**=>**'image/jpeg'**]);

#### 文件下载响应

download()方法用于生成下载文件的响应，return该方法的返回值会使当前页面出现下载提示，该方法的第二个参数是下载后文件的文件名

eg：

**return** response()->download(base\_path(**'public/img/myHeadImg.jpg'**),**'我的照片'**,[**'Content-Type'**=>**'image/jpeg'**]);

### 1.5.4 会话Session

#### 1.session配置和简介

由于HTTP协议是无状态的，因此laravel提供了session来**在服务器上存储(cookie在浏览器端存储)**用户请求信息，Session 配置文件位于 config/session.php，Laravel 默认使用的 Session 驱动为 file 驱动，也可以使用 memcached 或者 redis 驱动获取更快的 Session 性能。

**'driver'** => env(**'SESSION\_DRIVER'**, **'file'**),

**'table'** => **'sessions'**,//当使用数据库驱动时默认表为sessions

**'lifetime'** => 120, //默认 sessions数据保存120分钟

Session 驱动定义请求的 Session 数据存放在哪里，Laravel 可以处理多种类型的驱动：

* file – Session 数据存储在 storage/framework/sessions 目录下；
* cookie – Session 数据存储在经过加密的安全的 Cookie 中；
* database – Session 数据存储在数据库中
* memcached / redis – Session 数据存储在 [Memcached](http://laravelacademy.org/tags/memcached)/[Redis](http://laravelacademy.org/tags/redis) 缓存数据库中；
* array – Session 数据存储在简单 PHP 数组中，在多个请求之间是非持久化的

**注意：想要使用session的路由必须加载中间件’web’，eg：**

Route::*group*([**'middleware'**=>**'web'**],**function** (){  
 Route::*get*(**'session1'**,[**'uses'**=>**'studentController@session1'**]);  
 Route::*get*(**'session2'**,[**'uses'**=>**'studentController@session2'**]);  
});

#### 2.访问session 实例

laravel访问session有三种方式

1. **通过HTTP request的session()方法**
2. **通过session全局辅助函数**
3. **通过session facade类**

eg：

**public function** session1(Request $request){  
 *// 1.通过http session方法使用* $request->session()->put(**'name'**,**'DJ'**);  
 *//2.通过全局辅助函数session()使用* session()->put(**'sex'**,**'男'**);//或

session([**'user'**=>**'123'**,**'password'**=>**'123'**]);  
 *//3.通过session 门面类使用*

Session::*put*(**'age'**,**'18'**);

}  
**public function** session2(Request $request){  
 *// 1.通过http session方法使用* **echo** $request->session()->get(**'name'**,**'默认姓名'**);  
 **echo '<br>'**;  
 **echo** session->get(**'sex'**,**'默认性别'**); //或

**echo** session (**'sex'**,**'默认性别'**);  
 **echo '<br>'**;  
 **echo** Session::*get*(**'age'**,**'默认年龄'**);  
}

#### 3.session常用方法

上节三种使用session的方式都提供了对应的session常用方法

**(1)all()** all方法获取session中的所有数据

eg:

**public function** session2(Request $request){  
 *// 1.通过http session方法使用* **echo** *var\_dump*($request->session()->all()) ;  
 **echo '<br>'**;  
 **echo** *var\_dump*(session()->all());  
 **echo '<br>'**;  
 **echo** *var\_dump*(Session:: all());  
}

**(2)has()** has方法检查数据项在 Session 中是否存在, 如果存在并且不为 null 的话返回 true

eg:

**if** ($request->session()->has(**'name'**)) {  
 }

**(3)exists()** 判断某个值在 Session 中是否存在，只要存在即使是 null 的话也返回true

#### 4.session存取数据

**存入**

**(1)put()** 一次存入一个变量到session中

eg:

$request->session()->put(**'name'**,**'DJ'**);

**注意：使用全局辅助函数session()可以一次存入多个变量,eg：**

session([**'user'**=>**'123'**,**'password'**=>**'123'**]);

**(2)push()**  向数组中推送数据，如果是关联数组，可以通过’.’来推送数据

eg:

$request->session()->push(**'student1'**,**'DJ'**);

$request->session()->push(**'student1'**,**'16'**);

**echo** Session::*get*(**'student1.0'**);

$\_SESSION[**'arr'**]=**array**();//新建一个空数组

session()->push(**'arr.name'**,**'张政'**); //向session中的数组arr推送数据，相当于arr[name]=’张政’

session()->push(**'arr.age'**,**'22'**);

dd(session(**'arr.name'**,**'无名'**)); //通过arr.name访问arr数组中的name的值

**读取**

**(3)** **get()** 从session中读取一个变量数据

eg:

**echo** Session::*get*(**'student1.0'**);

**删除**

**(4)pull()** 从session获取并删除数据

eg:

**echo** Session::*pull*(**'student1.0'**,**'已被删除'**);

**(5)forget()** 从session删除指定数据

eg:

session()->forget([**'student2.1'**,**'student2.0'**]);

**(6)flush()** 清空session中所有数据

eg:

**echo** Session::*flush*();

**一次性数据**

**(7)flash()** 在 Session 中存储只在下个请求中有效的数据,即’一次性’数据，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

eg:

session()->flash(**'ycx'**,**'一次性数据'**);

**(8)** **reflash()** 将session中**所有的**一次性数据保留到下一个请求，使其可以再被使用一次

eg:

session()->reflash();

**(9)** **keep()** 将session中被该方法指定的一次性数据保留到下一个请求，使其可以再被使用一次

eg:

session()->keep([**'ycx1'**,**'ycx2'**]);

### 1.5.5 浏览器缓存cookie

不同于服务器端缓存session，cookie是存储于浏览器的缓存，相比于session其安全性较差，单个cookie保存的数据不能超过4K，很多浏览器都限制一个站点最多保存20个cookie，整个站点诸多不同页面共用一个cookie文件

#### 1. cookie和session的关系

session 的运行依赖 session id，而 session id 是存在 cookie 中的，因此如果禁用浏览器端的cookie会影响服务器端的session运行，但可以通过url传递session id

由于HTTP协议是无状态的协议，所以服务端需要记录用户的状态时，就需要用某种机制来识具体的用户，这个机制就是Session.典型的场景比如购物车，当你点击下单按钮时，由于HTTP协议无状态，所以并不知道是哪个用户操作的，所以服务端要为特定的用户创建了特定的Session，用用于标识这个用户，并且跟踪用户，这样才知道购物车里面有几本书。那么服务端如何识别特定的客户？这个时候Cookie就登场了。每次HTTP请求的时候，客户端都会发送相应的Cookie信息到服务端。实际上大多数的应用都是用 Cookie 来实现Session跟踪的，第一次创建Session的时候，服务端会在HTTP协议中告诉客户端，需要在 Cookie 里面记录一个Session ID，以后每次请求把这个会话ID发送到服务器，我就知道你是谁了，如果客户端的浏览器禁用了 Cookie 怎么办？一般这种情况下，会使用一种叫做URL重写的技术来进行会话跟踪，即每次HTTP交互，URL后面都会被附加上一个诸如 sid=xxxxx 这样的参数，服务端据此来识别用户。

cookie还有一个常用的地方是存储用户账号，将账号密码写到Cookie里面，访问网站的时候，网站页面的脚本可以读取这个信息，就自动帮你把用户名给填了，能够方便一下用户

#### 2. 添加cookie

**Cookie::make()**

在使用cookie之前，应先引入cookie的门面Illuminate\Support\Facades\Cookie，通过**Cookie::make(变量名,变量值,存在时间)**创建一个cookie值，再通过**withCookie(cookie值)**方法将其添加到response中去

$myFirstCookie=Cookie::*make*(**'myCookie'**,**'这是我的cookie值'**,10);

**return** Response::*make*(**'helloWorld!'**,200,[**'Content-Type'**=>**'text/plain'**])->withCookie($myFirstCookie);

**Cookie::forever()**

要设置一个永不过期的cookie可以使用**Cookie::forever(变量名,变量值)**方法，该方法实际上设置一个时效为5年的cookie，eg：

$myFirstCookie=Cookie::*forever*(**'myForeverCookie'**,**'永久cookie值'**);  
 **return** Response::*make*(**'helloWorld!'**,200,[**'Content-Type'**=>**'text/plain'**])->withCookie($myFirstCookie);

**Cookie::queue()**

还可以通过**Cookie::queue(变量名,变量值,存在时间)**方法创建一个cookie并将其添加到下一个response中去，eg：

Cookie::*queue*(**'cookie1'**,**'value1'**,10);  
**return** Response::*make*(**'helloWorld!'**,200,[**'Content-Type'**=>**'text/plain'**]);

#### 3. 获取cookie

服务器端获取cookie可以从请求request中通过**Request::cookie(变量名)**方法获取，或者通过**Cookie::get(变量名)**方法获取，如果不加参数则会以数组形式获取所有的cookie值，eg：

$cookieValue= $request->cookie(**'cookie1'**);  
**return** Response($cookieValue,200,[**'Content-Type'**=>**'text/plain'**]);

$arr=**array**();  
 $arr[**'cookie1'**]=Cookie::*get*(**'cookie1'**);  
 $arr[**'cookie2'**]=Cookie::*get*(**'myCookie22222'**);  
 $arr[**'cookie3'**]=Cookie::*get*(**'myCookieeeeee'**);  
**return** response()->json($arr)->withCallback($request->input(**'callback'**));

#### 4. 删除cookie

**forget(变量名)**方法可以清除cookie，但也需要将该方法的返回值通过**withCookie()**加入到response中，eg;

$deleteCookie=Cookie::*forget*(**'myCookie22222'**);  
**return** response()->make(**'删除成功'**,200)->withCookie($deleteCookie);

### 1.5.6 中间件Middleware

在一个请求真正处理前，我们可能需要对请求做各种各样的判断，才能让它继续向服务器深处传递，中间件可以被看做 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。比如一个活动，只有在指定日期后开始，如果没有到该日期，就重定向到活动宣传页面

#### 创建中间件

中间件一般放在app/Http/Middleware文件下，要使用中间件，需要引入闭包类 use Closure;

在创建的文件中添加中间件，中间件类不需要继承任何父类，只需要在中间件类中添加方法**public function** handle($request, Closure $next)即可，第一个参数接收request请求，中间件对其进行过滤，过滤后的请求通过$next()函数向服务器内部传递，**next()函数将返回一个执行后的响应response**，eg：

**namespace** App\Http\Middleware;  
  
**use** Closure;  
**use** Illuminate\Support\Facades\Auth;  
  
**class** RedirectIfAuthenticated  
{**public function** handle($request, Closure $next, $guard = **null**)  
 {

*//过滤操作*  
 **if** (Auth::*guard*($guard)->check()) {  
 **return** redirect(**'/home'**);  
 }  
 *//符合条件，向下一步传递请求*  
 **return** $next($request);  
 }  
}

有些中间件需要在处理request之后执行一些操作，这时需要先执行请求，再执行中间件逻辑操作，eg：

**namespace** App\Http\Middleware;  
  
**use** Closure;  
  
**class** AfterMiddleware  
{  
 **public function** handle($request, Closure $next)  
 {  
 $response = $next($request);  
 *// 执行中间件动作* **return** $response;  
 }  
}

#### 2.注册中间件

创建中间件后，需要将中间件注册到app\Http\kernel.php中才能使用

**全局中间件**会在每一个HTTP请求期间被执行，只需要将该中间件注册到app\Http\kernel.php的**middleware属性数组**中，eg：

**protected $middleware** = [  
 \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::***class***,  
 \Illuminate\Foundation\Http\Middleware\ValidatePostSize::***class***,  
 \App\Http\Middleware\TrimStrings::***class***,  
 \Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::***class***,  
];

**路由中间件**会在指定的路由路径上添加中间件，只有通过指定路由的url访问服务器才会触发中间件，需要将中间件注册到app\Http\kernel.php的**routeMiddleware属性数组**中，eg：

**protected $routeMiddleware** = [  
 **'auth'** => \Illuminate\Auth\Middleware\Authenticate::***class***,  
 **'auth.basic'** => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::***class***,  
 **'bindings'** => \Illuminate\Routing\Middleware\SubstituteBindings::***class***,  
 **'can'** => \Illuminate\Auth\Middleware\Authorize::***class***,  
 **'guest'** => \App\Http\Middleware\RedirectIfAuthenticated::***class***,  
 **'throttle'** => \Illuminate\Routing\Middleware\ThrottleRequests::***class***,  
 **'myFirMW'**=>\App\Http\Middleware\ActivityMW::***class***];

中间件注册后，可以通过middleware()方法将路由中间件分配到单一路由上，也可以通过1.4.5路由群组来分配到多个路由，eg：

*//分配到单个路由*

Route::*get*(**'wuming'**,[**'uses'**=>**'studentController@jsonp'**])->name(**'bieming'**)->middleware([**'myFirMW'**]);

*//分配到多个路由*Route::*group*([**'middleware'**=>[**'myFirMW'**]],**function** (){  
 Route::*get*(**'activity1'**,[**'uses'**=>**'studentController@activity1'**]);  
 Route::*get*(**'activity2'**,[**'uses'**=>**'studentController@activity2'**]);  
});

**路由中间件组**可以分配多个中间件到路由，只需要在kernel.php的**middlewareGroups属性数组**中添加中间件组即可，eg：

**protected $middlewareGroups** = [  
 **'web'** => [  
 \App\Http\Middleware\EncryptCookies::***class***,  
 \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::***class***,  
 \Illuminate\Session\Middleware\StartSession::***class***,  
 *// \Illuminate\Session\Middleware\AuthenticateSession::class,* \Illuminate\View\Middleware\ShareErrorsFromSession::***class***,  
 \App\Http\Middleware\VerifyCsrfToken::***class***,  
 \Illuminate\Routing\Middleware\SubstituteBindings::***class***,  
 ],  
  
 **'api'** => [  
 **'throttle:60,1'**,  
 **'bindings'**,  
 ],  
];

## 视图View

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，视图一般存放在resources/views文件夹下，eg：

resources/views/hello.blade.php

<html>

<body>

<h1>Hello, {{ $name }}</h1>

</body>

</html>

此时，我们可以通过view()函数来返回它

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

return view('hello' , ['name'=>$name]);

});

**注意: 1.视图也可以传递视图参数，view(视图文件名,视图参数数组)，需要注意的是视图文件中参数需要用"{{}}"包裹起来**

**2.视图还可以存放在resources/views的子目录下，如：视图文件为resources\views\myView\hello.blade.php，代码为return view('** **myView.hello' , ['name'=>$name]);**

**3.有些时候，我们需要判断视图是否存在，然后再输出视图，可以在不带参数的 view 之后使用 exists 方法，若存在返回true，否则返回false，即：**

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

if(view()->exists('myView\hello'))

return view('myView\hello',['name'=>$name]);

else

return '视图不存在';

});

## 模型Model

model是数据库的操作层，一般直接放在app目录下，模型需要继承laravel的model类，具体介绍详见**2.3 Eloquent ORM**，eg：

文件app\ myModel.php

namespace app;

use Illuminate\Database\Eloquent\Model;

class myModel extends Model{

public static function Info(){

return 'Im Model!';

}

}

使用：app\Http\Controllers\myController.php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

use App\myModel;

class myController extends Controller{

function getInfo($id){

return myModel::Info();

}

}

## IOC和DI

Php有两种运行模式，WEB模式和CLI（命令行）模式。当我们在终端敲入php这个命令的时候，使用的是CLI模式；当使用Nginx或者别web服务器作为宿主处理一个到来的请求时，会调用Php运行，此时使用的是WEB模式。当我们请求一个Php文件时，比如Laravel 的public\index.php文件时，Php 为了完成这次请求，会发生5个阶段的生命周期切换：

模块初始化（MINIT），即调用php.ini中指明的扩展的初始化函数进行初始化工作，如mysql扩展。

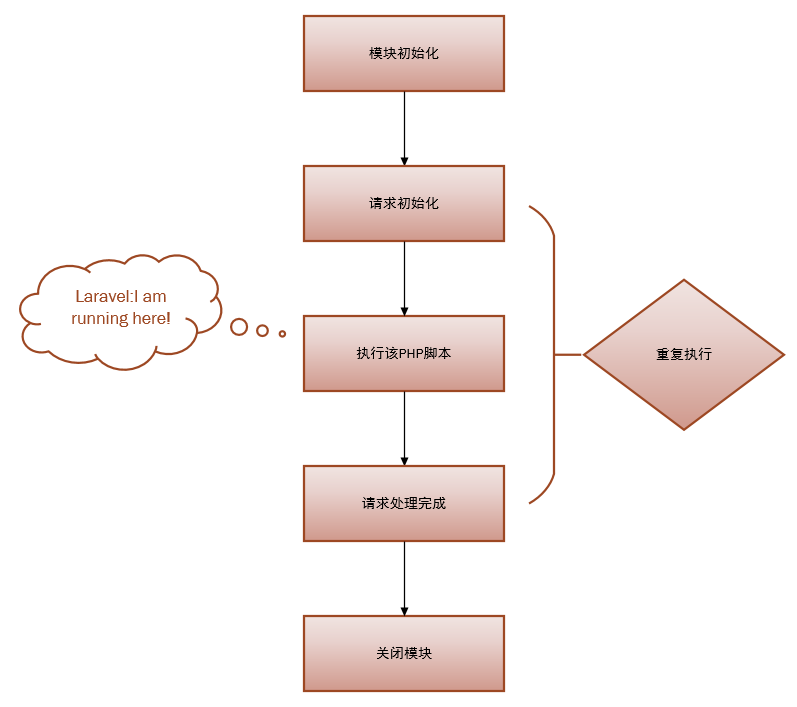
请求初始化（RINIT），即初始化为执行本次脚本所需要的变量名称和变量值内容的符号表，如$\_SESSION变量。

执行该PHP脚本。

请求处理完成(Request Shutdown)，按顺序调用各个模块的RSHUTDOWN方法，对每个变量调用unset函数，如unset $\_SESSION变量。

关闭模块(Module Shutdown) ， PHP调用每个扩展的MSHUTDOWN方法，这是各个模块最后一次释放内存的机会。这意味着没有下一个请求了。

WEB模式和CLI（命令行）模式很相似，区别是：CLI 模式会在每次脚本执行经历完整的5个周期，因为你脚本执行完不会有下一个请求；而WEB模式为了应对并发，可能采用多线程，因此生命周期1和5有可能只执行一次，下次请求到来时重复2-4的生命周期，这样就节省了系统模块初始化所带来的开销。

可以看到，Php生命周期是很对称的。说了这么多，就是为了定位Laravel运行在哪里，没错，Laravel仅仅运行再第三个阶段：

参考：<https://lufficc.com/blog/the-core-conception-of-laravel>

知道这些有什么用？你可以**优化**你的Laravel代码，可以更加**深入**的了解Larave的singleton（单例）。至少你知道了，每一次请求结束，Php的变量都会unset，Laravel的singleton只是在某一次请求过程中的singleton；你在Laravel 中的静态变量也不能在多个请求之间共享，因为每一次请求结束都会unset。理解这些概念，是写高质量代码的第一步，也是最关键的一步。因此记住，Php是一种脚本语言，所有的变量只会在这一次请求中生效，下次请求之时已被重置，而不像Java静态变量拥有全局作用。

# Laravel数据库

laravel中提供了原始查找DB façade、查询构造器和Eloquent ORM三种数据库操作方式

## 原始查找DB façade

### 2.1.1 新建数据表与连接数据库

我们可以通过phpmyadmin或者控制台来创建数据库，然后打开config目录下的database.php文件，修改对应属性，该文件可以定义所有的数据库连接，并指定哪个连接是默认连接，如：

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

'charset' => 'utf8',

'collation' => 'utf8\_unicode\_ci',

接着修改根目录下的.env文件，将响应的属性设置为对应值，如：

DB\_HOST=localhost

DB\_DATABASE=laravel

DB\_USERNAME=root

DB\_PASSWORD=123

### 2.1.2 使用DB facade实现CURD

CURD是指数据库中创建、更新、读取、删除操作，我们可以使用DB门面来实现CURD操作

**语法：**

**DB：：update(' update语句',参数绑定);**

**DB：：select('select语句', 参数绑定);**

**DB：：delete(' delete语句', 参数绑定);**

**DB：：insert(' insert语句', 参数绑定);**

**DB：：statement(' statement语句', 参数绑定);**

eg：

**namespace app\Http\Controllers;  
  
use App\Http\Controllers\Controller;  
use Illuminate\Support\Facades\DB;  
  
class studentController extends Controller  
{  
 public function info(){  
 DB::*insert*('insert into student value(null,"DJ","18","男"),(null,"张三","20","男")');  
 $res=DB::*select*('select *\** from student where sex = ?',["男"]);  
 $name='';  
 foreach($res as $student){  
 $name.=$student->name;  
 }  
 return $name;  
 }  
}**

**注意： 1. 参数绑定都是 where 子句约束中的值。参数绑定可以避免 SQL 注入攻击。参数绑定是数组，eg：**

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

**2. select方法以对象数组的形式返回查询的结果集，查询到的每一行数据是一个数组元素，也是一个对象，可以通过foreach遍历取出**

**3.除了参数绑定，还可以使用命名绑定，命名绑定使用“:名称”占位，参数使用关联数组eg：**

**$res=DB::*select*('select *\** from student where sex = :sex' , ['sex'=>'男']);  
4. select方法返回查询结果集**

**insert方法返回插入是否成功的bool值**

**update方法返回受更新语句影响的行数**

**delete方法返回被删除的行数**

**statement是通用方法，无返回值**

## 查询构造器

### 2.2.1 获取结果集

DB门面提供了table()方法，该方法返回一个查询构造器实例，查询构造器实例提供了很多方法获取结果集：

1. **取出所有行get()**

get()方法返回一个stdClass的数组，该数组的每一个stdClass对象都代表表中的一行数据，该数组的全体元素是结果集中的所有行，eg：

**public function** construct(){  
 $allRes=DB::*table*(**'student'**)->where(**'age'**,**'18'**)->get();  
 $age=**''**;  
 **foreach**($allRes **as** $student){  
 $age.=$student->**age**.**'.'**;  
 }  
 $age=*substr*($age,0,*strlen*($age)-1);  
 **return** $age;  
}

1. **取出第一行first()**

first()方法返回一个单独的stdClass对象而不是stdClass数组，该对象是结果集中的第一行，eg：

**public function** construct(){  
 $allRes=DB::*table*(**'student'**)->where(**'sex'**,**'男'**)->first();  
 $age=$allRes->**age**;  
 **return** $age;  
}

还可以通过value(‘字段名’)方法直接结果集中第一行的某个字段值,eg:

**public function** construct(){  
 $allRes=DB::*table*(**'student'**)->value(**'age'**);  
 **return** $allRes;  
}

1. **获取单个字段的数组pluck()**

如果想要获取包含单个列值的数组，可以使用pluck()方法，eg：

**public function** construct(){  
 $allRes=DB::*table*(**'student'**)->pluck(**'name'**);  
 $name=**''**;  
 **foreach**($allRes **as** $studentName){  
 $name.=$studentName;  
 }  
 **return** $name;  
}

注意：5.3起laravel弃用lists()方法

1. **获取组块结果集chunk()**

该方法一次获取结果集的一部分，即获取子结果集,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

**public function** construct(){  
 $name=**''**;  
 $allRes=DB::*table*(**'student'**)->orderBy(**'name'**)->chunk(2,**function** ($someStudent)**use**(&$name){  
 **foreach**($someStudent **as** $student){  
 $name.=$student->**name**;  
 }  
 $name.=**'|'**;  
 });  
 **return** $name;  
}

**注意： 1.使用chunk(子结果集数量,回调函数 )方法前一定要先用orderBy(‘字段名’)函数排序！**

**2.可以通过在闭包函数中返回 false 来阻止chunk()函数继续分割出组块结果集，eg：**

$allRes=DB::*table*(**'student'**)->orderBy(**'name'**)->chunk(2,**function** ($someStudent)**use**(&$name){  
 **foreach**($someStudent **as** $student){  
 $name.=$student->**name**;  
 }  
 $name.=**'|'**;  
 **return false**;  
});

### 2.2.2 使用查询构造器进行CURD

1. **查询**

通过select方法，我们可以进行查询操作，eg：

**public function** construct(){  
 $name=**''**;  
 $allRes=DB::*table*(**'student'**)->select(**'name as NAME'**,**'age'**,**'sex'**)->where(**'sex'**,**'女'**)->distinct()->get();  
 **foreach** ($allRes **as** $student){  
 $name.=$student->**NAME**.**'.'**;  
 }  
 **return** $name;  
}

**注意： 1.distinct()方法允许强制查询返回不重复的结果集**

**2.如果希望添加查询字段到已有的select查询中，可以使用addSelect方法，eg：**

$query = DB::table('users')->select(**'name as NAME'**,**'age'**);

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

**3.使用join方法可以进行内连接查询，leftJoin进行左外连接查询**

**eg：**

$allRes=DB::*table*(**'student'**)->join(**'score'**,**'student.snum'**,**'='**,**'score.Snum'**)->select(**'student.snum'**,**'Cnum'**,**'student.name'**,**'score'**)->get();

$allRes=DB::*table*(**'student'**)->leftJoin(**'score'**,**'student.snum'**,**'='**,**'score.Snum'**)->select(**'student.snum'**,**'Cnum'**,**'student.name'**,**'score'**)->get();

**4.联合查询使用union()方法将两个查询语句的结果结合**

$someRes=DB::*table*(**'student'**)->select(**'\*'**)->where(**'sex'**,**'='**,**'女'**);  
$allRes=DB::*table*(**'student'**)->select(**'\*'**)->where(**'snum'**,**'='**,**'1'**)->union($someRes)->get();

**5.where子句需要三个参数，第一个参数是列名，第二个参数是操作符，第三个参数是该列要比较的值。还可以传递条件数组，相当于mysql中的and，eg：**

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

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

[**'subscribed', '>', '1'**],

])->get();

**该方法还有orWhere()版本，相当于mysql语法中的or，eg：**

$allRes=DB::*table*(**'student'**)->select(**'\*'**)->where(**'snum'**,**'='**,**'1'**)->orWhere(**'sex'**,**'='**,**'女'**)->get();

**更多where方法如whereBetween请参照手册http://laravelacademy.org/post/6955.html**

**6.聚合函数如count, max, min, avg 和 sum都有对应的方法，可以在构造查询后调用方法,eg：**

$users = DB::*table*(**'users'**)->count();  
$maxPrice = DB::*table*(**'orders'**)->max(**'price'**);

$avgPrice = DB::*table*(**'orders'**)  
 ->where(**'finalized'**, 1)  
 ->avg(**'price'**);

**7.排序orderBy()方法，eg：**

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

**8.** **groupBy 和 having 方法用于对结果集进行分组，eg：**

$users = DB::*table*(**'users'**)  
 ->groupBy(**'account\_id'**)  
 ->having(**'account\_id'**, **'>'**, 100)  
 ->get();

**9.limit()方法可以限定返回的结果集的数目，eg：**

$allRes=DB::*table*(**'student'**)->select(**'\*'**)->where(**'snum'**,**'='**,**'1'**)->orWhere(**'sex'**,**'='**,**'女'**)->limit(2)->get();

1. **插入**

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，该方法返回表示插入是否成功的bool值：

$allRes=DB::*table*(**'student'**)-> insert([

[**'name'**=>**'张纯'**,**'class'**=>**'1'**,**'hobby'**=>**'羽毛球,篮球'**],

[**'name'**=>**'张司'**,**'class'**=>**'2'**,**'hobby'**=>**'羽毛球,篮球'**]

]);

注意： 1. 如果数据表有自增ID，则可以通过insertGetId()方法在插入数据时获取该条记录的自增id，但使用该方法只能一次插入一条记录，eg:

$allRes=DB::*table*(**'student'**)-> insertGetId(

[**'name'**=>**'张纯'**,**'class'**=>**'1'**,**'hobby'**=>**'羽毛球,篮球'**]

);

1. **更新**

查询构建器可以通过使用 update 方法更新已有记录。update()方法返回受更新影响的记录数，eg：

$allRes=DB::*table*(**'student'**)->where(**'name'**,**'='**,**'张纯'**)->update([**'sex'**=>**'女'**,**'class'**=>**'1'**]);

**注意： 1.where()方法必须放在update()之前，先选中，再修改**

**2.increment()/decrement()方法可以增加/减少列值，默认增加1，它的第一个参数是需要修改的列。第二个参数可选，用于控制列值增加/减少的数目，第三个参数可选，是一个数组，用于修改其他字段eg：**

$allRes=DB::*table*(**'student'**)->where(**'name'**,**'='**,**'张纯'**)->increment (**'class'**,2,[**'name'**=>**'改名了'**]);

1. **删除**

查询构建器还可以通过 delete 方法从表中删除记录，可以在调用 delete 方法前通过添加 where 子句来添加约束条件(同update，where方法应放在update方法前面)：

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

如果希望清除整张表，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();

## Eloquent ORM

Laravel 内置的 Eloquent ORM 提供了一个美观、简单的与数据库打交道的实现，每张数据表都对应一个与该表进行交互的“模型”， 模型允许我们在表中进行数据查询，以及插入、更新、删除等操作，Eloquent 模型本质上就是查询构建器，可以在Eloquent查询中使用查询构建器的所有方法

### 2.3.1 定义模型

Eloquent 模型通常位于app 目录下，也可以将其放在其他可以被composer.json 文件自动加载的地方。Eloquent 模型都继承自 Illuminate\Database\Eloquent\Model 类

eg：

**namespace** App;  
**use** Illuminate\Database\Eloquent\Model;  
**class** student **extends** Model{  
}

**注意： 1.在默认情况下，模型名的复数即为对应表名，如上例中模型student默认对应的表名为students表，也可以手动指定：**

**class** student **extends** Model{

**protected $table**=**'students'**;  
}

**2.同样的，每张表默认主键名为 ’id’ ，若表的主键不是id，也可以手动指定：**

**class** student **extends** Model{  
 **protected $table**=**'students'**;  
 **protected $primaryKey**=**'snum'**;  
}

**另外，Eloquent 默认主键字段是自增的整型数据，这意味着主键将会被自动转化为 int 类型，如果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** **public $incrementing**=**false**;

**3.模型默认使用的数据库在根目录的.env文件中已被定义，如果想要为模型指定不同的数据库连接，可以通过 $connection 属性来设置：**

**class** Flight **extends** Model  
{  
 **protected $connection** = **'connection-name'**;  
}

**注意：namespace要注意大小写！**

### 2.3.2 获取模型

将 Eloquent 模型看作功能强大的查询构建器，可以使用查询构造器从模型(student)中查询数据

eg：

$student=student::*select*([**'name as NAME'**,**'sex'**])->where(**'sex'**,**'='**,**'女'**)->get();

$flights = App\Flight::*where*(**'active'**, 1)  
 ->orderBy(**'name'**, **'desc'**)  
 ->take(10)  
 ->get();

还可以直接从模型中取出数据(取出的数据都是数组类型Illuminate\Database\Eloquent\Collection 的一个实例)：

**all() 返回模型表的所有数据**

**find(id) 返回主键值为id的数据记录**

**findOrFail(id) 返回主键值为id的数据记录,如果找不到就报错**

**注意： 1.上述三个方法只能直接对model使用，不能用于查询构建器**

**2.当有多个主键时，使用主键数组传递参数**

eg：

$student=student::*all*()[1]->**name**;

$student=student::*find*(**'2'**)->**name**;

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

$student=student::*select*([**'name as NAME'**,**'sex'**])->where(**'sex'**,**'='**,**'女'**)->all();//错误

### 2.3.3插入模型

**1.save()方法**

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，save方法将返回bool值来显示是否保存成功，eg：

**//student.php**

**class** student **extends** Model  
{

**protected $primaryKey** = **'snum'**;  
 **const *CREATED\_AT*** = **'createTime'**;

**const *UPDATED\_AT*** = **'updateTime'**;

**public $timestamps** = **true**;  
}

**//studentController.php**

$newStuent=**new** student();  
$newStuent->**name**=**'嘻嘻'**;  
$newStuent->**class**=**'1'**;  
$bool=$newStuent->save();  
dd($bool);

**注意： 1.laravel自动管理创建和修改记录的时间戳，默认开启，若不需要laravel管理，则设置$timestamps为false，若需要自定某个字段作为时间戳的存储字段，则修改*CREATED\_AT*和*UPDATED\_AT*为相应的字段名即可，created\_at 和 updated\_at 时间戳在 save 方法被调用时会自动被设置**

1. **如果需要自定义时间戳格式，设置模型中的 $dateFormat 属性。该属性决定日期被如何存储到数据库中，以及模型被序列化为数组或 JSON 时日期的格式**
2. **laravel框架默认的时区是UTC时区，想使用中国时区不止需要修改php.ini，还需要打开config/app.php修改**

**'timezone' => 'Asia/Shanghai'**

**2.create()方法**

该方法本质是对字段进行批量赋值，所以要先允许模型可以批量赋值，即设置**$fillable**属性**(guarded属性与之相反，**不被包含在其中的属性都是可以被赋值的**，二者只能用一个)**为对应字段组成的数组，eg：

**class** student **extends** Model  
{

**protected $primaryKey** = **'snum'**;  
 **protected $fillable**=[**'name'**,**'class'**];  
}

$student=student::*create*([**'name'**=>**'哈哈'**,**'class'**=>**'10'**]);  
dd($student);

### 2.3.4更新模型

**1.save方法**

更新模型时应当先通过find等方法获取模型中的记录，然后对记录进行更新，eg：

$student=student::*find*(**'1'**);  
$student->**name**=**'ketty'**;  
$student->save();

**2.批量更新**

通过查询构造器的方法进行更新操作

$student=student::*where*(**'sex'**,**'='**,**'女'**)->update([**'name'**=>**'我是女生'**,**'class'**=>**'2'**]);

**注意：通过查询构造器批量更新不会触发updated\_at更新数据，这是因为在进行批量更新时并没有从数据库获取模型。**

### 2.3.5删除模型

返回被删除的行数

**1.通过模型删除**

要删除记录，可以调用模型实例collection上的 delete 方法，eg：

$flight =Flight::*find*(1);  
$flight->delete();

**2.根据主键删除**

如果知道要删除的记录的主键，可以调用 destroy 方法直接删除而不需要获取它，eg：

$student=student::*destroy*(**'18'**);

$student=student::*destroy*([**'17'**,**'16'**,**'15'**]);

**3.删除指定数据**

eg：

$student=student::*where*(**'sex'**,**'男'**)->delete();

## 分页

laravel提供了非常易于使用的分页技术

### 2.2.1分页方法

通过paginate()方法实现分页，该方法同查询构件器的get，pluck，first方法以及Eloquent模型的all，find等方法一样可以获取结果集，该方法只有一个参数，就是每页的记录数

eg：

$students=DB::*table*(**'students'**)->paginate(3);  
**return** view(**'homePage'**,[**'students'**=>$students]);

通过paginate()返回的结果使用links或render方法即可显示分页

eg：

**@section('paginate')** {{$students->links()}}  
**@stop**

如何自定义pagenate样式：

https://www.insp.top/article/replace-laravel-pagination-default-template

## 数据迁移

数据迁移就像是数据库版本的git，它能为我们控制数据库的版本，当数据库结构需要改变时我们能很方便地对其进行控制，**迁移位于 database/migrations 目录下，每个迁移文件名都包含时间戳从而允许 Laravel 判断其顺序**

### 2.5.1 生成迁移

使用 Artisan 命令 make:migration 来创建一个新的迁移：

**php artisan make:migration 迁移文件名**

--table 是修改指定的表 --create 是创建新表

eg：

php artisan make:migration create\_users\_table --create=users //创建users表

php artisan make:migration add\_votes\_to\_users\_table --table=users //给users表增加votes字段

### 2.5.2 编写迁移

迁移类包含了两个方法：up 和 down。up 方法用于新增表，列或者索引到数据库，而 down 方法就是 up 方法的反操作，和up 里的操作相反

在这两个方法中使用Schema 门面类进行创建和修改表

1. **创建表create方法**

create 方法接收两个参数，第一个是表名，第二个是定义新表的闭包

eg：

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

$table->increments('id');

});

1. **检查表/列是否存在**

hasTable 和 hasColumn 方法检查表或列是否存在

eg：

if (Schema::hasTable('users')) {

//

}

if (Schema::hasColumn('users', 'email')) {

//

}

1. **连接其他数据库**

如果我们希望修改另一个非当前的数据库中的表，可以使用connection方法

eg：

Schema::connection('foo')->create('users',function ($table) {

$table->increments('id');

});

1. **设置存储引擎**

要设置表的存储引擎，在 Schema 构建器上设置 engine 属性：

eg：

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

$table->engine = 'InnoDB';

$table->increments('id');

});

1. **重命名/删除表**

要重命名一个已存在的数据表，使用 rename 方法：

要删除一个已存在的数据表，可以使用 drop 或 dropIfExists 方法：

eg：

Schema::rename($oldName, $newName);

Schema::drop('users');

Schema::dropIfExists('users');

1. **更新表结构**

使用 Schema 门面上的 table 方法，和 create 方法一样，table 方法接收两个参数

eg：

Schema::table('users', function ($table) {

$table->string('email');

});

具体的构建方法(如上例string())参见：

<http://laravelacademy.org/post/6964.html>

1. **更新表数据**

**1.列修改器**

| **修改器** | **描述** | |
| --- | --- | --- |
| ->after('column') | | 将该列置于另一个列之后 (仅适用于MySQL) |
| ->comment('my comment') | 添加注释信息 | |
| ->default($value) | 指定列的默认值 | |
| ->first() | 将该列置为表中第一个列 (仅适用于MySQL) | |
| ->nullable() | 允许该列的值为NULL | |
| ->storedAs($expression) | 创建一个存储生成列（只支持MySQL） | |
| ->unsigned() | 设置 integer 列为 UNSIGNED | |
| ->virtualAs($expression) | 创建一个虚拟生成列（只支持MySQL） | |

eg:

Schema::table('users', function ($table) {

$table->string('email')->nullable();

});

**2.更新列属性**

change 方法允许我们修改已存在的列为新的类型，或者修改列的属性

eg：

Schema::table('users', function ($table) {

$table->string('name', 50)->change();

});

eg:

use Illuminate\Database\Schema\Blueprint;

use Illuminate\Database\Migrations\Migration;

class CreateFlightsTable extends Migration{

public function up()

{

Schema::create('flights', function (Blueprint $table) {

$table->increments('id');

$table->string('name');

$table->string('airline');

$table->timestamps();

});

}

/\*\*

\* 撤销迁移

\*

\* @return void

\*/

public function down()

{

Schema::drop('flights');

}

}

### 2.5.3 运行、回滚迁移

要运行应用中所有未执行的迁移，可以使用 Artisan 命令提供的 migrate 方法：

**php artisan migrate**

回滚迁移将会回滚最后一批运行的迁移

**php artisan migrate:rollback**

回滚所有迁移

**php artisan migrate:reset**

# blade模板

Blade 视图文件使用 .blade.php 文件扩展并存放在 resources/views文件夹下

## 模板继承

模板继承可以让你定义一个或多个父模板，提供给子模板来进行扩展。

blade指令不以‘;’结尾

|  |  |
| --- | --- |
| **指令名** | **指令作用** |
| **@extends(‘父模板文件名’)** | 继承父模板 |
| **@yield(id[,’可选的默认内容’])** | 预定义可替代模块，内容只能替换不能扩展 |
| **@section(’id’)** | 预定义可替代模块，内容可替换可扩展 |
| **@show** | 允许子页面里面替换(扩展)的内容追加到此处结尾 |
| **@stop** | 执行到此处时将该section的内容替换(扩展)到子页面 |

**注意： 1.@yield()和@section()都可以预定义可替代模块，但区别是前者不能扩展，也就是说@yield类似于默认值，若子页面使用*@section(’@yield的id’,’自定义内容’)*则显示自定义内容，而@section一般配合@show或者@stop使用，内容可扩展，eg：**

**//father.blade.php**

<**title**>**@yield('tit'**,**'hello world!')**</**title**>**//默认为<title>hello world!</title>**

**//child.blade.php**

**@extends('father')**

**@section('tit'**,**'你好，世界！') //子模板替换其为<title>你好，世界!</title>**

**//或者**

**/\***

**@section('tit')** 哈哈，你好世界  
 **@stop //子模板替换其为<title>你好，世界!</title>**

**\*/**

**2.@parent用于指代父模板中的内容(@section和@show之间的部分)，该指令通常用于扩展子模板的内容，但要注意，使用@yield预定义了父模块，子模块不能使用@parent，因为@yield指令不能扩展，eg：**

**//father.blade.php**

<**header**>  
 **@section('head')** whats your name?  
 **@show**</**header**>

**//child.blade.php**

**@extends('father')  
@section('head')** <**span**>**@parent** Im head !</**span**>  
**@stop**

**2.@extends()必须是文件的第一行，前面不能有任何符号甚至空格**

## 模板基础语法

### 3.2.1 子模板中使用php变量

在渲染模板时，有时我们可能需要将php程序运行得出的某个变量值输出显示到子模板中，这时，我们可以通过传递参数来实现，你也可以理解为子模板页面为php程序留下了“插槽”，供其插入变量

**语法：**

**{{变量名}}**

eg：

//father.blade.php

<**header**>  
 **@section('head')** <**span class="myspan"**>{{$question}}</**span**>  
 **@show**</**header**>

//child.blade.php

**@section('head')** <**span**>**@parent** Im head !</**span**>  
**@stop**

//studentController.php

$myQuestion=**'where are you from?'**;  
**return** view(**'myView.child'**,[**'question'**=>$myQuestion]);

**插槽中还可以使用php内置函数，eg：**

**@section('foot')** {{ **'Unix时间戳：'**.*time*()}}  
 <**br**>  
 {{**'时间：'**.*date*(**'Y-m-d H:i:s'**,*time*())}}  
**@show**

**注意： 1.若希望原样输出{{$name}}而不将其作为插槽解析，则使用@符号，eg：**

**@{{ 'Unix时间戳：'.time()}}**

### 3.2.2 引入子视图include

通过@include指令可以引入子视图到本视图页面，eg：

//sonView.blade.php文件

<**p**>引入include视图{{$name}}</**p**>

//father.blade.php

<**footer**>

**@section('foot')**

{{**'Unix时间戳：'**.*time*()}}

<**br**>

{{**'时间：'**.*date*(**'Y-m-d H:i:s'**,*time*())}}

**@show**

</**footer**>

//child.blade.php

**@extends('father')**

**@section('foot')**

**@include('myView.sonView'**,[**'name'**=>**'DJ'**]**)**

**@stop**

### 3.2.3 流程控制

**1.if语句**

if语句可以进行条件判断，流程控制中的变量通过插槽传入，eg：

//father.blade.php

<**footer**>

**@section('foot')**

**默认值**

**@show**

</**footer**>

//child.blade.php

**@extends('father')**

**@section('foot')**

**@include('myView.sonView'**,[**'name'**=>**'DJ'**]**)  
@stop**

//sonView.blade.php文件

<**p**>

**@if(**$name===**'DJ')** Im DJ!  
 **@elseif(**$name=**'zz')** Im zz!  
 **@else** who am I?  
 **@endif**</**p**>

**2.unless语句**

unless意为除非，相当于if取反

//father.blade.php

<**footer**>

**@section('foot')**

**默认值**

**@show**

</**footer**>

//child.blade.php

**@extends('father')**

**@section('foot')**

**@include('myView.sonView'**,[**'name'**=>**'DJ'**]**)  
@stop**

//sonView.blade.php文件

<**p**>

**@unless(**$name!==**'DJ')** Im DJ!  
 **@endunless**</**p**>

**3.for循环**

<**p**>  
 **@for(**$i=0;$i<18;$i++**)** {{**'现在是'**.$i}}  
 <**br**>  
 **@endfor**</**p**>

**4.foreach循环**

//father.blade.php

<**header**>  
 **@section('head')** <**span class="myspan"**>{{$question}}</**span**>  
 **@show**</**header**>

//child.blade.php

**@extends('father'**,[**'question'**=>**'问题'**]**)  
@section('head')  
 @include('myView.sonView'**,[**'students'**=>$students]**)  
@stop**

//sonView.blade.php文件

<**p**>

**@foreach (**$students **as** $student**)** <**p**>我叫{{ $student->**name**}}</**p**>  
 **@endforeach**</**p**>

## 模板中的url

有时候需要在模板中动态生成url，有以下三种方法可以生成url，一般使用route和url方法

### 3.3.1 url()方法

通过路由路径生成url

**语法:**

**url(‘路由路径’);**

eg:

//routes/web.php文件

Route::*get*(**'myURL'**,[**'uses'**=>**'studentController@myUrl'**]);  
Route::*get*(**xixi/orm**,[**'uses'**=>**'studentController@orm'**]);

//app/Http/Controllers/studentController.php文件

**class** studentController **extends** Controller{  
 **public function** orm(){  
 **return 'helloWORLD!'**;  
 }  
 **public function** myUrl(){  
 $students=student::*where*(**'sex'**,**'女'**)->get();  
 **return** view(**'myView.child1'**,[**'students'**=>$students]);  
 }  
}

// child.blade.php文件

**@section('foot')** <**a href="**{{url(**xixi/orm**)}}**"**>点击url</**a**>  
**@stop**

//点击后即可从 http://localhost:8080/myclass/myURL跳转到http://localhost:8080/myclass/xixi/orm

### 3.3.2action()方法

通过指定控制器的方法名生成url，直接指定控制器的方法处理，跳过路由

eg：

**@section('foot')** <**a href="**{{action(**'studentController@orm'**)}}**"**>点击</**a**>  
**@stop**

### 3.3.3 route()方法

通过路由的名称生成url

eg：

//routes/web.php文件

Route::*get*(**'myURL'**,[**'uses'**=>**'studentController@myUrl'**]);

Route::*get*(**'xixi/orm'**,[**'uses'**=>**'studentController@orm'**])->name(**'bieming'**);

// child.blade.php文件

**@section('foot')** <**a href="**{{route(**'bieming'**)}}**"**>点击</**a**>  
**@stop**

## 模板使用静态资源

我们在使用laravel时，经常需要使用css，图片等静态资源，在blade中使用静态资源，在实际项目中，这些静态资源都是放在public文件下的，但视图view放在resource文件夹下，并且laravel是会将静态资源打包的，这时，我们就需要使用到**url类中的asset(url)函数**

asset中的url参数默认根目录是public文件夹，eg：

<**link rel="stylesheet" href="**{{ URL::*asset*(**'css/bootstrap.css'**) }}**"**>

# Composer

## Composer简介

现代编程语言都有**包管理器**，比如java的maven，nodeJS的npm等，包管理器用来管理别人写好的包和该包所需的依赖，比如我们一般不会为每一个项目都写一个jQuery，否则就是重复劳动，Composer则是laravel提供的**依赖管理工具**，它涵盖包和库，功能更加强大，避免复制粘贴，提高开发效率

Composer安装完成后，由于国内访问可能被墙，因此需要配置中文镜像，首先查看Composer当前的镜像地址

**composer config -g repo-packagist//如果提示Do not run Composer as root/super user!，请切换非root用户执行该指令**

系统全局配置中文镜像：

**composer config -g repo.packagist composer https://packagist.phpcomposer.com**

## 使用Composer

**1.创建composer.json文件**

可以通过**composer init**命令一步步引导进行创建配置文件，配置文件是.json文件，是json格式，内部内容一般如下：

{

"name": "meta/silk",

"description": "another e-commerce website",

"keywords": ["silk", "online shop", "good"],

"version": "1.0.0",

"require": {

"php": ">=5.3.0"

},

"type" "library",

"homepage": "http://www.xxx.com ",

"time": "2014-12-30",

"license": "MIT",

"authors": [{

"name": "Elvis Lim",

"email": "elvis@xxx.com",

"homepage": "http://www.xxx.com",

"role": "Engineer"

}]

}

name 包的名称，通常包名包含两部分，并且以 / 分隔。斜杆前面部分，代表包的所有者，斜杆后面部分代表包的名称

description 简单描述

keywords 本包的关键字，一组准确地关键字易于使用者的搜索和使用

version 包的版本号

require 本包依赖的包

require-dev 只会在开发过程中使用的包依赖，正式发布不需要

type 包的类型，用于定制安装逻辑，主要有：

library：默认值。它将复制文件到 vendor 目录。

project：它表示这是个项目，而不是库。比如像 Symfony 标准版这种应用。

metapackage：一个含有依赖的空包，能触发安装，但不包含文件，不会向文件系统写任何东西。

composer-install：为其他的定制类型的包提供安装器的包。

homepage 项目的网站 URL

license 许可证，涉及版权法律相关

authors 作者

autoload 它将决定你的包的自动加载相关配置，一般有三种自动加载：

Classmap方式自动加载

使用Files方式(ps:通常作为函数库的载入方式（而非类库）)

PSR0/4加载方式

**2.常用composer命令**

**(1)查找库/包**

**composer search 库/包名**

eg:

composer search laravel

composer search monolog

**(2)查看库/包信息**

根据查到的库/包名，查询相关信息(如版本)

**composer show 参数 库/包名**

eg:

composer show -all laravel/laravel

composer show -all monolog/monolog

**(3)申明依赖**

根据查到的库/包名和版本信息，申明本包的依赖，**打开创建的composer.json文件，给require属性添加值**

**"require":**

**{**

**"包/库名1": "版本号",**

**"包/库名2": "版本号"**

**}**

eg:

"require":

{

"monolog/monolog": "2.0.x-dev"

}

**(4)安装本包所需依赖**

根据已经写好的composer.json配置文件，下载相关依赖，**注意:该命令一定要cd到composer.json所在的文件夹下执行**

**composer install**

**(5)更新和删除依赖**

修改composer.json配置文件的require，删除相关依赖，**注意:该命令一定要cd到composer.json所在的文件夹下执行**

**composer update**

## Composer安装laravel

通过Composer安装laravel框架有两种方法：

1. **create-project 命令**

**composer create-project --prefer-dist laravel/laravel 项目名称**

1. **Laravel 安装器**

首先，通过 Composer 安装 Laravel 安装器：

**composer global require "laravel/installer"**

安装完成后，通过简单的 laravel new 命令即可在当前目录下创建一个新的 Laravel 应用，如：

laravel new myBlog

**Composer安装laravel的常见问题**

1. 没有设置app\_key值

在控制台进入laravel应用根目录，输入php artisan key:generate，检查.env文件中的app\_key值是否为空

1. 检查根目录下是否存在.htacess文件，若没有，就将public/.htacess拷贝过来

## Artisan控制台

### 4.4.1 Artisan简介

Artisan是laravel自带的命令行工具，它提供了许多有用的命令来帮助我们进行开发

### 4.4.2 Artisan基本用法

必须在laravel根目录下执行

**1.查看所有可用artisan命令**

**php artisan list**

**2.生成命令**

**php artisan make:类型 控制器名**

类型可以为:

make:auth

make:command

make:controller

make:event

make:factory

make:job

make:listener

make:mail

make:middleware

make:migration

make:model

make:notification

make:policy

make:provider

make:request

make:resource

make:rule

make:seeder

make:test

**eg:**

php artisan make:controller myController

php artisan make: model student

php artisan make:middleware activity

**3.创建控制器**

**php artisan make:controller 控制器名**

**eg:**

php artisan make:controller myController

## 用户认证auth

生成auth所需文件只需在artisan命令行输入**php artisan make:auth** ，它会为我们生成一些对应的文件，如：routes/web.php文件中的

Auth::routes();

Route::get('/home', 'HomeController@index')->name('home');

resources\views文件夹下会多出一个auth目录和layout目录，里面包含了登陆页面的相关代码

# 常用功能

## 文件上传

文件系统配置文件位于 config/filesystems.php中

### 5.1.1 磁盘

在文件系统配置文件中的disks属性可以配置所有“磁盘”， 每个磁盘描述了特定的存储驱动和存储位置， root属性配置文件存储路径，driver属性是驱动，共有"local", "ftp", "s3", "rackspace"等几种驱动方式，laravel给我们默认配置好了三个磁盘空间，如果需要可以自己配置磁盘

**public 磁盘**是laravel默认配置的磁盘之一，用于存储可以被公开访问的文件，默认情况下， public 磁盘使用 local 驱动并将文件存储在 storage/app/public目录下，如果希望该文件存储可被访问，可通过**php artisan storage:link**指令生成指向storage/app/public目录的软链，然后即可通过asset("storage/文件名")来访问存储文件，eg：

php artisan storage:link

<a href='{{asset("storage/联想笔记本.txt")}}'>点击下载</a>

**local磁盘**是laravel默认配置的磁盘之一，相比于public它的默认存储空间在storage/app文件夹下

### 5.1.2 磁盘交互

Storage 门面类用于和我们配置的任意磁盘进行交互

|  |  |
| --- | --- |
| **语法** | **方法描述** |
| disk(‘磁盘名’) | 获取磁盘 |
| put(‘新文件路径’ , ’文件内容’) | 将文件内容存储到文件，文件内容可以是字符串也可以是fopen方法读取的文件内容 |
| get(‘文件相对磁盘路径’) | 读取文件内容 |
| exists(‘文件相对路径’) | 判断文件是否存在 |
| path(‘文件相对路径’) | 返回磁盘存储根路径+文件路径 |
| copy(‘old路径’ , ’new路径’) | 复制文件 |

eg：

Storage::disk('local')->put('/new/xixi.txt' , Storage::disk('local')->get('public\.gitignore'));

//将文件.gitignore的内容拷贝到xixi.txt中

Storage::disk('local')->copy('public\.gitignore', 'new/hihi.txt');

### 5.1.3 文件上传

4.文件上传

## 邮件

邮件功能配置在config/mail.php中，Laravel 为 SMTP、Mailgun、SparkPost、Amazon SES、PHP 的 mail 函数，以及sendmail 提供了驱动

基于 API 的驱动如 Mailgun(收费服务) 和 SparkPost(免费服务) 通常比 SMTP 服务器更简单、更快，所以如果可以的话，尽可能使用这些服务，所有的 API 驱动要求应用已经安装 Guzzle HTTP 库，你可以通过 Composer 包管理器来安装它

### 5.2.1 邮件功能配置

在config/mail.php文件中，driver属性指定默认的邮件驱动，一般默认为SMTP，如果需要改成SparkPost，则需要将driver 选项值改为 sparkpost，并将config/service.php中的响应选项修改，具体参见<http://laravelacademy.org/post/6899.html>

host是发送邮件的主机地址，from属性配置发件邮箱和发件人名称，username是smtp服务器的账号，password是smtp服务器的密码，这些属性都用了env()方法配置了默认值，真实值都在.env文件中配置，在.env中，我们一般设置

**MAIL\_DRIVER=smtp**

**MAIL\_HOST=你的smtp服务器地址**

**MAIL\_PORT=465**

**MAIL\_USERNAME=你的发件邮箱账号**

**MAIL\_PASSWORD=你的smtp服务授权码**

**MAIL\_ENCRYPTION=ssl**

**注意: 1.邮箱必须开通SMTP服务，参考**

[**https://help.aliyun.com/knowledge\_detail/51622.html**](https://help.aliyun.com/knowledge_detail/51622.html)

**2.** **开通服务生成的授权码要记住，填写在密码的位置**

### 5.2.2 邮寄类

在 Laravel 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，默认是没有该目录的，当我们创建第一个邮寄类时会产生

**php artisan make:mail 邮寄类名**

在创建好的邮寄类中有一个build方法，我们将在这个方法中完成邮件的内容和发送

1. **配置发件人**

通过from方法可以指定发送者，eg：

1. **2**

## 多用户登录

有时候，我们需要做两个用户登录，一个前台，一个后台，这时候

参考<http://laravelacademy.org/post/5925.html>