安装Laravel

Laravel5的环境要求如下：

- PHP介于5.6.4和7.1之间。
- OpenSSL PHP扩展
- PDO PHP扩展
- Mbstring PHP扩展
- Tokenizer PHP扩展
- XML PHP扩展

严格来说，Laravel无需安装过程，这里所说的安装其实就是把Laravel框架放入WEB运行环境（前提是你的WEB运行环境已经OK），可以通过下面几种方式获取和安装Laravel。

Composer安装

Laravel5支持使用Composer安装，如果还没有安装 Composer，你可以按 Composer安装 中的方法安装。在 Linux 和 Mac OS X 中可以运行如下命令：

```
curl -sS https://getcomposer.org/installer | php
mv composer.phar /usr/local/bin/composer
```

在 Windows 中，你需要下载并运行 Composer-Setup.exe。

如果遇到任何问题或者想更深入地学习 Composer。请参考 Composer 文档（英文）。Composer 中文。

如果你已经安装有 Composer 请确保使用的是最新版本，你可以用 composer self-update 命令更新 Composer 为最新版本。

然后在命令行下面，切换到你的web根目录下面并执行下面的命令：

```
composer create-project --prefer-dist laravel/laravel blog "5.3.*"
```

表示安装最新的5.3版本。

如果出现错误提示，请根据提示操作或者参考Composer中文文档。

无论你采用什么方式获取的Laravel框架，现在只需要做最后一步来验证是否正常运行。

在浏览器中输入地址：

```
http://localhost/blog/public/
```

如果浏览器输出如图所示：

> 如果是mac或者linux环境，请确保storage目录有可写权限

如果你无法正常运行并显示Laravel的欢迎页面，那么请检查下你的服务器环境：

> - PHP5.6以上版本
> - WEB服务器是否正常启动

目录结构

下载最新版框架后，解压缩到web目录下面，可以看到初始的目录结构如下：

```
project  应用部署目录
├─app                                    应用目录
│  ├─Http                                Http目录
│  │  ├─Controllers                      控制器
│  │  ├─Middleware                       中间件
│  │  ├─Providers                        服务提供者
├─bootstrap                              框架的启动和自动载入配置
│  ├─cache              路由和服务缓存文件
├─config                                 配置文件
├─database             数据库迁移
├─public                                 WEB 部署目录（对外访问目录）
│  ├─index.php                           应用入口文件
│  └─.htaccess                           用于 apache 的重写
├─resources                              resources目录
│  ├─lang                                语言包目录
│  ├─views                               视图目录
├─routes                                 路由目录
├─storage              存储目录
│  ├─app                                 应用的运行时目录
│  ├─logs              日志文件
│  ├─ framework                          存储框架生成的文件和缓存
├─vendor                                 第三方类库目录Composer
├─composer.json                          composer 定义文件
├─readme.md                              README 文件
```

> 如果是mac或者linux环境，请确保`storage`目录有可写权限

5.3的部署建议是public目录作为web目录访问内容，其它都是web目录之外，当然，你必须要修改public/index.php中的相关路径。如果没法做到这点，请记得设置目录的访问权限或者添加目录列表的保护文件。

URL访问

Laravel URL 基于 自定义 路由 访问

路由配置

```
routes/web.php
```

默认路由

```
Route::get('/', function () {
    return view('welcome');
});
```

URL访问

```
http://serverName/index.php
```

> 需要注意：路由规则（区分大小写）

隐藏入口文件

在Laravel5.3中，出于优化的URL访问原则，还支持通过URL重写隐藏入口文件，下面以Apache为例说明隐藏应用入口文件index.php的设置。

下面是Apache的配置过程，可以参考下：

1、`httpd.conf`配置文件中加载了`mod_rewrite.so`模块

2、`AllowOverride None` 将None改为 `All`

3、在应用入口文件同级目录添加`.htaccess`文件，内容如下：

```
<IfModule mod_rewrite.c>
    <IfModule mod_negotiation.c>
        Options -MultiViews
    </IfModule>

    RewriteEngine On

    # Redirect Trailing Slashes If Not A Folder...
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteRule ^(.*)/$ /$1 [L,R=301]

    # Handle Front Controller...
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^ index.php [L]

    # Handle Authorization Header
    RewriteCond %{HTTP:Authorization} .
    RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
</IfModule>
```

命名空间

Laravel5采用命名空间方式定义和自动加载类库文件，有效的解决了多模块和Composer类库之间的命名空间冲突问题，并且实现了更加高效的类库自动加载机制。

> 如果不清楚命名空间的基本概念，可以参考PHP手册：PHP命名空间

> 特别注意的是，如果你需要调用PHP内置的类库，或者第三方没有使用命名空间的类库，记得在实例化类库的时候加上 \，例如：
>
> ```
> // 错误的用法
> $class = new stdClass();
> $xml  =  new SimpleXmlElement($xmlstr);
> // 正确的用法
> $class = new \stdClass();
> $xml  =  new \SimpleXmlElement($xmlstr);
> ```

在Laravel5.3中，只需要给类库正确定义所在的命名空间，并且命名空间的路径与类库文件的目录一致，那么就可以实现类的自动加载，从而实现真正的惰性加载。

例如， `\Illuminate\Support\Facades\File`类的定义为：

```
namespace Illuminate\Support\Facades;

class File
{
}
```

如果我们实例化该类的话，应该是：

```
$class = new \Illuminate\Support\Facades\File();
```

系统会自动加载该类对应路径的类文件，其所在的路径是

`vendor/laravel/framework/src/Illuminate/Support/Facades/File.php`

> 5.3默认的目录规范是小写，类文件命名是驼峰法，并且首字母大写。

配置目录

```
├─config
|   ├─app.php
|   └─auth.php
|   └─broadcasting.php
|   └─cache.php
|   └─compile.php
|   └─database.php
|   └─filesystems.php
|   └─mail.php
|   └─queue.php
|   └─services.php
|   └─session.php
|   └─view.php
|   └──...更多配置
```

配置格式

Laravel支持多种格式的配置格式，但最终都是解析为PHP数组的方式。

PHP数组定义

返回PHP数组的方式是默认的配置定义格式，例如：

```
//项目配置文件
return [
    'compiled' => realpath(storage_path('framework/views'))
];
```

配置参数名不区分大小写（因为无论大小写定义都会转换成小写），新版的建议是使用小写定义配置参数的规范。

还可以在配置文件中可以使用二维数组来配置更多的信息，例如：

```
//项目配置文件
return [
    'sqlite' => [
        'driver' => 'sqlite',
        'database' => env('DB_DATABASE', database_path('database.sqlite')),
        'prefix' => '',
    ]
]
```

二级配置

配置参数支持二级，例如，下面是一个二级配置的设置和读取示例：

```
$config = [
    'user'  =>  [
        'type'  =>  1,
        'name'  =>  'laravel',
    ],
    'db'    =>  [
        'type'      =>  'mysql',
        'user'      =>  'root',
        'password'  =>  '',
    ],
];
// 设置配置参数
Config::set($config);
// 读取二级配置参数
echo Config::get('user.type');
// 或者使用助手函数
echo config('user.type');
```

读取配置

设置完配置参数后，就可以使用get方法读取配置了，例如：

```
echo Config::get('文件名.配置参数1');
```

系统定义了一个助手函数config，以上可以简化为：

```
echo config('文件名.配置参数1');
```

读取所有的配置参数：

```
dd(config());
```

或者你需要判断是否存在某个设置参数：

```
Config::has('文件名.配置参数2');
```

如果需要读取二级配置，可以使用：

```
echo Config::get('文件名.配置参数.二级参数');
echo config('文件名.配置参数.二级参数');
```

动态配置

使用`set`方法动态设置参数，例如：

```
Config::set('配置参数','配置值');
// 或者使用助手函数
config('配置参数','配置值');
```

也可以批量设置，例如：

```
Config::set([
    '配置参数1'=>'配置值',
    '配置参数2'=>'配置值'
]);
// 或者使用助手函数
config([
    '配置参数1'=>'配置值',
    '配置参数2'=>'配置值'
]);
```

环境变量配置

Laravel5.3支持使用环境变量配置。

在开发过程中，可以在应用根目录下面的`.env`来模拟环境变量配置，`.env`文件中的配置参数定义格式采用`ini`方式，例如：

```
APP_ENV=local
APP_DEBUG=true
```

注意，环境变量不支持数组参数，如果需要使用数组参数可以，使用下划线分割定义配置参数名：

```
DB_USERNAME=root
DB_PASSWORD=secret
```

获取环境变量的值可以使用下面的两种方式获取：

```
env('DB_USERNAME');
env('DB_PASSWORD');
```

可以支持默认值，例如：

```
// 获取环境变量 如果不存在则使用默认值root
env('DB_USERNAME','root');
```

可以直接在应用配置中使用环境变量，例如：

```
return [
    'host' => env('DB_HOST', '127.0.0.1')
]
```

路由定义

路由定义采用`Illuminate\Support\Facades\Route`类的`rule`方法注册，通常是在应用的路由配置文件`routes/web.php`进行注册

基本路由

```
// 注册路由到News控制器的read方法
Route::get('new/{id}', "NewsController@read");
```

我们访问：

```
http://serverName/new/5
```

命名路由

```
Route::get('new/{id}', "NewsController@read")->name('new');
```

> 注意，路由命名标识必须唯一，定义后可以用于URL的快速生成。

请求类型

```
Route::post('new/{id}', "NewsController@update");
```

表示定义的路由规则在POST请求下才有效。

请求类型包括：

类型	描述
GET	GET请求
POST	POST请求
PUT	PUT请求
PATCH	PATCH请求
DELETE	DELETE请求
OPTIONS	OPTIONS请求
*	任何请求类型

系统提供了为不同的请求类型定义路由规则的简化方法，例如：

```
Route::get('new/{id}','NewsController@read'); // 定义GET请求路由规则
Route::post('new/{id}','NewsController@update'); // 定义POST请求路由规则
Route::put('new/{id}','NewsController@update'); // 定义PUT请求路由规则
Route::delete('new/{id}','NewsController@delete'); // 定义DELETE请求路由规则
Route::any('new/{id}','NewsController@read'); // 所有请求都支持的路由规则
```

如果要定义get和post请求支持的路由规则，也可以用：

```
Route::match(['get', 'post'], 'new/{id}', 'NewsController@read');
```

规则表达式

规则表达式通常包含静态地址和动态地址，或者两种地址的结合，例如下面都属于有效的规则表达式：

```
Route::get('/','SiteController@index'); // 首页访问路由
Route::get('my','MemberController@myinfo'); // 静态地址路由
Route::get('new/{year}/{month}/{day}','NewsController@read'); // 静态地址和动态地址结合
Route::get('{user}/{blog_id}','BlogController@read'); // 全动态地址
```

规则表达式的定义以/为参数分割符（无论你的PATH_INFO分隔符设置是什么，请确保在定义路由规则表达式的时候统一使用/进行URL参数分割）。 每个参数中以“`{}`”包裹的参数都表示动态变量，并且会自动绑定到操作方法的对应参数。

可选定义

支持对路由参数的可选定义，例如：

```
Route::get('blog/{year}/{month?}','BlogController@archive');
```

`{month?}`变量用`{ ?}`包含起来后就表示该变量是路由匹配的可选变量。

以上定义路由规则后，下面的URL访问地址都可以被正确的路由匹配：

```
http://serverName/index.php/blog/2015
http://serverName/index.php/blog/2015/12
```

> 可选参数只能放到路由规则的最后，如果在中间使用了可选参数的话，后面的变量都会变成可选参数。

变量规则

Laravel5.3支持在规则路由中为变量用正则的方式指定变量规则，弥补了动态变量无法限制具体的类型问题，并且支持全局规则设置。使用方式如下：

全局变量规则

如果您希望路径参数始终受给定正则表达式的约束，则可以使用该`pattern`方法。您应该在以下`boot`方法中定义这些模式`app\Providers\RouteServiceProvider`

```
/**
 * Define your route model bindings, pattern filters, etc.
 *
 * @return void
 */
public function boot()
{
    Route::pattern('id', '[0-9]+');

    parent::boot();
}
```

局部变量规则

局部变量规则，仅在当前路由有效：

```
Route::get('new/{name}', 'NewsController@read')->where('name', '\w+');
```

如果一个变量同时定义了全局规则和局部规则，局部规则会覆盖全局变量的定义。

路由地址

解析规则是从操作开始解析，然后解析控制器，例如：

```
// 路由到news控制器
Route::get('blog/{id}','BlogController@read')
```

Blog类定义如下：

```
namespace App\Http\Controllers;

class BlogController extends Controller
{
    public function read($id){
        return 'read:'.$id;
    }
}
```

资源路由

5.3支持设置RESTFul请求的资源路由，方式如下：

```
Route::resource('blog','BlogController');
```

设置后会自动注册7个路由规则，如下：

请求类型	生成路由规则	操作方法	路由名称
GET	`blog`	index	blog.index
GET	`blog/create`	create	blog.create
POST	`blog`	store	blog.store
GET	`blog/{id}`	show	blog.show
GET	`blog/{id}/edit`	edit	blog.edit
PUT	`blog/{id}`	update	blog.update
DELETE	`blog/{id}`	destroy	blog.destroy

以支持下面的URL访问：

```
http://serverName/blog
http://serverName/blog/128
http://serverName/blog/28/edit
```

```
namespace App\Http\Controllers;
class BlogController extends Controller {
    public function index(){
    }

    public function show($id){
    }

    public function edit($id){
    }
}
```

可以改变默认的id参数名，例如：

```
Route::resource('blog','BlogController',['var'=>['blog'=>'blog_id']]);
```

控制器的方法定义需要调整如下：

```
namespace App\Http\Controllers;
class BlogController extends Controller {
    public function index(){
    }

    public function show($blog_id){
    }

    public function edit($blog_id){
    }
}
```

也可以在定义资源路由的时候限定执行的方法（标识），例如：

```
// 只允许index show edit update 四个操作
Route::resource('blog','BlogController',['only'=>['index','show','edit','update']]);
// 排除index和destroy操作
Route::resource('blog','BlogController',['except'=>['index','destroy']]);
```

路由分组

路由组允许您跨大量路由共享路由属性，例如中间件或命名空间，而无需在每条路由上定义这些属性。共享属性以数组格式指定为方法的第一个参数。`Route::group`

参数	说明
middleware	中间件
namespace	命名空间
domain	子域路由
prefix	路由前缀

中间件

要将中间件分配给组中的所有路由，可以使用`middleware`组属性数组中的键。中间件按照它们在数组中列出的顺序执行：

```
Route::group(['middleware' => 'auth'], function () {
    Route::get('/', function ()    {
        // Uses Auth Middleware
    });

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

命名空间

路由组的另一个常见用例是使用`namespace`组数组中的参数将相同的PHP命名空间分配给一组控制器：

```
Route::group(['namespace' => 'Admin'], function () {
    // Controllers Within The "App\Http\Controllers\Admin" Namespace
});
```

子域路由

路由组也可用于处理子域路由。可以像路由URI一样为子域分配路由参数，允许您捕获子域的一部分以供路由或控制器使用。可以使用`domain`组属性数组上的键指定子域：

```
Route::group(['domain' => '{account}.myapp.com'], function () {
    Route::get('user/{id}', function ($account, $id) {
        //
    });
});
```

路由前缀

该`prefix`组属性可被用于前缀与给定URI的组中的每个路由。例如，您可能希望为组中的所有路径URI添加前缀`admin`

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

闭包定义

我们可以使用闭包的方式定义一些特殊需求的路由，而不需要执行控制器的操作方法了，例如：

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

参数传递

闭包定义的时候支持参数传递，例如：

```
Route::get('hello/{name}', function ($name) {
    return 'Hello,'.$name;
});
```

规则路由中定义的动态变量的名称 就是闭包函数中的参数名称，不分次序。

因此，如果我们访问的URL地址是：

```
http://serverName/hello/laravel
```

则浏览器输出的结果是：

```
Hello,laravel
```

命名路由

命名路由允许方便地生成特定路由的URL或重定向。您可以通过将 `name`方法链接到路径定义来指定路径的名称：

```
Route::get('user/profile', function () {
    //
})->name('profile');
```

您还可以为控制器操作指定路由名称：

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

生成指定路由的URL

为给定路径指定名称后，可以在生成URL或通过全局`route`函数重定向时使用路径名称：

```
// Generating URLs...
$url = route('profile');

// Generating Redirects...
return redirect()->route('profile');
```

如果命名路由定义了参数，则可以将参数作为第二个参数传递给route函数。给定的参数将自动插入URL中的正确位置：

```
Route::get('user/{id}/profile', function ($id) {
    //
})->name('profile');

$url = route('profile', ['id' => 1]);
```

生成的url地址为：

`http://serverName/user/1/profile`

路由信息

您可以使用current□currentRouteName以及currentRouteAction对方法Route门面访问有关处理传入的请求路由信息：

```
namespace App\Http\Controllers;
use Illuminate\Support\Facades\Route;
class UserController extends Controller
{

    public function index()
    {
        $route = Route::current();
        echo 'name: ' . Route::currentRouteName() . "<br/>";
        echo 'action: ' . Route::currentRouteAction() . "<br/>";
    }
}
```

路由定义，如下所示：

```
Route::resource('user','UserController');
```

则浏览器输出的结果是：

```
name: user.index
action: App\Http\Controllers\UserController@index
```

介绍

您可能希望使用Controller类组织此行为，而不是将所有请求处理逻辑定义为路由文件中。控制器可以将相关的请求处理逻辑分组到单个类中。控制器存储在目录中。`app/Http/Controllers`

控制器定义

一个典型的控制器类定义如下：

```
namespace App\Http\Controllers;
class SiteController extends Controller
{
    public function index(){
        return 'index';
    }
}
```

控制器类文件的实际位置是

```
app/Http/Controllers/SiteController.php
```

路由定义，如下所示：

```
Route::get('site', 'SiteController@index');
```

现在，当请求与指定的路由URI匹配时，将执行该类的`index`方法`SiteController`。当然，路由参数也将传递给方法。

多级控制器

支持任意层次级别的控制器，并且支持路由，例如：

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

use App\Http\Controllers\Controller;

class BlogController extends Controller
{
    public function index()
    {
        return "index";
    }

    public function create()
    {
        return "create";
    }

    public function edit($id)
    {
        return "edit";
    }
}
```

该控制器类的文件位置为：

```
app/Http/Controllers/One/BlogController.php
```

如果要在路由定义中使用多级控制器，可以使用：

```
Route::get('blog/v1/create','One\BlogController@create');
```

访问地址可以使用

```
http://serverName/blog/v1/create
```

资源控制器

资源控制器可以让你轻松的创建RESTFul资源控制器，可以通过命令行生成需要的资源控制器，例如：

```
php artisan make:controller BlogController --resource
```

然后你只需要为资源控制器注册一个资源路由：

```
Route::resource('blog', 'BlogController');
```

设置后会自动注册7个路由规则，如下：

请求类型	生成路由规则	操作方法	路由名称
GET	`blog`	index	blog.index
GET	`blog/create`	create	blog.create
POST	`blog`	store	blog.store
GET	`blog/{id}`	show	blog.show
GET	`blog/{id}/edit`	edit	blog.edit
PUT	`blog/{id}`	update	blog.update
DELETE	`blog/{id}`	destroy	blog.destroy

请求信息

要通过依赖项注入获取当前HTTP请求的实例，您应该在控制器方法上键入提示类。传入的请求实例将由服务容器自动注入：`Illuminate\Http\Request`

```php
namespace App\Http\Controllers;
use Illuminate\Http\Request;

class UserController extends Controller
{
    public function store(Request $request)
    {
        $name = $request->input('name');
    }
}
```

获取URL信息

```php
// 获取URL地址中的PATH_INFO信息 不含后缀
echo 'path: ' .$request->path() . '<br/>';
// 获取当前URL地址 不含QUERY_STRING
echo 'url: ' . $request->url() . "<br/>";
// 获取包含域名的完整URL地址
echo 'full url: ' . $request->fullUrl() . "<br/>";
```

输出结果为：

```
path: user
url: http://laravel.cn/user
full url: http://laravel.cn/user?name=php
```

获取请求参数

```php
echo '<pre>请求方法：' . $request->method() . '<br/>';
echo '访问ip地址：' . $request->ip() . '<br/>';
echo '请求参数<br/>';
var_export($request->all());
echo '<br/>' . '请求参数：仅包含name<br/>';
var_export($request->only(['name']));
echo '<br/>' . '请求参数：排除name<br/>';
var_export($request->except(['name']));
```

输出结果为：

```
请求方法：GET
访问ip地址：::1
请求参数：
array (
  'name' => 'php',
)
请求参数：仅包含name
array (
  'name' => 'php',
)
请求参数：排除name
array (
)
```

输入变量

概述

可以通过`Request`对象完成全局输入变量的检测、获取和安全过滤，支持包括`$_GET``$_POST``$_SESSION``$_COOKIE`等系统变量，以及文件上传信息。

检测变量是否设置

您应该使用该`has`方法来确定请求中是否存在值。如果值存在且不是空字符串，则`has`返回该方法`true`：

```
if ($request->has('name')) {
    //
}
```

变量获取

变量获取使用`Illuminate\Http\Request`类的方法包括：

方法	描述
get	获取 $_GET 变量
session	获取 $_SESSION 变量
cookie	获取 $_COOKIE 变量
file	获取 $_FILES 变量
route	获取 路由信息

检索所有输入数据

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

检索输入值

```
$request->input('name');
```

您可以将默认值作为方法的第二个参数传递`input`。如果请求中不存在请求的输入值，则返回此值：

```
$name = $request->input('name', 'Sally');
```

使用包含数组输入的表单时，使用“ 点 ”表示法来访问数组：

```
$name = $request->input('products.0.name');

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

检索JSON输入值

向您的应用程序发送JSON请求时，`input`只要请求标头正确设置为，就可以通过该方法访问JSON数据。您甚至可以使用“ 点 ”语法来挖掘JSON数组：`Content-Typeapplication/json`

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

检索Cookie

```
$value = $request->cookie('name');
```

检索输入数据的一部分

```
$request->only('username', 'password');
```

或者使用数组方式`$request->only(['username', 'password']);`

排除部分变量

```
$request->except('credit_card');
```

或者使用数组方式 `$request->except(['credit_card']);`

请求类型

获取请求类型

在很多情况下面，我们需要判断当前操作的请求类型是GET□POST□PUT□DELETE或者HEAD，一方面可以针对请求类型作出不同的逻辑处理，另外一方面有些情况下面需要验证安全性，过滤不安全的请求。

用法如下

```php
// 是否为 GET 请求
if ($request->isMethod('get')) echo "当前为 GET 请求";
// 是否为 POST 请求
if ($request->isMethod('post')) echo "当前为 POST 请求";
// 是否为 PUT 请求
if ($request->isMethod('put')) echo "当前为 PUT 请求";
// 是否为 DELETE 请求
if ($request->isMethod('delete')) echo "当前为 DELETE 请求";
// 是否为 Ajax 请求
if ($request->ajax()) echo "当前为 Ajax 请求";
// 是否为 Pjax 请求
if ($request->pjax()) echo "当前为 Pjax 请求";
// 是否为 HEAD 请求
if ($request->isMethod('head')) echo "当前为 HEAD 请求";
// 是否为 Patch 请求
if ($request->isMethod('patch')) echo "当前为 PATCH 请求";
// 是否为 OPTIONS 请求
if ($request->isMethod('options')) echo "当前为 OPTIONS 请求";
```

连接数据库

> Laravel内置了抽象数据库访问层，把不同的数据库操作封装起来，我们只需要使用公共的Db类进行操作，而无需针对不同的数据库写不同的代码和底层实现□Db类会自动调用相应的数据库驱动来处理。采用PDO方式，目前包含了Mysql□SqlServer□PgSQL□Sqlite等数据库的支持。

环境变量配置

根目录`.env`中修改下面的配置参数：

```
// 数据库类型
DB_CONNECTION=mysql
// 服务器地址
DB_HOST=127.0.0.1
// 数据库连接端口
DB_PORT=3306
// 数据库名
DB_DATABASE=laravel
 // 数据库用户名
DB_USERNAME=root
// 数据库密码
DB_PASSWORD=
```

配置文件定义

常用的配置方式是`config/database.php`中修改下面的配置参数：

```
'mysql' => [
    'driver' => 'mysql',
    'host' => env('DB_HOST', '127.0.0.1'),
    'port' => env('DB_PORT', '3306'),
    'database' => env('DB_DATABASE', 'forge'),
    'username' => env('DB_USERNAME', 'forge'),
    'password' => env('DB_PASSWORD', ''),
    'charset' => 'utf8',
    'collation' => 'utf8_unicode_ci',
    'prefix' => '',
    'strict' => true,
    'engine' => null,
]
```

基本使用

配置数据库连接后，可以使用`Illuminate\Support\Facades\DB`类运行查询。DB提供了每种类型的查询方法：`select`、`update`、`insert`、`delete`，和`statement`

```
DB::select('select * from users where active = ?', [1]);
DB::insert('insert into users (id, name) values (?, ?)', [1, 'Dayle']);
```

也支持命名占位符绑定，例如：

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

可以使用多个数据库连接，使用

```
DB::connection($config)->select('...');
```

$config是数据库连接的配置参数名

查询数据

基本查询

查询一个数据使用：

```
DB::table('users')->where('name', 'John')->first();
```

查询数据集使用：

```
DB::table('users')->select('name', 'email as user_email')->get();
```

值和列查询

```
DB::table('users')->where('name', 'John')->value('email')
```

查询某一列的值

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

foreach ($titles as $title) {
    echo $title;
}
```

自定义键列

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

foreach ($roles as $name => $title) {
    echo $title;
}
```

数据集分批处理

如果你需要处理成千上百条数据库记录，可以考虑使用chunk方法，该方法一次获取结果集的一小块，然后填充每一小块数据到要处理的闭包，该方法在编写处理大量数据库记录的时候非常有用。

比如，我们可以全部用户表数据进行分批处理，每次处理 100 个用户记录：

```
DB::table('users')->orderBy('id')->chunk(100, function ($users) {
    foreach ($users as $user) {
        //
    }
});
```

你可以通过从闭包函数中返回false来中止对数据集的处理：

```
DB::table('users')->orderBy('id')->chunk(100, function ($users) {
    // 处理结果集...
    return false;
});
```

添加数据

添加一条数据

使用 `Db` 类的 `insert` 方法向数据库提交数据

```
DB::table('users')->insert(
    ['email' => 'john@example.com', 'votes' => ]
);
```

添加数据后如果需要返回新增数据的自增主键，可以使用`insertGetId`方法：

```
$id = DB::table('users')->insertGetId(
    ['email' => 'john@example.com', 'votes' => ]
);
```

> insertGetId 方法添加数据成功返回添加数据的自增主键

添加多条数据

```
DB::table('users')->insert([
    ['email' => 'taylor@example.com', 'votes' => ],
    ['email' => 'dayle@example.com', 'votes' => ]
]);
```

更新数据

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

自增或自减一个字段的值

`increment/decrement` 如不加第二个参数，默认值为1

```
// votes 字段加 1
DB::table('users')->increment('votes');
// votes 字段加 5
DB::table('users')->increment('votes', 5);
// votes 字段减 1
DB::table('users')->decrement('votes');
// votes 字段减 5
DB::table('users')->decrement('votes', 5);
```

您还可以在操作期间指定要更新的其他列：

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

删除数据

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

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

如果您希望截断整个表，这将删除所有行并将自动递增ID重置为零，您可以使用以下truncate方法：

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

查询方法

条件查询方法

where方法

可以使用where方法进行AND条件查询：

```
DB::table('users')
                    ->where('votes', '>', 100)
                    ->where('user_id', '>', 5)
                    ->get();
```

orWhere方法

使用orWhere方法进行OR查询：

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

混合查询

where方法和orWhere方法在复杂的查询条件中经常需要配合一起混合使用，下面举个例子：

```
DB::table('users')
            ->where('name', '=', 'John')
            ->orWhere(function ($query) {
                $query->where('votes', '>', 100)
                      ->where('title', '<>', 'Admin');
            })
            ->get();
```

生成的sql语句类似于下面：

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

> 注意闭包查询里面的顺序，而且第一个查询方法用where或者orWhere是没有区别的。

查询语法

查询表达式

查询表达式支持大部分的SQL查询语法，也是`Laravel`查询语言的精髓，查询表达式的使用格式：

```
where('字段名','表达式','查询条件');
orWhere('字段名','表达式','查询条件');
whereBetween/whereNotBetween('字段名','数组');
whereIn/whereNotIn('字段名','数组');
whereNull/whereNotNull('字段名');
```

（=）等于

```
where('id','=',100);
```

和下面的查询等效

```
where('id',100);
```

表示的查询条件就是 `id = 100`

（<>） 不等于

例如：

```
where('id','<>',100);
```

表示的查询条件就是 `id <> 100`

（>）大于

例如：

```
where('id','>',100);
```

表示的查询条件就是 `id > 100`

（>=）大于等于

```
where('id','>=',100);
```

表示的查询条件就是 `id >= 100`

（<）小于

```
where('id','<',100);
```

表示的查询条件就是 `id < 100`

（<=）小于等于

```
where('id','<=',100);
```

表示的查询条件就是 `id <= 100`

like同sql的LIKE

```
where('name', 'like', 'T%')
```

查询条件就变成 name like 'T%'

whereBetween/whereNotBetween

```
whereBetween('votes', [1, 100]);
whereNotBetween('votes', [1, 100]);
```

查询条件就变成 votes BETWEEN 1 AND 100

whereIn/whereNotIn

```
whereIn('id', [1, 2, 3]);
whereNotIn('id', [1, 2, 3]);
```

查询条件就变成 id NOT IN (1,2, 3)

[NOT] NULL

查询字段是否（不）是Null例如：

```
whereNull('name');
whereNull('title');
whereNotNull('name');
```

如果你需要查询一个字段的值为字符串null或者not null应该使用：

```
where('title','=', 'null');
where('name','=', 'not null');
```

聚合查询

在应用中我们经常会用到一些统计数据，例如当前所有（或者满足某些条件）的用户数、所有用户的最大积分、用户的平均成绩等等Laravel为这些统计操作提供了一系列的内置方法，包括：

方法	说明
count	统计数量
max	获取最大值
min	获取最小值
avg	获取平均值
sum	获取总分

用法示例：

获取用户数：

```
DB::table('users')->count();
```

或者根据字段统计：

```
Db::table('users')->count('id');
```

获取用户的最大积分：

```
Db::table('users')->max('score');
```

获取积分大于0的用户的最小积分：

```
Db::table('users')->where('score','>','0')->min('score');
```

获取用户的平均积分：

```
Db::table('users')->avg('score');
```

统计用户的总成绩：

```
Db::table('users')->sum('score');
```

时间查询

方法	说明
whereDate	用于将列的值与日期进行比较
whereMonth	用于将列的值与一年中的特定月份进行比较
whereDay	用于将列的值与一个月中的特定日期进行比较
whereYear	用于将列的值与特定年份进行比较
whereTime	用于将列的值与特定时间进行比较

用法示例：

```
DB::table('users')->whereDate('created_at', '2016-12-31')->get();

DB::table('users')->whereMonth('created_at', '12')->get();

DB::table('users')->whereDay('created_at', '31')->get();

DB::table('users')->whereYear('created_at', '2016')->get();

DB::table('users')->whereTime('created_at', '=', '11:20:45')->get();
```

最后生成的SQL语句是

```
select * from `users` where date(`created_at`) = 2016-12-31
```

```
select * from `users` where month(`created_at`) = 12
```

```
select * from `users` where day(`created_at`) = 31
```

```
select * from `users` where year(`created_at`) = 2016
```

```
select * from `users` where time(`created_at`) = 11:20:45
```

高级查询

闭包查询

```
DB::table('users')
            ->where('name', '=', 'John')
            ->orWhere(function ($query) {
                $query->where('votes', '>', 100)
                      ->where('title', '<>', 'Admin');
            })
            ->get();
```

最后生成的SQL语句是

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

子查询

whereIn

```
DB::table('users')
            ->whereIn('id',function ($query) {
                $query->select('id')
                      ->from('profile')
                     ->where('status',1);
})
             ->get();
```

```
select * from `users` where `id` in (select `id` from `profile` where `status` = 1)
```

whereExists

```
DB::table('users')
            ->whereExists(function ($query) {
                $query->select(DB::raw(1))
                      ->from('orders')
                      ->whereRaw('orders.user_id = users.id');
            })
            ->get();
```

最后生成的SQL语句是

```
select * from users
where exists (
    select 1 from orders where orders.user_id = users.id
)
```

链式操作

数据库提供的链式操作方法，可以有效的提高数据存取的代码清晰度和开发效率，并且支持所有的CURD操作。

使用也比较简单，假如我们现在要查询一个Users表的满足状态为1的前10条记录，并希望按照用户的创建时间排序，代码如下：

```
DB::table('users')
    ->where('status',1)
    ->take(10)
    ->orderBy('create_time')
    ->get();
```

这里的where、orderBy和take方法就被称之为链式操作方法，除了get方法必须放到最后一个外（因为get方法并不是链式操作方法），链式操作的方法调用顺序没有先后，例如，下面的代码和上面的等效：

```
DB::table('users')
    ->orderBy('create_time')
    ->take(10)
    ->where('status',1)
    ->get();
```

其实不仅仅是查询方法可以使用连贯操作，包括所有的CURD方法都可以使用，例如：

```
DB::table('users')
    ->where('id',1)
    ->select('id','name','email')
    ->first();
DB::table('users')
    ->where('status',1)
    ->where('id',1)
    ->delete();
```

where

where方法的用法是ThinkPHP查询语言的精髓，也是Laravel ORM的重要组成部分和亮点所在，可以完成包括普通查询、表达式查询、快捷查询、区间查询、组合查询在内的查询操作。where方法的参数支持字符串和数组，虽然也可以使用对象但并不建议。

表达式查询

新版的表达式查询采用全新的方式，查询表达式的使用格式：

```
Db::table('users')
    ->where('id','>',1)
    ->where('name','laravel')
    ->get();
```

更多的表达式查询语法，可以参考查询语法部分。

数组条件

可以通过数组方式批量设置查询条件。

普通查询

最简单的数组查询方式如下：

```
$map['name'] = 'laravel';
$map['status'] = 1;
// 把查询条件传入查询方法
Db::table('users')->where($map)->get();
```

最后生成的SQL语句是

```
SELECT * FROM users WHERE `name`='laravel' AND status=1
```

表达式查询

可以在数组条件中使用查询表达式，例如：

```
$map[]  = ['id','>',1];
$map[]  = ['mail','like','%laravel@qq.com%'];
Db::table('users')->where($map)->get();
```

字符串条件

使用字符串条件直接查询和操作，例如：

```
Db::table('users')->whereRaw('type=1 AND status=1')->get();
```

最后生成的SQL语句是

```
SELECT * FROM users WHERE type=1 AND status=1
```

table

table方法主要用于指定操作的数据表。

用法

例如：

```
// table方法必须指定完整的数据表名
DB::table('users')->where('name', 'John')->first();
```

select

select方法属于模型的连贯操作方法之一，主要目的是标识要返回或者操作的字段，可以用于查询和写入操作。

用于查询

在查询操作中select方法是使用最频繁的。

```
DB::table('users')->select('name','email')->get();
```

这里使用select方法指定了查询的结果集中包含id,title,content三个字段的值。执行的SQL相当于：

```
SELECT name,email FROM table
```

可以给某个字段设置别名，例如：

```
DB::table('users')->select('name', 'email as user_email')->get()
```

执行的SQL语句相当于：

```
SELECT name,email as user_email FROM table
```

使用SQL函数

可以在select方法中直接使用函数，例如：

```
DB::table('users')->select(DB::raw('id,SUM(score)')) ->get();
```

执行的SQL相当于：

```
SELECT id,SUM(score) FROM table
```

> 除了get方法之外，所有的查询方法，包括first等都可以使用select方法。

addSelect

如果您已有一个查询构建器实例，并且希望在其现有select子句中添加一列，则可以使用以下`addSelect`方法：

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

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

orderBy

orderBy方法属于模型的连贯操作方法之一，用于对操作的结果排序。 用法如下：

```
DB::table('users')->orderBy('name', 'desc')->get();
```

注意：连贯操作方法没有顺序，可以在get方法调用之前随便改变调用顺序。

支持对多个字段的排序，例如：

```
DB::table('users')->orderBy('name', 'desc')-->orderBy('id', 'asc')->get();
```

> 当你的orderBy排序中使用了SQL函数的时候，请使用DB::raw例如：

```
DB::table('users')->orderBy(DB::raw('rand()'))->get();
```

limit

limit方法也是模型类的连贯操作方法之一，主要用于指定查询和操作的数量，特别在分页查询的时候使用较多□Laravel的limit方法可以兼容所有的数据库驱动类的。

限制结果数量

例如获取满足要求的10个用户，如下调用即可：

```
DB::table('users')
                ->where('status',1)
                ->select('name','email')
                ->limit(5)
                ->get();
```

limit方法也可以用于写操作，例如更新满足要求的3条数据：

```
Db::table('users')
->where('score',100)
->limit(3)
->update(['level'=>'A']);
```

分页查询

用于文章分页查询是limit方法比较常用的场合，例如：

```
DB::table('article')->offset(10)->limit(25)->get();
```

表示查询文章数据，从第10行开始的25条数据（可能还取决于where条件和orderBy排序的影响 这个暂且不提）。

你也可以这样使用，作用是一样的：

```
DB::table('article')->skip(10)->take(25)->get();
```

对于大数据表，尽量使用limit限制查询结果，否则会导致很大的内存开销和性能问题。

groupBy

GROUP BY方法也是连贯操作方法之一，通常用于结合合计函数，根据一个或多个列对结果集进行分组 。

groupBy方法只有一个参数，并且只能使用字符串。

例如，我们都查询结果按照用户id进行分组统计：

```
Db::table('users')
    ->selectRaw('user_id,username,max(score)')
    ->groupBy('user_id')
    ->get();
```

生成的SQL语句是：

```
select user_id,username,max(score) from `users` group by `user_id`
```

也支持对多个字段进行分组，例如：

```
Db::table('users')
    ->selectRaw('user_id,test_time,username,max(score)')
    ->groupBy('user_id')->groupBy('test_time')
    ->get();
```

生成的SQL语句是：

```
select user_id,test_time,username,max(score) from `users` group by `user_id`,
`test_time`)
```

having

所述groupBy和having方法可以用于组查询结果。该having方法的签名是类似于的where方法：

```
Db::table('users')
    ->selectRaw('test_time,username,max(score)')
    ->groupBy('user_id')
    ->having('test_time','>',3)
    ->get();
```

生成的SQL语句是：

```
select test_time,username,max(score) from `users` group by `user_id` having `test_time`
> 3
```

该havingRaw方法可用于将原始字符串设置为having子句的值。例如：

```
Db::table('users ')
    ->selectRaw('username,max(score)')
    ->groupBy('user_id')
    ->havingRaw('count(test_time)>3')
    ->get();
```

生成的SQL语句是：

```
select username,max(score) from `users` group by `user_id` having count(test_time)>3
```

join

join通常有下面几种类型，不同类型的join操作会影响返回的数据结果。

- Inner Join: 等同于 JOIN（默认的JOIN类型）,如果表中有至少一个匹配，则返回行
- Left Join: 即使右表中没有匹配，也从左表返回所有的行
- Right Join: 即使左表中没有匹配，也从右表返回所有的行
- Cross Join: 交叉连接在第一张表和被连接表之间生成一个笛卡尔积

内部Join

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

生成的SQL语句是：

```
select `users`.*, `contacts`.`phone`, `orders`.`price` from `users` inner join
`contacts` on `users`.`id` = `contacts`.`user_id` inner join `orders` on `users`.`id` =
`orders`.`user_id`
```

左Join

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

生成的SQL语句是：

```
select * from `users` left join `posts` on `users`.`id` = `posts`.`user_id`
```

右Join

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

```
select * from `users` right join `posts` on `users`.`id` = `posts`.`user_id`
```

交叉Join

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

生成的SQL语句是：

```
select * from `sizes` cross join `colours`
```

union

UNION操作用于合并两个或多个 SELECT 语句的结果集。 使用示例：

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

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

生成的SQL语句是：

```
select * from `users` where `last_name` is null) union (select * from `users` where
`first_name` is null
```

union all

例如：

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

$users = DB::table('users ')
            ->where('status',1)
            ->unionAll($first)
            ->get();
```

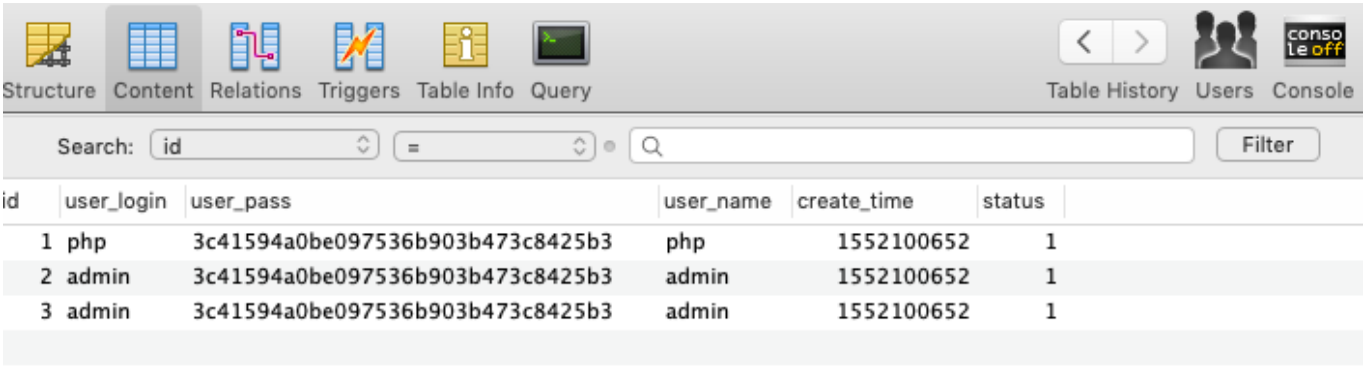
生成的SQL语句是：

```
(select * from `users` where `status` = 1) union all (select * from `users` where
`score` = 100)
```

distinct

DISTINCT 方法用于返回唯一不同的值

例如数据库表中有以下数据



id	user_login	user_pass	user_name	create_time	status
1	php	3c41594a0be097536b903b473c8425b3	php	1552100652	1
2	admin	3c41594a0be097536b903b473c8425b3	admin	1552100652	1
3	admin	3c41594a0be097536b903b473c8425b3	admin	1552100652	1

以下代码会返回`user_login`字段不同的数据

```
Db::table('users')->distinct()->select('user_login')->get();
```

生成的SQL语句是：

```
select distinct `user_login` from `users`
```

返回以下数组

```
[
    {
        "user_login": "php"
    },
    {
        "user_login": "admin"
    }
]
```

lock

查询构建器还包含一些函数，可帮助您对`select`语句执行“悲观锁定”。要使用“共享锁”运行语句，可以`sharedLock`在查询中使用该方法。在您的事务提交之前，共享锁可防止选定的行被修改：

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

生成的SQL语句是：

```
select * from `users` where `votes` > 100 lock in share mode
```

或者，您可以使用该`lockForUpdate`方法“for update”锁可防止修改行或使用另一个共享锁选择：

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

生成的SQL语句是：

```
select * from `users` where `votes` > 100 for update
```

数据集

返回的数据集对象是`Illuminate\Support\Collection`，提供了和数组无差别用法，并且另外封装了一些额外的方法。

可以直接使用数组的方式操作数据集对象，例如：

```
// 获取数据集
$users=  Db::table('users')->get();
// 直接操作第一个元素
$item  = $users[];
// 获取数据集记录数
$count = count($users);
// 遍历数据集
foreach($users as $user){
   echo $user->name;
   echo $user->id;
}
```

需要注意的是，如果要判断数据集是否为空，不能直接使用`empty`判断，而必须使用数据集对象的`isEmpty`方法判断，例如：

```
$users=  Db::table('users')->get();
if($users->isEmpty()){
    echo '数据集为空';
}
```

`Collection`类包含了下列主要方法：

方法	描述
isEmpty	是否为空
toArray	转换为数组
all	所有数据
merge	合并其它数据
diff	比较数组，返回差集
flip	交换数据中的键和值
intersect	比较数组，返回交集
keys	返回数据中的所有键名
pop	删除数据中的最后一个元素
shift	删除数据中的第一个元素
reduce	通过使用用户自定义函数，以字符串返回数组
reverse	数据倒序重排
chunk	数据分隔为多个数据块
each	给数据的每个元素执行回调
filter	用回调函数过滤数据中的元素
sort	对数据排序
shuffle	将数据打乱
slice	截取数据中的一部分
toJson	转换为`Json`格式

助手函数

如果你只是需要渲染模板输出的话，可以使用系统提供的助手函数`view`，可以完成相同的功能：

```
return view('admin.banner.item');
```

模板赋值

助手函数

```
return view('admin.config.list', [
    'name'  => 'Laravel',
    'email' => 'laravel@qq.com'
]);
```

compact方法

```
$banner = new Banner();
$item = $banner->find($id);
return view('admin.banner.item', compact('item'));
```

share 方法

```
view()->share('base_url', config("app.url"));
view()->share('assets_url', config("app.url"));
```

模板渲染

view

```
echo view('admin.orders.delivery_list',compact('lists'));
exit;
```

表示读取的模板是

```
/resources/views/admin/orders/batch_delivery.blade.php
```

手动记录

一般情况下，系统的日志记录是自动的，无需手动记录，但是某些时候也需要手动记录日志信息，Log类提供了3个方法用于记录日志。

方法	描述
Log::debug	调试，用于调试信息
Log::warning	警告，程序可以运行但是还不够完美的错误
Log::error	错误，一般会导致程序的终止

例如：

```
Log::error("签名错误...");
Log::warning('call back: 订单'.$data['out_trade_no'].'支付成功开始支付后业务');
Log::debug("call back return xml:" . $xmlData);
```

表示写入的日志是

```
storage/logs/laravel-error-2019-05-14.log
storage/logs/laravel-warning-2019-05-14.log
storage/logs/laravel-debug-2019-05-14.log
```

独立日志

```
Log::useFiles(storage_path("logs/crontab.txt"));
```

表示写入的日志是

```
storage/logs/crontab.txt
```

分页

Laravel5内置了分页实现，要给数据添加分页输出功能在5.0变得非常简单，可以直接在Model类查询的时候调用paginate方法：

```
class banner extends Illuminate\Database\Eloquent\Model
{
}
```

```
$sql = Banner::query();
$pager = $sql->orderBy('banner_sort', 'ASC')->orderBy('banner_id', 'DESC')->paginate();
return view('admin.banner.list', compact('pager'));
```

模板文件中分页输出代码如下：

```
@foreach ($pager as $item)
    <tr>
        <td>{{  $item['banner_id'] }}</td>
        <td>{{  $item['banner_name'] }}</td>
    </tr>
@endforeach

{{ $pager->render() }}
```

上传文件

Laravel5.0对文件上传的支持更加简单。

> 内置的上传只是上传到本地服务器，上传到远程或者第三方平台的话需要自己扩展。

假设表单代码如下：

```
<form action="/index/index/upload" enctype="multipart/form-data" method="post">
<input type="file" name="image" /> <br>
<input type="submit" value="上传" />
</form>
```

然后在控制器中添加如下的代码：

```
public function upload(Request $request){
    // 获取表单上传文件
    $file = $request->file();

    if (!$file)
    {
        return array('error'=>1,'message'=>"文件上传失败，请检查后重试");
    }

    foreach ($file as $k => $v)
    {
        $tmpName = $v->getPathName();
        $fileExtension = $v->getClientOriginalExtension();
        $filePath = md5_file($tmpName) . '.' . $fileExtension;

        // 移动到框架应用根目录/public/uploads/ 目录下
        $v->move(base_path('public')    .   '/uploads', $filePath);
     }
}
```

部分 Linux 主机设置了 open_basedir□可将用户访问文件的活动范围限制在指定的区域，通常是入口文件根目录的路径） 选项，导致 Laravel5 访问白屏或者报错

如果把Laravel5部署在了LAMP/LNMP环境上很有可能出现白屏的情况，这个时候需要开启 php 错误提示来判断是否是因为设置了open_basedir选项出错。

打开 php.ini 搜索 display_errors，把 Off 修改为 On就开启了 php 错误提示，这时再访问之前白屏的页面就会出现错误信息。如果错误信息如下那么很有可能就是因为open_basedir的问题。

Warning: require(): open_basedir restriction in effect. File(/home/wwwroot/chunice.com/thinkphp/start.php) is not within the allowed path(s): (
/index.php on line **21**

Warning: require(/home/wwwroot/chunice.com/thinkphp/start.php): failed to open stream: Operation not permitted in **/home/wwwroot/chu**

Fatal error: require(): Failed opening required '/home/wwwroot/chunice.com/public/../thinkphp/start.php' (include_path='.:/usr/local/php/lib/p

php.ini 修改方法

把权限作用域由入口文件目录修改为框架根目录

打开 `php.ini` 搜索 `open_basedir`,把

```
open_basedir = "/home/wwwroot/laravel5/public/:/tmp/:/var/tmp/:/proc/"
```

修改为

```
open_basedir = "/home/wwwroot/laravel5/:/tmp/:/var/tmp/:/proc/"
```

如果你的 `php.ini` 文件的 `open_basedir` 设置选项是被注释的或者为 none那么你需要通过 Apache 或者 Nginx 来修改

> tips: *php.ini 文件通常是在 /usr/local/php/etc 目录中，当然了这取决于你 LAMP 环境配置*

Apache 修改方法

Apache 需要修改 `httpd.conf` 或者同目录下的 `vhost` 目录下 `你的域名.conf` 文件，如果你的生成环境是 LAMP 一键安装包配置那么多半就是直接修改 `你的域名.conf` 文件

```
apache
├─vhost
    ├─www.laravel.com.conf
    ├─......
├─httpd.conf
```

打开 你的域名.conf 文件 搜索 `open_basedir`,把

```
php_admin_value open_basedir
"/home/wwwroot/www.laravel.com/public/:/tmp/:/var/tmp/:/proc/"
```

然后重新启动 `apache` 即可生效

> tips: *域名.conf 文件通常是在 /usr/local/apache/conf 目录中，当然了这取决于你 LAMP 环境配置*

Nginx/Tengine 修改方法

`Nginx` 需要修改 `nginx.conf` 或者 `conf/vhost` 目录下 你的域名.conf 文件，如果你的生成环境是 LNMP/LTMP 一键安装包配置那么多半就是直接修改 你的域名.conf 文件

```
nginx
├─conf
    ├─vhost
        ├─www.laravel.com.conf
    ├─nginx.conf
    ├─......
├─nginx.conf
```

打开 你的域名.conf 文件 搜索 `open_basedir`,把

```
fastcgi_param  PHP_VALUE
"open_basedir=/home/wwwroot/www.laravel.com/public/:/tmp/:/proc/";
```

修改为

```
fastcgi_param  PHP_VALUE  "open_basedir=/home/wwwroot/www.laravel.com/:/tmp/:/proc/";
```

然后重新启动 `Nginx` 即可生效

> tips: *域名.conf 文件通常是在 /usr/local/nginx/conf/vhost 目录中，当然了这取决于你 LNMP/LTMP 环境配置*

fpm/fastcgi user.ini 修改方法

打开 项目根目录下找到 user.ini 文件，搜索 `open_basedir`,把

```
open_basedir=/home/wwwroot/www.laravel.com/public/:/tmp/:/proc/
```

修改为

```
open_basedir=/home/wwwroot/www.laravel.com/:/tmp/:/proc/
```

然后重新启动 `web 服务器` 即可生效

URL重写

可以通过URL重写隐藏应用的入口文件`index.php`,下面是相关服务器的配置参考：

[Apache]

1. httpd.conf配置文件中加载了mod_rewrite.so模块
2. AllowOverride None 将None改为 All
3. 把下面的内容保存为.htaccess文件放到应用入口文件的同级目录下

```
<IfModule mod_rewrite.c>
  Options +FollowSymlinks -Multiviews
  RewriteEngine On

  RewriteCond %{REQUEST_FILENAME} !-d
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteRule ^(.*)$ index.php/$1 [QSA,PT,L]

</IfModule>
```

[IIS]

如果你的服务器环境支持ISAPI_Rewrite的话，可以配置httpd.ini文件，添加下面的内容：

```
RewriteRule (.*)$ /index\.php\?s=$1 [I]
```

在IIS的高版本下面可以配置web.Config□在中间添加rewrite节点：

```
<rewrite>
 <rules>
 <rule name="OrgPage" stopProcessing="true">
 <match url="^(.*)$" />
 <conditions logicalGrouping="MatchAll">
 <add input="{HTTP_HOST}" pattern="^(.*)$" />
 <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" />
 <add input="{REQUEST_FILENAME}" matchType="IsDirectory" negate="true" />
 </conditions>
 <action type="Rewrite" url="index.php/{R:1}" />
 </rule>
 </rules>
 </rewrite>
```

[Nginx]

```
location / { //……省略部分代码
  try_files $uri $uri/ /index.php?$query_string;
}
```