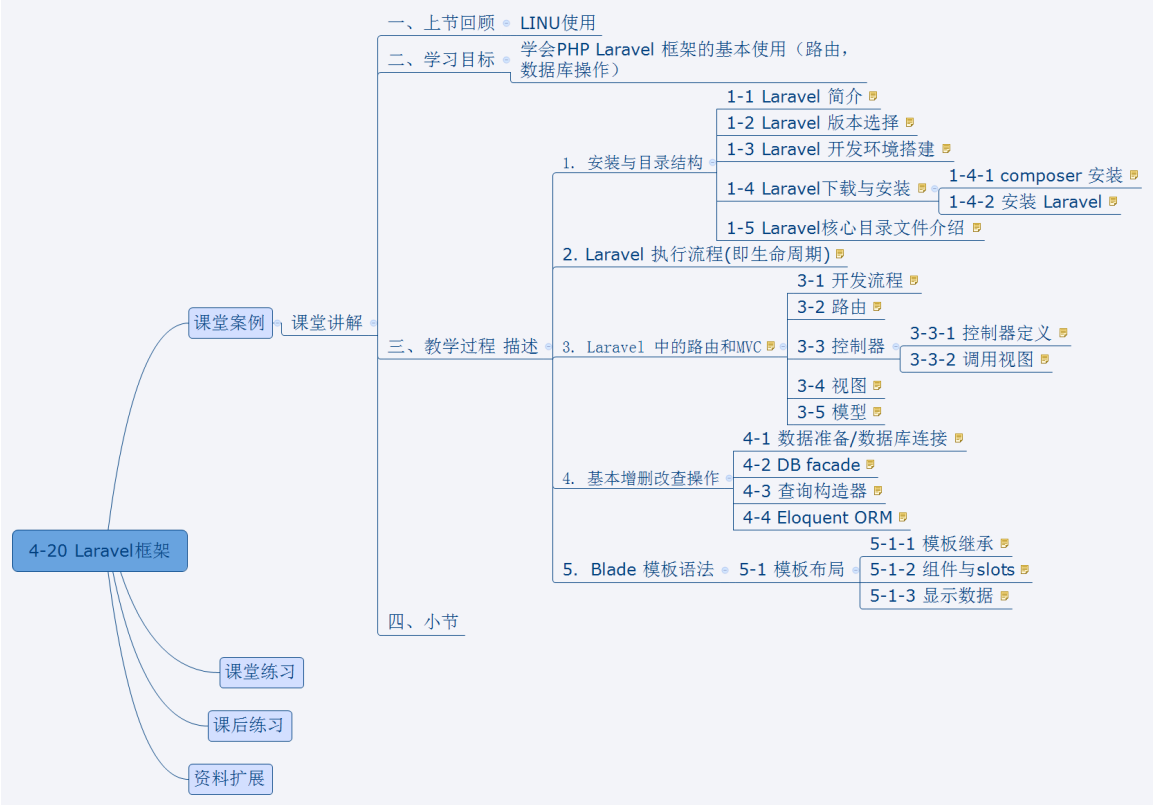
4-20 Laravel框架



1. 课堂案例

课堂讲解

一、上节回顾

LINU使用

二、学习目标

学会PHP Laravel 框架的基本使用（路由，数据库操作）

三、教学过程 描述

1. 安装与目录结构

1-1 Laravel 简介

Laravel 是PHP的一个框架

首先了解什么是框架？

为解决一定问题并按照一定的设计模式搭建的项目架构，通俗的讲：框架就是一套开发工具，

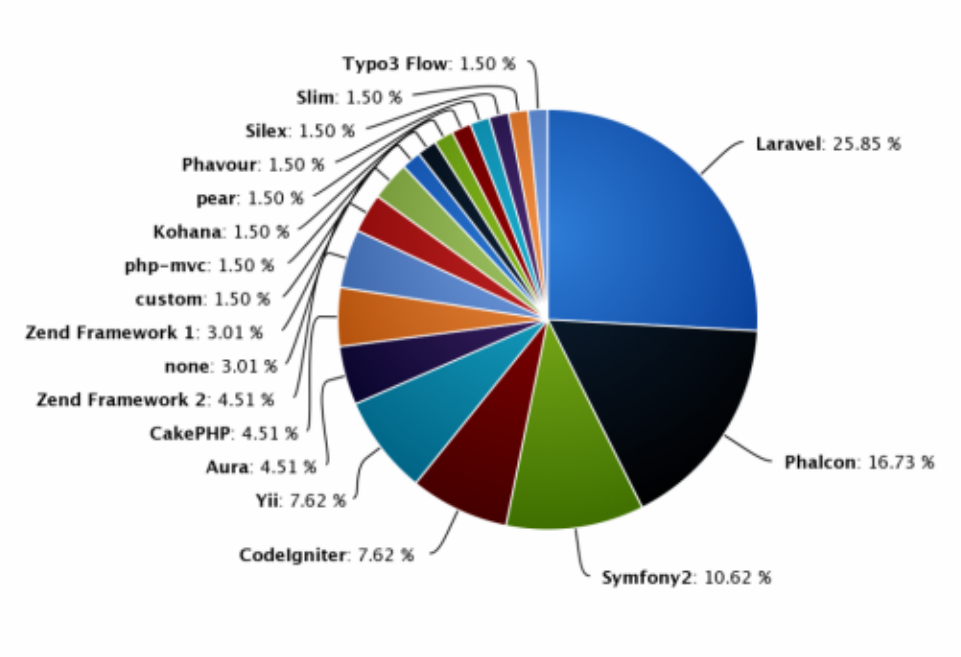
为什么使用框架？

框架提供了很多功能，比如数据库（DB），缓存(Cache)，

会话(Session)，文件上传，图片处理，分页，路由等....

，这就避免了开发者重复造轮子，将这此复杂的重复的工作交给大牛去编写！

PHP有很多的框架，我们应该如何去选择：下面是一个PHP框架排行版：



从上图可以看到，Laravel 是使用最多的框架。我们没有理由不去选择他

选择流行框架好处

1. 文档齐全

2. 社区活跃

3. 后期支持好

Laravel简介

1. Laravel 是一套简洁，优雅的PHP Web 开发框架

2. 具有富于表达性且简洁的语法

3. Laravel 是易于理解且强大的，它提供了强大的工具用以开发大型，健壮的应用

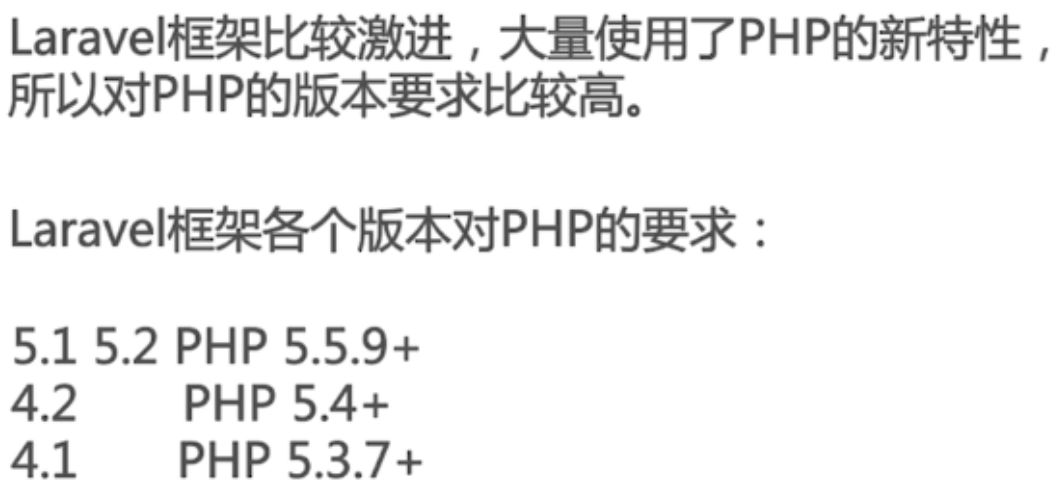
4. 具有验证、路由、session、缓存，数据库迁移工具、单元测试等常用的工具和功能

1-2 Laravel 版本选择

Laravel 版本较多

选择建议：

如果有服务器的完全控制权限，建议选择最新稳定版
否则，根据自己的服务器环境选择版本如：



我们的课程使用 Laravel 5.4 版本即最新版本

1-3 Laravel 开发环境搭建

Laravel 5.2　服务器要求:

PHP版本 >= 5.5.9
PHP扩展:OpenSSL
PHP扩展:PDO
PHP扩展:Mbstring
PHP扩展:Tokenizer

老师使用的是: WampServer Version 3.0.0 64bit　上面几个条件都满足

1-4 Laravel下载与安装

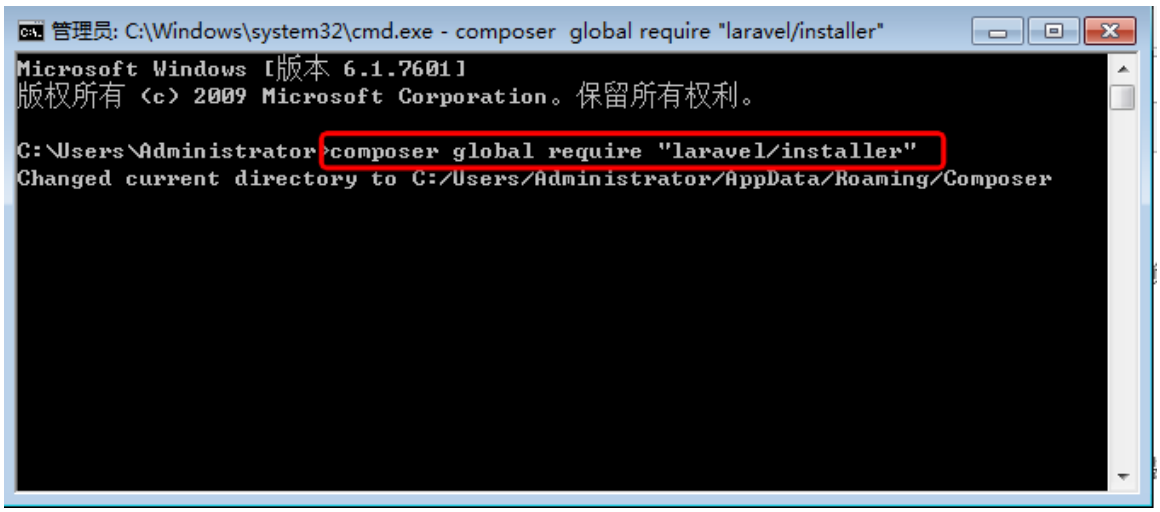
Laravel 官方推荐的安装方法是 使用 composer(PHP的包管理工具)来进行安装

composer
是一个PHP软件包的管理工具,大家暂时先别管具体的使用,只知道这是一个工具就好了,有时间老师会给大家讲解这个工具的使用方法,当前我们只讲解如何使用这个工具来安装 laravel 框架

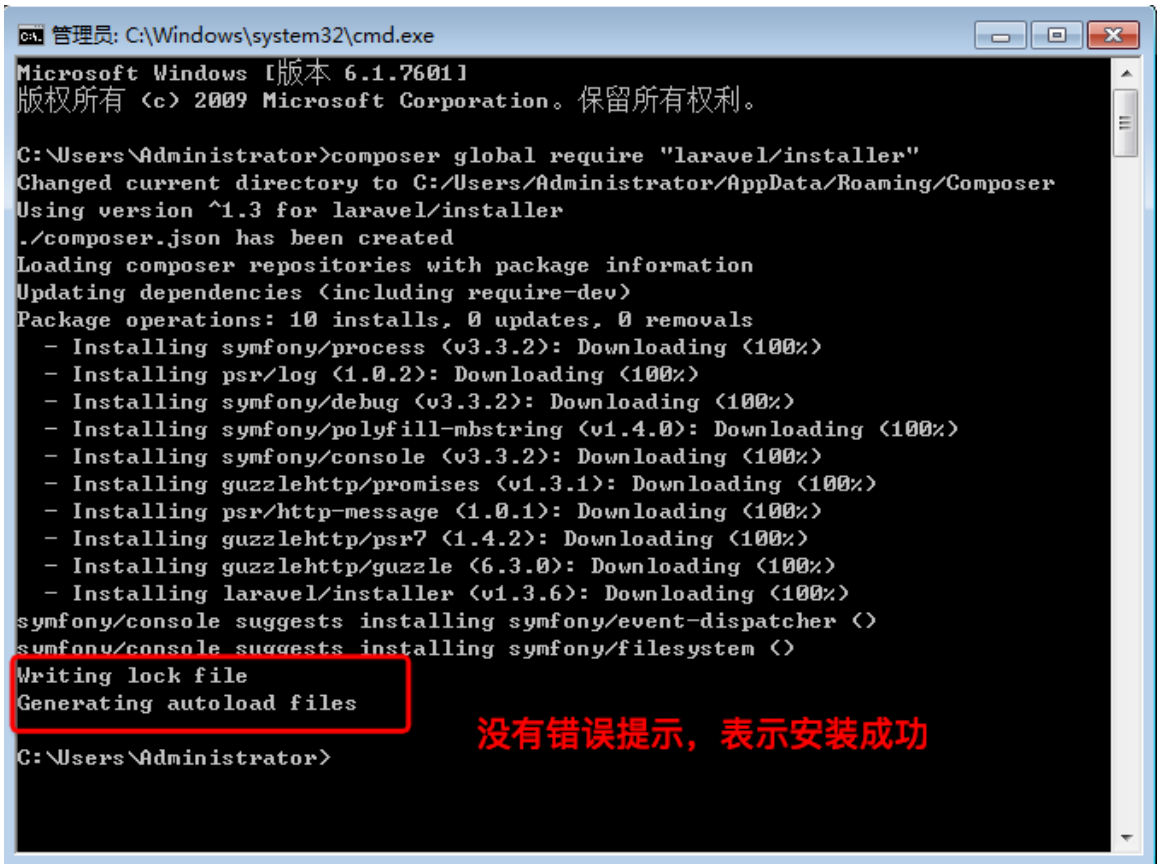
Laravel 安装步骤:

步骤1.　composer 工具的安装

步骤2.　使用 Composer 下载 Laravel 安装包: composer global require "laravel/installer"



这个步骤，根据网速快慢可能需要花个几分钟下载



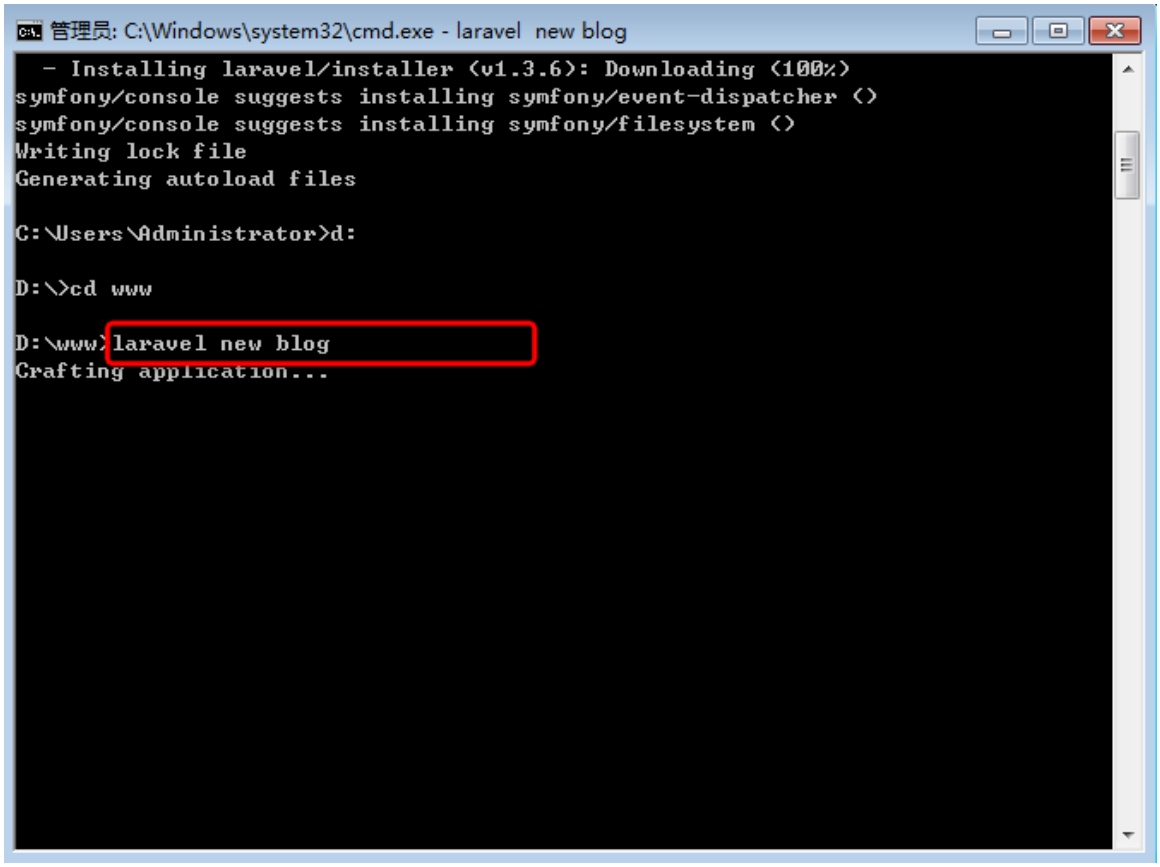
步骤3.　使用 laravel new 命令在指定目录创建一个新的 Laravel 项目：如

创建一个目录用于创建 Laravel 项目，比如：d:\www 目录做为项目的根目录

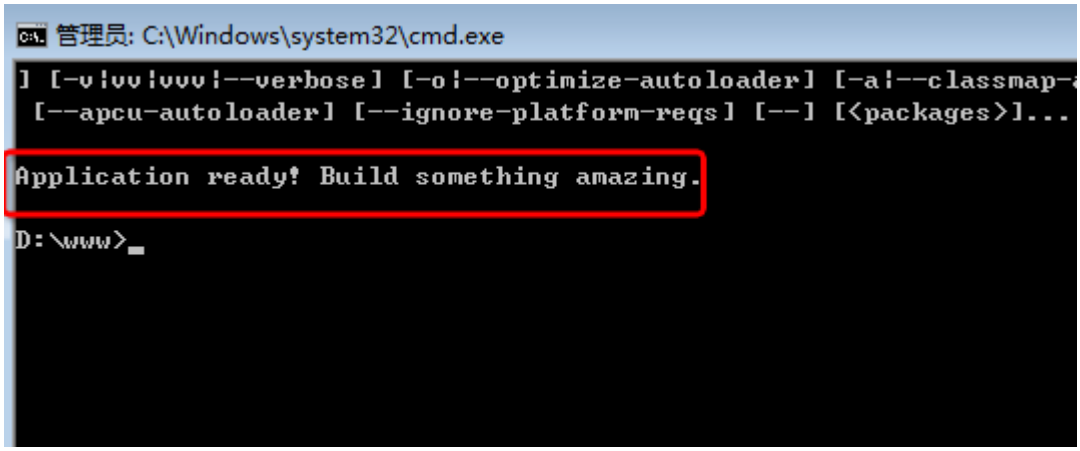
我们通过 命令行进入 此目录：

并执行 laravel new blog

blog 为我们要创建的项目名称，我们打算用laravel 那一个博客系统如下图：



根据网速不同，可能需要好几分钟



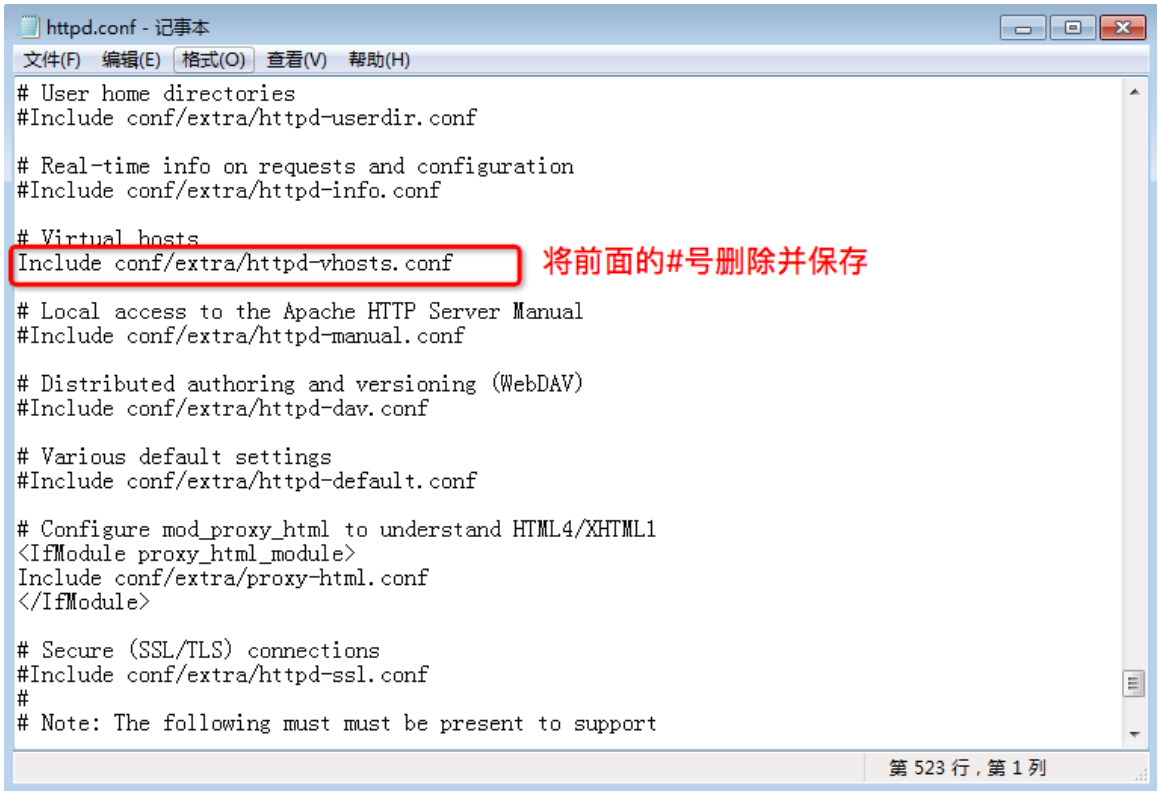
到此安装完成

步骤4. 指定 Web 服务器的网站根目录

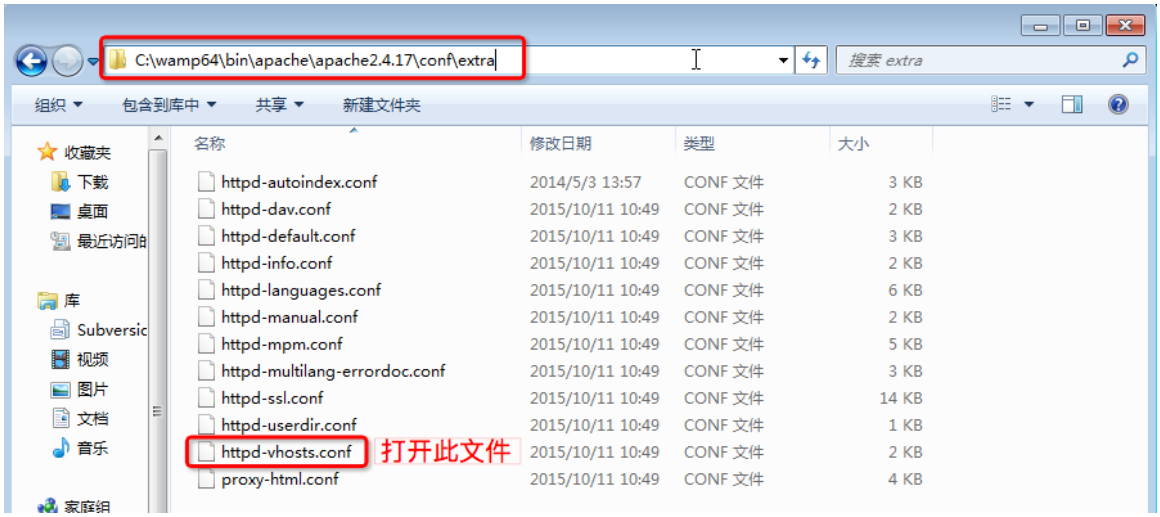
配置一个新的站点，将根目录指向：blog/public 目录，当前项目为：D:\www\blog\public

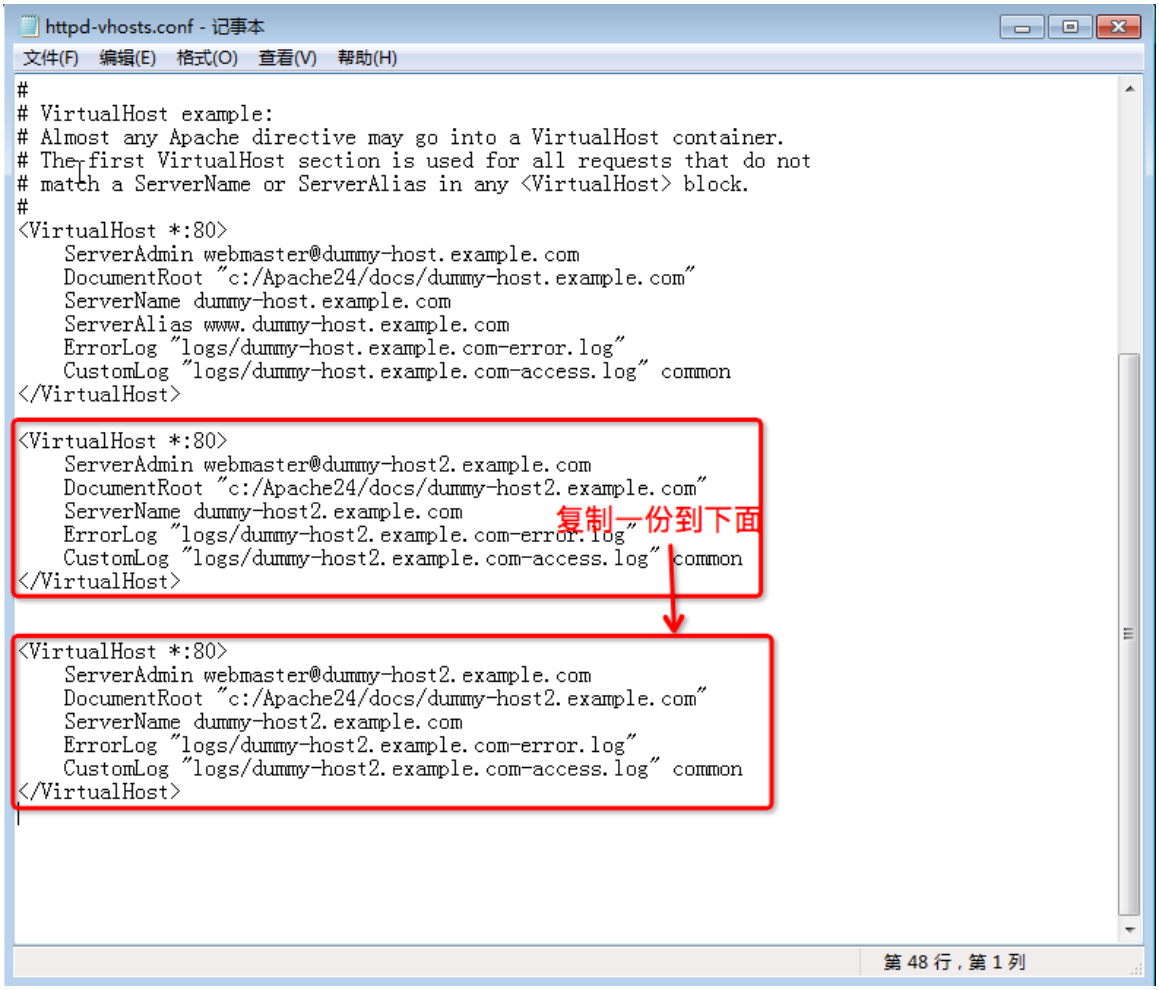
基于WAMP配置一个新站点，我们前面的课程我们已经讲解过，如果还不会也没关系，老师再次跟大家讲解一次：

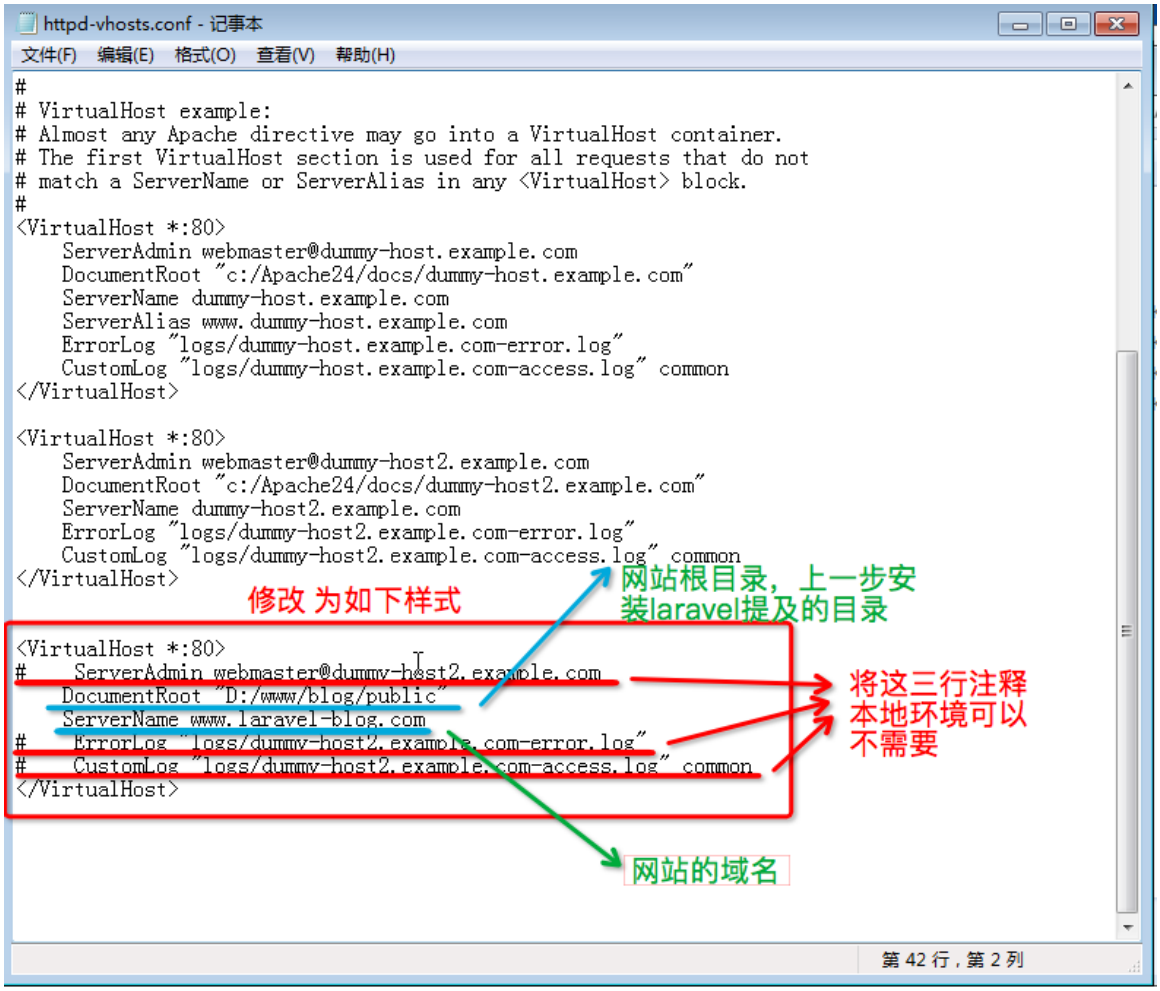
1> 首先点击WAMP图标打开httpd.conf 文件，搜索：httpd-vhosts.conf 如下：



2> 打开 wamp安装目录下面的的apache安装目录下面的 conf/extra/httpd-vhosts.conf 文件，即上图中红框的路径，老师的安装目录为下图：

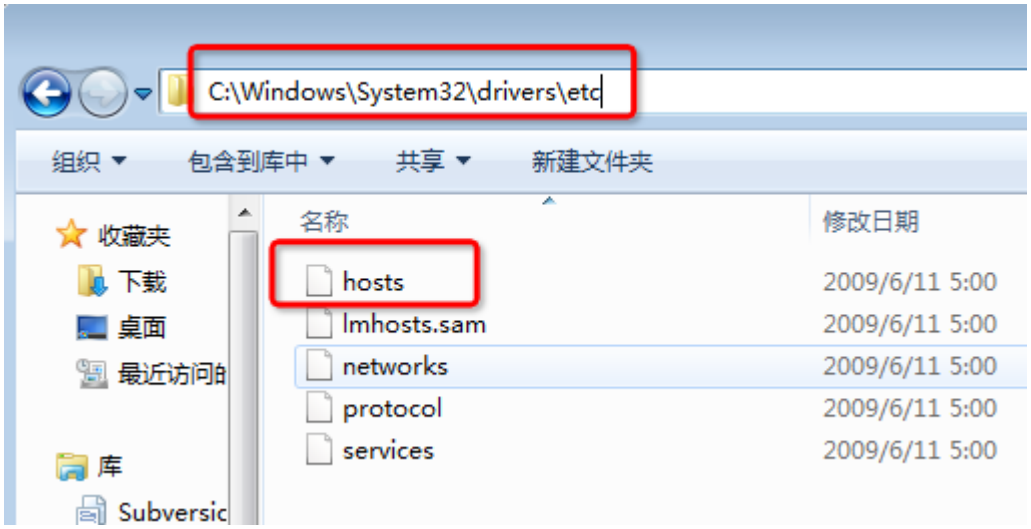


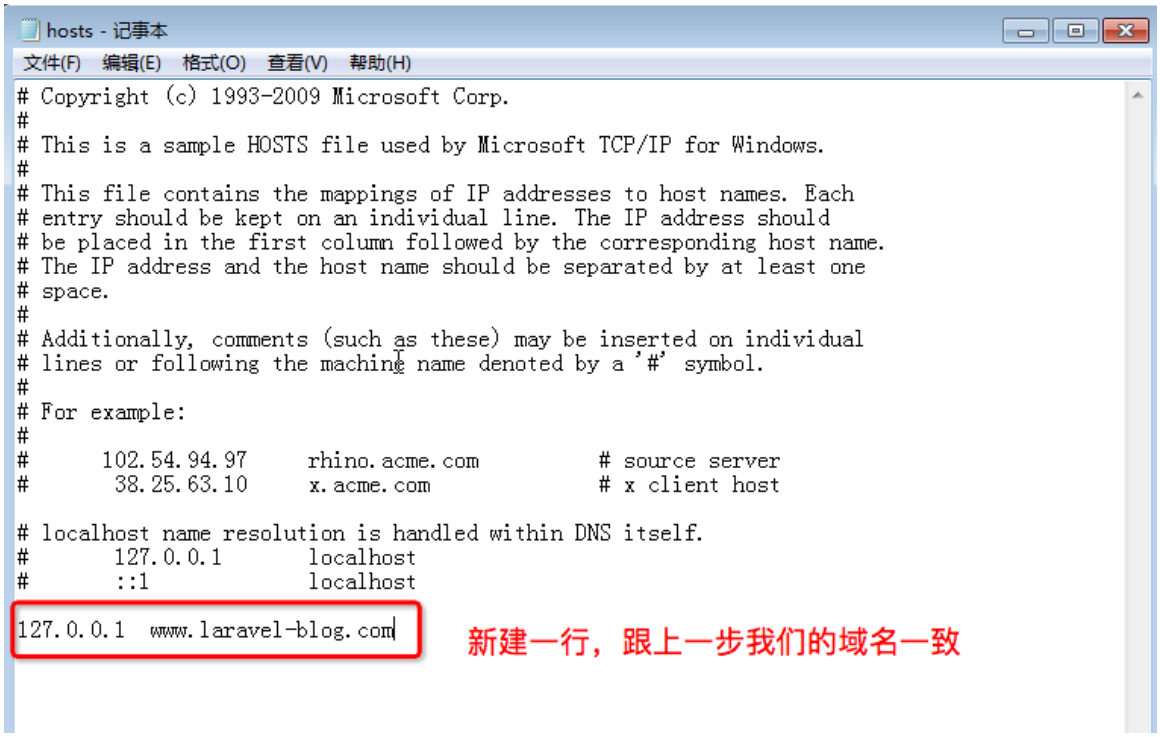

httpd-vhosts.conf - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
#
VirtualHost example:
Almost any Apache directive may go into a VirtualHost container.
The first VirtualHost section is used for all requests that do not
match a ServerName or ServerAlias in any <VirtualHost> block.
#
<VirtualHost *:80>
ServerAdmin webmaster@dummy-host.example.com
DocumentRoot "c:/Apache24/docs/dummy-host.example.com"
ServerName dummy-host.example.com
ServerAlias www.dummy-host.example.com
ErrorLog "logs/dummy-host.example.com-error.log"
CustomLog "logs/dummy-host.example.com-access.log" common
</VirtualHost>
<VirtualHost *:80>
ServerAdmin webmaster@dummy-host2.example.com
DocumentRoot "c:/Apache24/docs/dummy-host2.example.com"
ServerName dummy-host2.example.com
ErrorLog "logs/dummy-host2.example.com-error.log"
CustomLog "logs/dummy-host2.example.com-access.log" common
</VirtualHost>
复制一份到下面
<VirtualHost *:80>
ServerAdmin webmaster@dummy-host2.example.com
DocumentRoot "c:/Apache24/docs/dummy-host2.example.com"
ServerName dummy-host2.example.com
ErrorLog "logs/dummy-host2.example.com-error.log"
CustomLog "logs/dummy-host2.example.com-access.log" common
</VirtualHost>
第 48 行，第 1 列



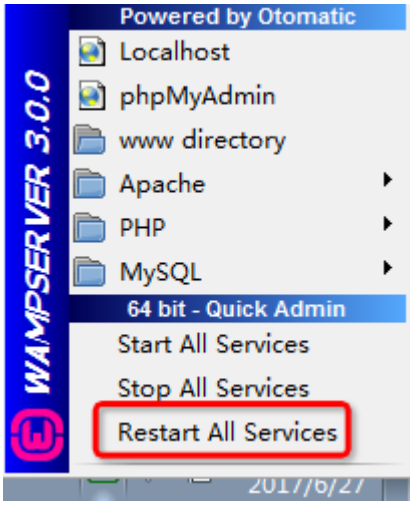
3>修改host 文件，以便绑定上图中的的域名　www.laravel-blog.com

打开：





4> 重启WAMP

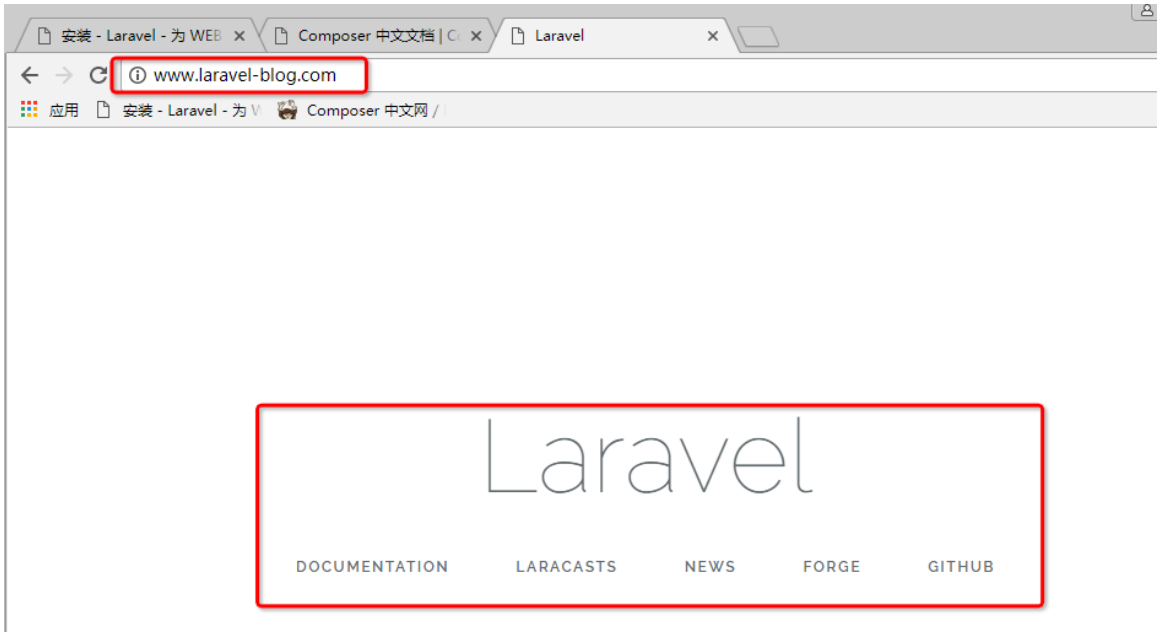


5> 如果未开启rewrite，则开启rewrite

步骤5. 目录权限 (window环境跳过此步骤)

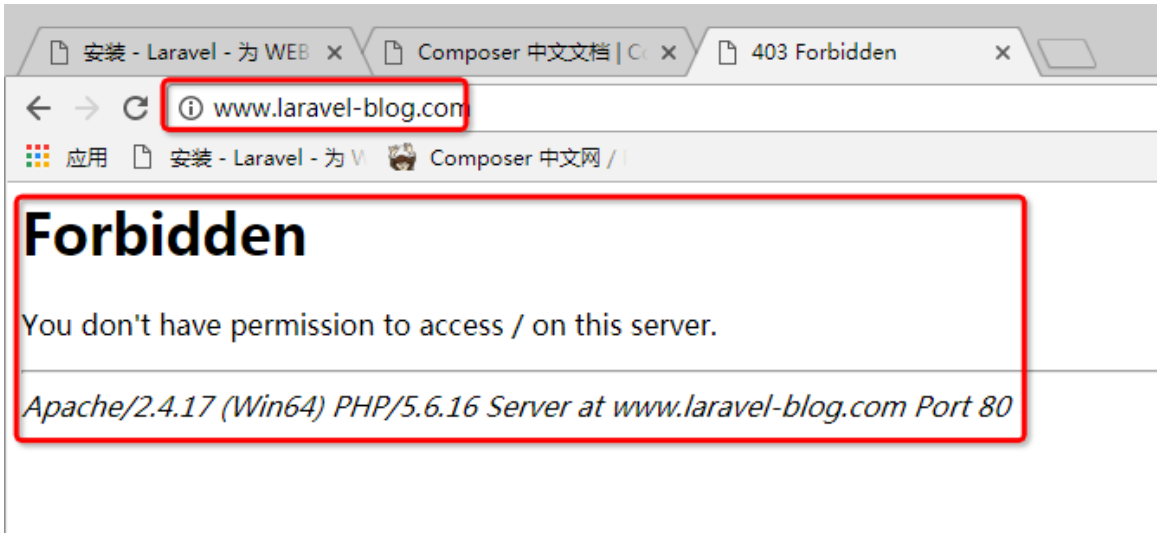
安装 Laravel 之后，你必须设置一些文件目录权限。如果是Linux 系统，如下两个目录storage 和 bootstrap/cache 目录必须让服务器有写入权限。

步骤6. 打开我们新配置的网站：http://www.laravel-blog.com

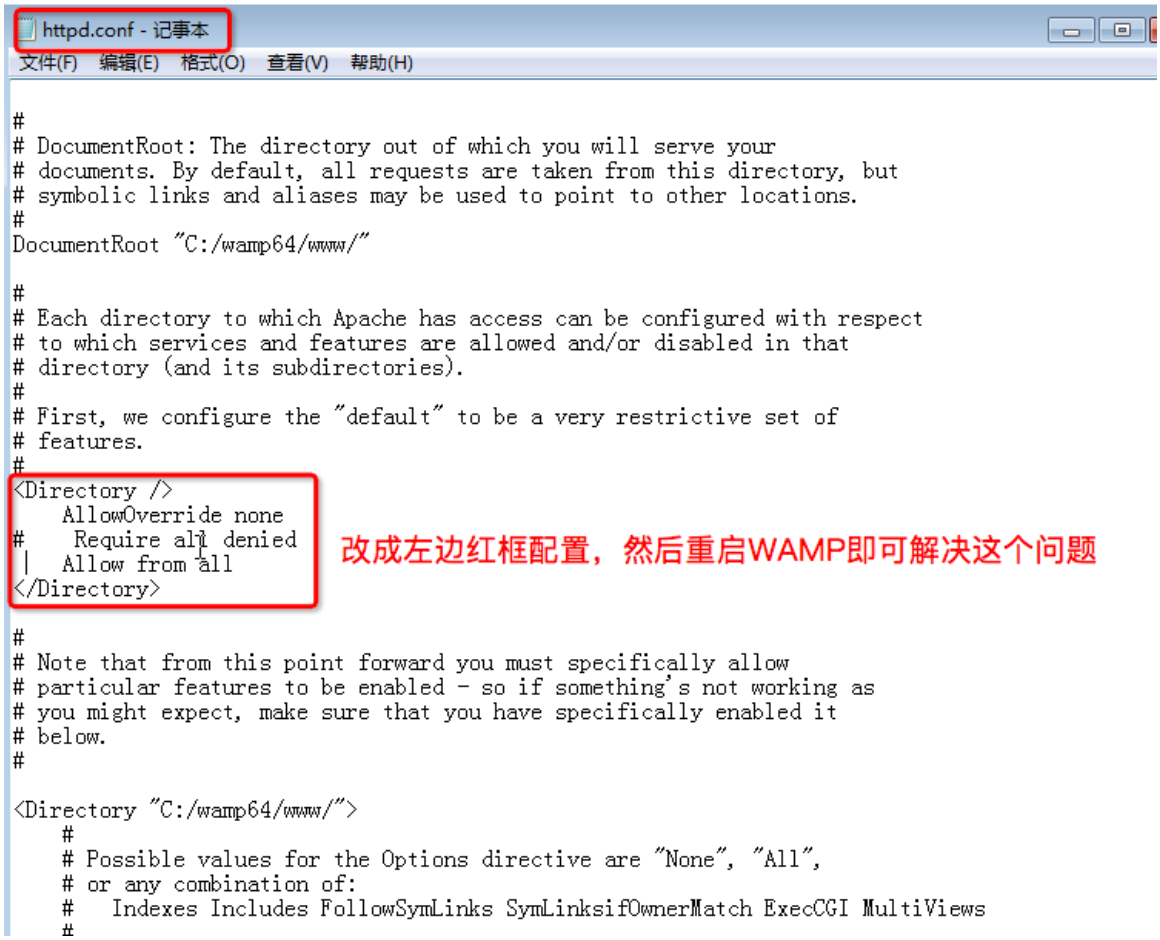


如上图，说明我们的Laravel 安装成功，

如果发现打开网站提示：

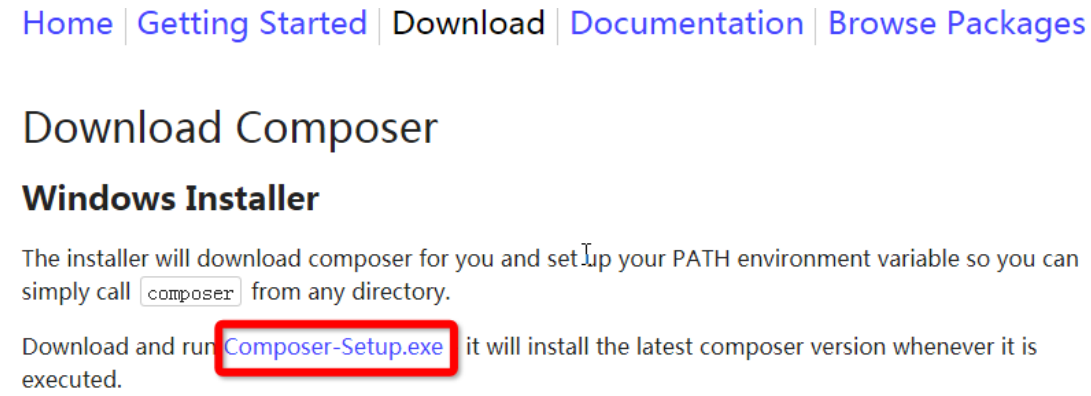


说明我们的配置权限有总是，可以再次打开httpd.conf 配置文件修改，如下图：



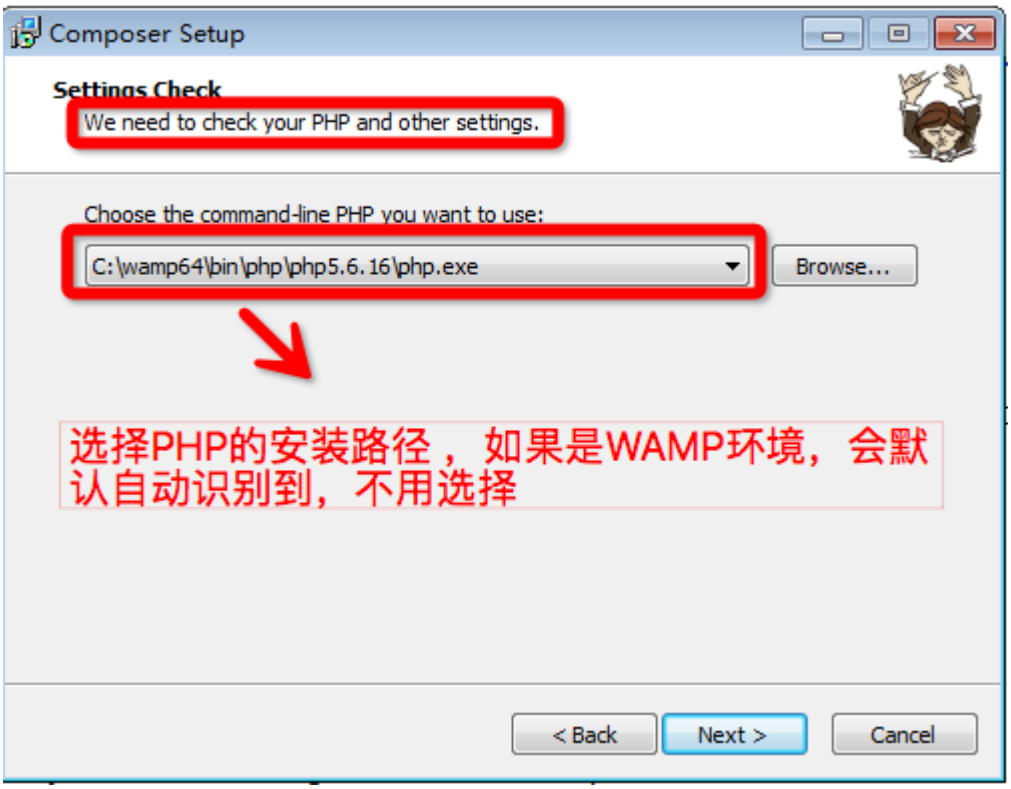
1-4-1 composer 安装

window 版本下载路径：https://getcomposer.org/download/



下载完后，直接运行：一路next 就OK

不过期间有一个步骤会要求选择PHP的安装路径，如下图：



按提示很快就能完成安装

1-4-2 安装 Laravel

安装步骤:

1. 使用 Composer 下载 Laravel 安装包:

   composer global require "laravel/installer"

此步骤需要从网络下载代码，可能需要花费较长时间，我的环境大概6分钟左右，根据网速而定。

2. 加环境变量

一般经过第一步，这个配置是会自动加到环境变量里，但可能跟系统有关，有些系统下，经过第一步的操作后，并没有加入到系统变量，自行检查下系统变量，如果没有，请将这个路径加到系统变量PATH下: ~/ 表示的是用户目录

   ~/.composer/vendor/bin

3. 进入到想要创建项目的目录:执行如下命令:

   laravel new blog

   blog 为项目名称、 此步骤需要下载laravel 框架源代码，laravel 框架5.4版本有50几M，下载时间跟网速有关，我的环境:因为服务器是国外，我花了8分钟左右

4. 创建一个虚拟主机: 主目录指向:

   public 目录

5. 验证:如果访问public目录出现如下页面，表示安装成功:

1-5 Laravel核心目录文件介绍

#根目录

- app 目录
- bootstrap 目录
- config 目录
- database 目录
- public 目录
- resources 目录
- routes 目录
- storage 目录
- tests 目录
- vendor 目录

#App 目录

- Console 目录
- Events 目录
- Exceptions 目录
- Http 目录
- Jobs 目录
- Listeners 目录
- Mail 目录
- Notifications 目录
- Policies 目录
- Providers 目录

简介

Laravel
默认的目录结构意在为构建不同大小的应用提供一个好的起点，当然，你可以自己按照喜好组织应用目录结构，Laravel 对类在何处被加载没有任何限制 -- 只要 Composer 可以自动载入它们即可。

为什么没有 Models 目录?

许多初学者都会困惑 Laravel 为什么没有 models 目录,当然，这是 laravel 故意为之，因为 models 这个词对不同开发者而言有不同的含义，容易造成歧义，有些开发者认为应用的模型指的是业务逻辑，还有些开发者则认为模型指的是与关联数据库的交互。

正是因为如此，我们默认将 Eloquent 的模型放置到 app 目录下，从而允许开发者自行选择放置的位置。

根目录

app 目录

app
目录，如你所料，这里面包含应用程序的核心代码。另外，你为应用编写的代码绝大多数也会放到这里，我们之后将很快对这个目录的细节进行深入探讨。

bootstrap 目录

bootstrap目录包含了几个框架启动和自动加载设置的文件。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。

config 目录

config
目录，顾名思义，包含所有应用程序的配置文件。通读这些配置文件可以应对自己对配置修改的需求。

database 目录

database 目录包含了数据迁移及填充文件，你还可以将其作为 SQLite 数据库的存放目录。

public 目录

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

resources 目录

resources 目录包含了视图、原始的资源文件 (LESS、SASS、CoffeeScript)，以及语言包。

routes 目录

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。

web.php 文件里定义的路由都会在 RouteServiceProvider 中被指定应用到 web 中间件组，具备 Session、CSRF 防护以及 Cookie 加密功能，如果应用无需提供无状态的、RESTful 风格的API，所有路由都会定义在 web.php 文件。

api.php 文件里定义的路由都会在 RouteServiceProvider 中被指定应用到 api 中间件组，具备频率限制功能，这些路由是无状态的，所以请求通过这些路由进入应用需要通过 API 令牌进行认证并且不能访问 Session 状态。

console.php
文件用于定义所有基于闭包的控制台命令，每个闭包都被绑定到一个控制台命令并且允许与命令行 IO 方法进行交互，尽管这个文件并不定义 HTTP 路由，但是它定义了基于命令行的应用入口（路由）。

storage 目录

storage 目录包含编译后的 Blade 模板、基于文件的 session、文件缓存和其它框架生成的文件。此文件夹分格成 app 、framework ，及 logs 目录。app 目录可用于存储应用程序使用的任何文件。framework 目录被用于保存框架生成的文件及缓存。最后，logs 目录包含了应用程序的日志文件。

storage/app/public 可以用来存储用户生成的文件，例如头像文件，这是一个公开的目录。你还需要在 public/storage 目录下生成一个软连接指向这个目录，你可以使用 php artisan storage:link 来创建软链接。

tests 目录

tests 目录包含自动化测试。Laravel 推荐了一个 PHPUnit 例子。每一个测试类都需要添加 Test 前缀，你可以使用 phpunit 或者 php vendor/bin/phpunit 命令来运行测试。

vendor 目录

vendor 目录包含所有 Composer 依赖。

app 目录

应用的核心代码位于 app 目录下，默认情况下，该目录位于命名空间 App 下， 并且被 Composer 通过 PSR-4 自动载入标准 自动加载。

app 目录下包含多个子目录，如 Console 、Http 、Providers 等。 其中 Console 和 Http 目录为进入应用程序核心提供了一个 API 。HTTP 协议和 CLI 是和应用进行交互的两种机制，但实际上并不包含应用逻辑。换句话说，它们是两种简单地发布命令给应用程序的方法。Console 目录包含你全部的 Artisan 命令，而 Http 目录包含你的控制器、中间件和请求。

其他目录将会在你通过 Artisan 命令 make 生成相应类的时候生成到 app 目录下。例如，app/Jobs 目录在你执行 make:job 命令生成任务类时，才会出现在 app 目录下。

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

Console 目录

Console 目录包含应用所有自定义的 Artisan 命令，这些命令类可以使用 make:command 命令生成。该目录下还有 Console Kernel 类，在这里可以注册自定义的 Artisan 命令以及定义调度任务。

Events 目录

Events 目录默认不存在，它会在你使用 event:generate 或者 event:make 命令以后才会生成。如你所料，此目录是用来放置 事件类的。事件类用于当指定事件发生时，通知应用程序的其它部分，并提供了很棒的灵活性及解耦。

Exceptions 目录

Exceptions
目录包含应用的异常处理，同时还是处理应用抛出的任何异常的好位置。如果你想自定义异常的记录和渲染，你应该修改此目录下的 Handler 类。

Http 目录

Http 目录包含了控制器、中间件以及表单请求等，几乎所有进入应用的请求处理都在这里进行。

Jobs 目录

Jobs 目录默认不存在，可以通过执行 make:job 命令生成，Jobs 目录用于存放队列任务，应用中的任务可以推送到队列，也可以在当前请求生命周期内同步执行。同步执行的任务有时也被看作命令，因为它们实现了 命令总线设计模式。

Listeners 目录

Listeners 目录默认不存在，可以通过执行 event:generate 和 make:listener 命令创建。Listeners 目录包含处理 事件 的类（事件监听器），事件监听器接收一个事件并提供对该事件发生后的响应逻辑，例如，UserRegistered 事件可以被 SendWelcomeEmail 监听器处理。

Mail 目录

Mail 目录默认不存在，但是可以通过执行 make:mail 命令生成，Mail 目录包含邮件发送类，邮件对象允许你在一个地方封装构建邮件所需的所有业务逻辑，然后使用 Mail::send 方法发送邮件。

Notifications 目录

Notifications 目录默认不存在，你可以通过执行 make:notification 命令创建， Notifications 目录包含应用发送的所有通知，比如事件发生通知。Laravel 的通知功能将通知发送和通知驱动解耦，你可以通过邮件，也可以通过 Slack、短信或者数据库发送通知。

Policies 目录

Policies 你可以通过执行 ·make:policy· 命令来创建， ·Policies· 目录包含了所有的授权策略类，策略用于判断某个用户是否有权限去访问指定资源。更多详情，请查看 授权文档。

Providers 目录

Providers 目录包含应用的服务提供者。服务提供者在启动应用过程中绑定服务到容器、注册事件，以及执行其他任务，为即将到来的请求处理做准备。

在新安装的 Laravel 应用中，该目录已经包含了一些服务提供者，你可以按需添加自己的服务提供者到该目录。

2. Laravel 执行流程(即生命周期)

生命周期概述

1. 入口文件

一个 Laravel 应用的所有请求的入口都是 public/index.php 文件。通过网页服务器 (Apache / Nginx) 所有请求都会导向这个文件。index.php 文件没有太多的代码，只是加载框架其他部分的一个入口。

index.php 文件载入 Composer 生成的自动加载器定义，并从 bootstrap/app.php 文件获取到 Laravel 应用实例。Laravel 的第一个动作就是创建一个自身应用实例 / 服务容器。

2. 加载HTTP / Console 内核

接下来，传入的请求会被发送给 HTTP 内核或者 console 内核，这根据进入应用的请求的类型而定。这两个内核服务是所有请求都经过的中枢。让我们现在只关注位于 app/Http/Kernel.php 的 HTTP 内核。

HTTP 内核继承自 Illuminate\Foundation\Http\Kernel 类，它定义了一个 bootstrappers 数组，数组中的类在请求真正执行前进行前置执行。
这些引导程序配置了错误处理，日志记录，检测应用程序环境，以及其他在请求被处理前需要完成的工作。

HTTP 内核同时定义了一个 HTTP 中间件列表，所有的请求必须在处理前通过这些中间件，这些中间件处理 HTTP session 的读写，判断应用是否在维护模式，验证 CSRF token 等等。

HTTP 内核的标志性 handle 方法是相当简单的：接收一个 Request 并返回一个 Response。你可以把内核想成一个代表你应用的大黑盒子。给它喂 HTTP 请求然后它就会吐给你 HTTP 响应。

3. 注册服务提供者

在内核引导启动的过程中最重要的动作之一就是载入 服务提供者到你的应用。所有的服务提供者都配置在 config/app.php 文件中的 providers 数组中。首先，所有提供者的 register 方法会被调用，接下来，一旦所有提供者注册完成，boot 方法将会被调用。

服务提供者负责引导启动框架的全部各种组件，例如数据库、队列、验证器以及路由组件。因为这些组件引导和配置了框架的各种功能，所以服务提供者是整个 Laravel 启动过程中最为重要的部分。

4. 分发请求

一旦应用完成引导和所有服务提供者都注册完成，Request 将会移交给路由进行分发。路由将分发请求给一个路由或控制器，同时运行路由指定的中间件。

整个服务都是基于服务提供者

服务提供者是 Laravel 应用的真正关键部分，应用实例被创建后，服务提供者就会被注册完成，并将请求传递给应用进行处理，真的就是这么简单！

了解 Laravel 是怎样通过服务提供者构建和引导一个稳定的应用是非常有价值的，当然，应用的默认服务提供者都存放在 app/Providers 目录中。

在新创建的应用中，AppServiceProvider 文件中方法实现都是空的。这个提供者是你添加应用专属的引导和服务的最佳位置，当然的，对于大型应用你可能希望创建几个服务提供者，每个都具有粒度更精细的引导。

3. Laravel 中的路由和MVC

Laravel 中有很多的概念，比如：依赖注入，服务提供者，绑定，窗口事件，中间件等等，但老师认为，做为初学者，可以先不理会这些概念，因为这些涉及到一些软件开发的高级思想，对于刚入门者来说，还很难理解，老师可不希望你们直接从入门到放弃。我们前期只要会简单的使用，会使用Laravel 做简单的项目，那么随着对Laravel 的深入，慢慢就会理解这些概念。

接下来的课程，老师也会绕过这些知识，直接进入使用！

3-1 开发流程

我们使用Laravel 开发的基本流程为：（首先当然是安装）

1. 定义路由，(Laravel 的开发都是从路由开始的)

2. 定义控制器

3. 创建视图（即模板）

3-2 路由

在讲解目录结构时，我们分析了路由，正常情况下都是在 routes/web.php 文件中定义，其它的路由今后用到我们再细讲

需要注意的是： 由于web.php 路由会应用中间件，会有一些安全检测，诸如 Session 和 CSRF 保护等特性，用到时我再讲解

大多数的应用构建，都是以在 routes/web.php 文件定义路由开始的。

可用的路由方法

我们可以注册路由来响应所有的 HTTP 操作：

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

多请求路由

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

```
Route::any('foo', function () {
});
```

CSRF 保护（对于表单，一般指post表单，需要在html中加一个CSRF令牌）原因如下：

任何指向 web 中 POST，PUT 或 DELETE 路由的 HTML 表单请求都应该包含一个 CSRF 令牌，否则，这个请求将会被拒绝。更多的关于 CSRF 的说明在 CSRF 说明文档：

```
<form method="POST" action="/profile">
    {{ csrf_field() }}
    ...
</form>
```

关于 CSRF 老师在接下来的 网站安全课程中会细讲

路由参数（分为必选路由参数，可选路由参数），其实就是我们常说的url 参数

必选路由参数

当然，有时我们需要在路由中捕获一些 URL 片段。

例如，我们需要从 URL 中捕获用户的 ID，我们可以这样定义路由参数：

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

也可以根据需要在路由中定义多个参数：

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

路由的参数通常都会被放在 {} 内，并且参数名只能为字母，当运行路由时，参数会通过路由闭包来传递。

注意：路由参数不能包含 - 字符。请用下划线 (_) 替换。

可选路由参数

声明路由参数时，如需指定该参数为可选，可以在参数后面加上 ? 来实现，但是相应的变量必须有默认值：

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

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

正则表达式约束（针对的是上面的参数做的约束）

你可以使用 where 方法来规范你的路由参数格式。where 方法接受参数名称和定义参数约束规则的正则表达式：

```
Route::get('user/{name}', function ($name) {
    //
})->where('name', '[A-Za-z]+');

Route::get('user/{id}', function ($id) {
    //
})->where('id', '[0-9]+');

Route::get('user/{id}/{name}', function ($id, $name) {
    //
})->where(['id' => '[0-9]+', 'name' => '[a-z]+']);
```

命名路由（就是给路由弄个别名，以便今后要使用route(别名)函数去生成 URL ）类似TP中的U()方法

为路由指定了名称后，我们可以使用全局辅助函数 route 来生成 URL 或者重定向到该条路由：

```
// 生成 URL...
$url = route('profile');

// 生成重定向...
return redirect()->route('profile');
```

如果是有定义参数的命名路由，可以把参数作为 route 函数的第二个参数传入，指定的参数将会自动插入到 URL 中对应的位置：

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

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

路由组（就是将一些路由的共有属性写到一起，这样不用为每一条路由定义）

支持的属性为：

1. 中间件

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

    Route::get('user/profile', function () {
        // 使用 `Auth` 中间件
    });
});
```

2.命名空间

另一个常见的例子是，为控制器组指定公共的 PHP 命名空间。这时使用 namespace 参数来指定组内所有控制器的公共命名空间：

```
Route::group(['namespace' => 'Admin'], function () {
    // 在 "App\Http\Controllers\Admin" 命名空间下的控制器
});
```

请记住，默认 RouteServiceProvider 会在命名空间组中引入你的路由文件，让你不用指定完整的 App\Http\Controllers

命名空间前缀就能注册控制器路由，因此，我们在定义的时候只需要指定命名空间 App\Http\Controllers 以后的部分。

3.子域名路由

路由组也可以用作子域名的通配符，子域名可以像 URI 一样当作路由组的参数，因此允许把捕获的子域名一部分用于我们的路由或控制器。可以使用路由组属性的 domain 键声明子域名。

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

4. 路由前缀

通过路由组数组属性中的 prefix 键可以给每个路由组中的路由加上指定的 URI 前缀，例如，我们可以给路由组中所有的 URI 加上路由前缀 admin :

```
Route::group(['prefix' => 'admin'], function () {
    Route::get('users', function ()    {
        // 匹配包含 "/admin/users" 的 URL
    });
});
```

3-3 控制器

3-3-1 控制器定义

在讲解目录结构时，我们有分析，控制器是定义在 app/Http/Controllers 目录下面，大家记住就行

1. 控制器 -- 定义

```
<?php

namespace App\Http\Controllers;



class UserController extends Controller
{

   public function show($id)
   {
      return "显示id".$id;
   }
}
```

定义一个路由指向该控制器:如

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

如果，控制器定义在 app/Http/Controllers/logic/UserController.php

那么路由规则为：

Route::get('userlogic/{id}','Logic\UserController@show');

3-3-2 调用视图

在控制器中，可以直接使用 view('名称')

如：

```
//调用模板
public function show($id){
    return view('show',['id'=>$id]);  //对应的模板文件为：resources/views/show.blade.php
}
```

3-4 视图

创建视图

视图的用途是用来存放应用程序中 HTML 内容，并且能够将你的控制器层（或应用逻辑层）与展现层分开。视图文件目录为 resources/views ，示例视图如下：

```
<!-- 此视图文件位置：resources/views/greeting.blade.php -->

<html>
  <body>
    <h1>Hello, {{ $name }}</h1>
  </body>
</html>
```

上述视图文件位置为 resources/views/greeting.blade.php ，我们可以通过全局函数 view 来使用这个视图，如下：

```
Route::get('/', function () {
  return view('greeting', ['name' => 'James']);
```

```
});
```

如你所见，view 函数中，第一个参数是 resources/views 目录中视图文件的文件名，第二个参数是一个数组，数组中的数据可以直接在视图文件中使用。在上面示例中，我们将 name 变量传递到了视图中，并在视图中使用 Blade 模板语言 打印出来。

当然，视图文件也可能存放在 resources/views 的子目录中，你可以使用英文句点 . 来引用深层子目录中的视图文件。例如，一个视图的位置为 resources/views/admin/profile.blade.php，使用示例如下：

```
return view('admin.profile', $data);
```

判断视图文件是否存在

如果需要判断一个视图文件是否存在，你可以使用 View Facade 上的 exists 方法来判定，如果视图文件存在，则返回值为 true：

```
use Illuminate\Support\Facades\View;

if (View::exists('emails.customer')) {
    //
}
```

传递数据到视图

如上述例子中，你可以使用数组将数据传递到视图文件：

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

当使用上面方式传递数据时，第二个参数（$data）必须是键值对数组（关联数组）。在视图文件中，你可以通过对应的关键字（$key）取用相应的数据值，例如 <?php echo $key; ?>。如果只需要传递特定数据而非一个臃肿的数组到视图文件，可以使用 with 辅助函数，示例如下：

```
return view('greeting')->with('name', 'Victoria');
```

把数据共享给所有视图

有时候可能需要共享特定的数据给应用程序中所有的视图，那这时候你需要 View Facade 的 share 方法。通常需要将所有 share 方法的调用代码放到 服务提供者 的 boot 方法中，此时你可以选择使用 AppServiceProvider 或创建独立的 服务提供者 。示例代码如下：

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        View::share('key', 'value');
    }

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

3-5 模型

在app hhvo

4. 基本增删改查操作

4-1 数据准备/数据库连接

首先创建一个数据库：laravel_blog, 同时创建一张表，用于存储测试数据，我们创建一个博客表 blog

```
CREATE TABLE `laravel_blog`.`<table_name>` (
    `id` int NOT NULL AUTO_INCREMENT,
    `title` varchar(250),
    `small_title` varchar(200),
    `content` text,
    `click` varchar(255),
    `create_time` int(11),
    `status` tinyint(1),
    PRIMARY KEY (`id`)
) ENGINE=`InnoDB` AUTO_INCREMENT=1000 DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci COMMENT='博客内容表';
```

连接数据库(配置)

打开 config/database.php

修改配置

修改 connections 数组对应的 mysql 项,同时修改 .env 文件中的配置如下:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel_blog
DB_USERNAME=root
DB_PASSWORD=
```

4-2 DB facade

首先在控制器引入:use Illuminate\Support\Facades\DB;

1. 查询数据

DB::select($sql,[]);

```php
//查询数据
public function blogSelect(){
    $result = DB::select("select * from blog where id>=?",[1000]);
    dd($result);  //dd() 方法，类似 tp里的dump方法
}
```

2. 添加数据

DB::insert($sql,[]);

```php
//添加数据
public function blogInsert(){
    for($i=1;$i<100;$i++){
        $result = DB::insert("insert blog (title,small_title,content,click,create_time,status)
            values(?,?,?,?,?,?)",[
            "测试标题".$i,"测试小标题".$i,"测试内容".$i,10,time(),1
            ]);
    }
    if($result){
        echo "成功";
    }
}
```

3. 修改数据

DB::update($sql,[]);

```php
//修改数据
public function blogUpdate(){
    $result = DB::update("update blog set title=? where id=?",[
        '测试标题2',
        1001
        ]);
    var_dump($result);
}
```

4. 删除数据

DB::delete($sql,[]);

```php
//删除数据
public function blogDelete($id){
    $result = DB::update("delete from blog where id=?",[
        $id
        ]);
    var_dump($result);
}
```

4-3 查询构造器

使用构造器的方式实现CURD, 类似TP里的连贯操作

1. 插入数据

```
$result = DB::table('blog')->insert([
        'title'=>'构造器方式插入标题',
        'small_title'=>'构造器方式小标题',
        'content'=>'构造器方式内容',
        'click'=>1090,
        'status'=>1
        ]);


$result = DB::table('blog')->insertGetId([
        'title'=>'构造器方式插入标题',
        'small_title'=>'构造器方式小标题',
        'content'=>'构造器方式内容',
        'click'=>1090,
        'status'=>1
        ]);
```

2. 更新数据

```
$nu = DB::table('blog')
          ->where('id',1105)
          ->update([
            'title'=>'构造器方式修改标题',
            'small_title'=>'构造器方式小标题22',
            ]);
      dd($nu);
```

自增:

```
$nu = DB::table('blog')
          ->where('id','1105')
          ->increment('click',3)
```

自减:

```
$nu = DB::table('blog')
          ->where('id','1105')
```

```
            ->decrement('click',3);
        dump($nu);
```

3. 删除数据

```
 $nu = DB::table('blog')
            ->where('id',1105) //where('id','>=',1005)
            ->delete();
```

4. 查询数据

```
$result = DB::table('blog')
            ->orderBy('id','desc')  //排序
            ->where('id','>',20) //筛选
            //->select('title','click','status')  //要查询的字段，类似 tp field()
            ->distinct()    //去重
            //->whereRaw('id > ? and click > ?',[20,10]) //多条件查询
            //->pluck('title','click');    //返回指定字段类似tp 的getField()
            ->chunk('2',function($rs){
                echo "<pre>";
                var_dump($rs);
                echo "</pre>";
                return false;
            }); //分段查询
            //->get(); //查询所有数据
            //->first(); //
        dd($result);
```

5. 聚合

count、max、min、avg 和 sum

6. offset、limit 、join、

7. whereIn、whereNotIn、whereBetween、whereNotBetween、whereNull、whereNotNull

8. where、orWhere

4-4 Eloquent ORM

使用步骤:

1. 新建一个模型 如: app\blogModel.php 如:

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

class Flight extends Model
{
    //
}
```

几个重要属性:

protected $table 表名

public $incrementing 是否有自增ID字段, 没有设置为flase

public $timestamps = false; 是否自动维护 created_at 和 updated_at 字段

protected $connection   数据库连接

2. 控制器 引用 并查询数据

```
use App\BlogModel;
$blog = App\BlogModel::all();

foreach ($blog as $item) {
   echo $item->name;
}
```

也可当做查询构造器使用如:

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

3. 添加数据

```
$blog = new Blog;
$blog->title = "veiol";
$blog->save();
```

4. 修改数据

```
$blog = Blog::find(1);
$blog->title = '这是新标题';
$blog->save();
```

5. 删除数据

```
$blog = Blog::find(1);
$blog->delete();
```

5. Blade 模板语法

5-1 模板布局

5-1-1 模板继承

Blade 的模板继承跟 TP里的类似

步骤一、定义页面布局

首先定义一个布局文件：resources/views/layouts/app.blade.php

```
<html>
  <head>
    <title>应用程序名称 - @yield('title')</title>
  </head>
  <body>
    @section('sidebar')
      这是 master 的侧边栏。
    @show

    <div class="container">
      @yield('content')
    </div>
  </body>
</html>
```

上面的布局中使用了两个关键命令：(@section @show)这是共同使用 和 @yield 两个命令，效果都用子模板的内容来替换这里的定义，但@section 允许扩展，即子模板可以保留布局的内容的同时，对布局进行扩展，而@yield 是直接用子模板中 section 的内容替换主模板的内容

步骤二、继承页面布局

新建一个模板文件：resources/views/child.blade.php

```
@extends('layouts.app')

@section('title', 'Page Title')

@section('sidebar')
  @parent
```

```
    <p>This is appended to the master sidebar.</p>
@endsection

@section('content')
    <p>This is my body content.</p>
@endsection
```

在子页面不应该出现 @section 之外的其它内容，换句话说，这个文件只能定义@section

这是区块内容

```
@endsection
```

步骤三、在控制器中或路由中调用模板

```
//调用布局
Route::get('layout/child',function(){
    return view('child');
});
```

5-1-2 组件与slots

组件和 slots 能提供类似于区块和布局

原理，就是先定义一些组件的模板文件，然后在其它模板中调用这些组件，并可以传递数据给组件使用

步骤一、定义一个组件模板 比如：/resources/views/component/alert.blade.php

```
<div class="alert alert-danger">
   {{ $slot }}
</div>
```

上面的定义中，在调用此组件时 {{ $slot }} 的部份会被替换为新的内容，同时也可以传递参数进来，如，修改上面的定义

```
<div class="alert alert-danger">
   <h3>{{  $title }}</h3>
   {{ $slot }}
</div>
```

步骤二、在其它模板中加载这个组件 比如：/resources/views/testcomponent.blade.php

```
<html>
   <head>
      <title>测试组件 alert</title>
   </head>
   <body>
      <div>
         @component('component.alert')
            <strong>哇！</strong> 出现了一些问题！
         @endcomponent
      </div>
   </body>
</html>
```

步骤三、在控制器或路由中调用testcomponent模板

```
//调用组件
Route::get('component/alert',function(){
   return view('testcomponent');
});
```

5-1-3 显示数据

你可以使用「中括号」包住变量以显示传递至 Blade 视图的数据。如下面的路由设置：

```
Route::get('greeting', function () {
   return view('welcome', ['name' => 'Samantha']);
});
```

你可以像这样显示 name 变量的内容：

Hello, {{ $name }}.

当然也不是说一定只能显示传递至视图的变量内容。你也可以显示 PHP 函数的结果。事实上，你可以在 Blade 中显示任意的 PHP 代码：

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

Blade {{ }} 语法会自动调用 PHP htmlspecialchars 函数来避免 XSS 攻击。

当数据存在时输出

有时候你可能想要输出一个变量，但是你并不确定这个变量是否已经被定义，我们可以用像这样的冗长 PHP 代码表达：

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

事实上，Blade 提供了更便捷的方式来代替这种三元运算符表达式：

{{ $name or 'Default' }}

在这个例子中，如果 $name 变量存在，它的值将被显示出来。但是，如果它不存在，则会显示 Default 。

显示未转义过的数据

在默认情况下，Blade 模板中的 {{ }} 表达式将会自动调用 PHP htmlspecialchars 函数来转义数据以避免 XSS 的攻击。如果你不想你的数据被转义，你可以使用下面的语法：

Hello, {!! $name !!}.

要非常小心处理用户输入的数据时，你应该总是使用 {{ }} 语法来转义内容中的任何的 HTML 元素，以避免 XSS 攻击。

Blade & JavaScript 框架

由于很多 JavaScript 框架都使用花括号来表明所提供的表达式，所以你可以使用 @ 符号来告知 Blade 渲染引擎你需要保留这个表达式原始形态，例如：

```
<h1>Laravel</h1>

Hello, @{{ name }}.
```

在这个例子里，@ 符号最终会被 Blade 引擎剔除，并且 {{ name }} 表达式会被原样的保留下来，这样就允许你的 JavaScript 框架来使用它了。

@verbatim 指令

如果你需要在页面中大片区块中展示 JavaScript 变量，你可以使用 @verbatim 指令来包裹 HTML 内容，这样你就不需要为每个需要解析的变量增加 @ 符号前缀了：

```
@verbatim
  <div class="container">
    Hello, {{ name }}.
  </div>
@endverbatim
```

控制结构

除了模板继承与数据显示的功能以外，Blade 也给一般的 PHP 结构控制语句提供了方便的缩写，比如条件表达式和循环语句。这些缩写提供了更为清晰简明的方式来使用 PHP 的控制结构，而且还保持与 PHP 语句的相似性。

If 语句

你可以通过 @if, @elseif, @else 及 @endif 指令构建 if 表达式。这些命令的功能等同于在 PHP 中的语法：

```
@if (count($records) === 1)
  我有一条记录！
@elseif (count($records) > 1)
```

```
    我有多条记录！
@else
    我没有任何记录！
@endif
```

为了方便，Blade 也提供了一个 @unless 命令：

```
@unless (Auth::check())
    你尚未登录。
@endunless
```

循环

除了条件表达式外，Blade 也支持 PHP 的循环结构，这些命令的功能等同于在 PHP 中的语法：

```
@for ($i = 0; $i < 10; $i++)
    目前的值为 {{ $i }}
@endfor

@foreach ($users as $user)
    <p>此用户为 {{ $user->id }}</p>
@endforeach

@forelse ($users as $user)
    <li>{{ $user->name }}</li>
@empty
    <p>没有用户</p>
@endforelse

@while (true)
    <p>我永远都在跑循环。</p>
@endwhile
```

当循环时，你可以使用 循环变量 来获取循环中有价值的信息，比如循环中的首次或最后的迭代。

当使用循环时，你可能也需要一些结束循环或者跳出当前循环的命令：

```
@foreach ($users as $user)
    @if ($user->type == 1)
        @continue
```

```
    @endif

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

    @if ($user->number == 5)
        @break
    @endif
@endforeach
```

你也可以使用命令声明包含条件的方式在一条语句中达到中断:

```
@foreach ($users as $user)
    @continue($user->type == 1)

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

    @break($user->number == 5)
@endforeach
```

循环变量

当循环时，你可以在循环内访问 $loop 变量。这个变量可以提供一些有用的信息，比如当前循环的索引，当前循环是不是首次迭代，又或者当前循环是不是最后一次迭代：

```
@foreach ($users as $user)
    @if ($loop->first)
        This is the first iteration.
    @endif

    @if ($loop->last)
        This is the last iteration.
    @endif

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

如果你是在一个嵌套的循环中，你可以通过使用 $loop 变量的 parent 属性来获取父循环中的 $loop 变量：

```
@foreach ($users as $user)
  @foreach ($user->posts as $post)
    @if ($loop->parent->first)
      This is first iteration of the parent loop.
    @endif
  @endforeach
@endforeach
```

$loop 变量也包含了其它各种有用的属性：

属性	描述
$loop->index	当前循环所迭代的索引，起始为 0。
$loop->iteration	当前迭代数，起始为 1。
$loop->remaining	循环中迭代剩余的数量。
$loop->count	被迭代项的总数量。
$loop->first	当前迭代是否是循环中的首次迭代。
$loop->last	当前迭代是否是循环中的最后一次迭代。
$loop->depth	当前循环的嵌套深度。
$loop->parent	当在嵌套的循环内时，可以访问到父循环中的 $loop 变量。

注释

Blade 也允许在页面中定义注释，然而，跟 HTML 的注释不同的是，Blade 注释不会被包含在应用程序返回的 HTML 内：

```
{{-- 此注释将不会出现在渲染后的 HTML --}}
```

PHP

在某些情况下，它对于你在视图文件中嵌入 php 代码是非常有帮助的。你可以在你的模版中使用 Blade 提供的 @php 指令来执行一段纯 PHP 代码：

```
@php
  //
@endphp
```

虽然 Blade 提供了这个功能，但频繁地使用也同时意味着你在你的模版中嵌入了太多的逻辑了。

引入子视图

你可以使用 Blade 的 @include 命令来引入一个已存在的视图，所有在父视图的可用变量在被引入的视图中都是可用的。

```
<div>
   @include('shared.errors')

   <form>
      <!-- Form Contents -->
   </form>
</div>
```

尽管被引入的视图会继承父视图中的所有数据，你也可以通过传递额外的数组数据至被引入的页面：

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

当然，如果你尝试使用 @include 去引用一个不存在的视图，Laravel 会抛出错误。如果你想引入一个视图，而你又无法确认这个视图存在与否，你可以使用 @includeIf 指令：

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

请避免在 Blade 视图中使用 __DIR__ 及 __FILE__ 常量，因为他们会引用视图被缓存的位置。

四、小节

2. 课堂练习

3. 课后练习

4. 资料扩展