

## 01. Laravel 入门和安装

学习要点：

1. 版本化方案
2. 安装和配置
3. 课程规划

本节课我们来开始进入 Laravel 框架的学习，主要了解框架的安装和配置信息。

### 一. 版本化方案

1. Laravel 框架是一款简洁、优秀且优雅的 PHP 开发框架；
2. Laravel 到底怎么读，由于不是真实的单词，导致争论较多
3. 目前已知：['lærəvel'] ['la:rəvel'] ['la:rəvəl'] 有这几种；
4. Laravel 从 6.x 开始进入到版本化方案，每六个月发布一次版本；

版本	发布时间	Bug 修复截止时间	安全修复截止时间
5.5 (LTS)	2017 / 08 / 30	2019 / 08 / 30	2020 / 08 / 30
5.6	2018 / 02 / 07	2018 / 08 / 07	2019 / 02 / 07
5.7	2018 / 09 / 04	2019 / 03 / 04	2019 / 09 / 04
5.8	2019 / 02 / 26	2019 / 08 / 26	2020 / 02 / 26
6 (LTS)	2019 / 09 / 03	2021 / 09 / 03	2022 / 09 / 03
7	2020 / 03 / 03	2020 / 09 / 03	2021 / 03 / 03

5. 这种版本策略在软件版本上也经常看到，比如 Firefox 火狐浏览器；
6. 早期一直用 v3.5 这个版本，后来就开始疯狂升级，升级了几年现在 v73.0.1；
7. 那么对于这种升级策略来说，基础语法几乎不会有太多变动，不会增加学习成本；
8. 也正是如此，课程已经不写什么版本了，用最新版本基本无碍；

### 二. 安装和配置

1. PHP 环境是 7.3+，目前最新版本是 7.4(非正式版)，采用 wamp 或其它继承环境；
2. 我们采用 composer 来进行安装，官网也提供了安装器，只不过有点问题(目前)；
3. 对于 composer 软件安装，windows 下载 composer\_setup.exe 文件安装即可；
4. 而对于 Linux/Mac 平台，可以通过命令行进行下载安装；

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

5. 打开 windows 下的运行：cmd，然后运行如下代码（Mac 和 Linux 控制台）：

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

6. 如果上述地址产生阻碍，可以使用国内阿里云镜像(提高速度，避免报错)：

```
composer config -g repo.packagist composer https://mirrors.aliyun.com/composer/
```

7. 如果你是首次安装 Laravel，那么直接执行如下命令即可；

```
composer create-project --prefer-dist laravel/laravel laravel
```

8. 如果在本地开发环境，推荐使用内置服务器:8000 的命令去访问测试；

```
php artisan serve
```

9. 使用 PhpStorm 打开项目目录，安装 Laravel Plugin 代码提示插件；

10. 更多的配置和问题，参考官方手册安装篇查阅即可；

### 三. 课程规划

1. 框架内容较多，课程要分为两个篇幅，基础篇和扩展篇(不一定做)；
2. 基础篇基本框架的核心内容语法等，扩展篇主要是工具集/扩展等内容；

## 02. 路由的定义和控制器

学习要点：

1. 路由的定义
2. 创建控制器

本节课我们来开始进入学习路由模块，然后简单了解控制器的创建。

### 一. 路由的定义

1. 什么是路由？路由就是提供接受 HTTP 请求的路径，并和程序交互的功能；
2. 简单点理解，就是为了提供访问程序的 URL 地址，所做的一些设置工作；
3. `phpstorm` 支持 `cmd` 操作，左下角 `Terminal` 按钮展开，调整字体即可；
4. 输入 `php artisan serve` 命令后，即支持 `localhost:8000` 内置服务器；
5. 路由的定义文件在根目录 `routes/web.php` 中，可以看到 `welcome` 页面；
6. 我们创建一个路由，让它返回一段信息，并设置响应的 `url` 地址；

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

`http://localhost:8000/index`

7. 在路由定义上，我们采用了`::get()`这个方法，它接受的就是 `GET` 提交；
8. `::post()`、`::put()`、`::delete()`是表单和 `Ajax` 的提交接受方式；
9. `::any()`表示不管你是哪种提交方式，我智能的全部接收响应；
10. `::match()`表示接收你指定的提交方式，用数组作为参数传递；

```
Route::match(['get', 'post'], 'index', function () {  
    return 'Hello, World!';  
});
```

11. 在路由的规则和闭包区域，我们可以设置和传递路由参数；

```
Route::get('index/{id}', function ($id) {  
    return 'Hello, World!' . $id;  
});
```

`http://localhost:8000/index/5`

12. 上面例子中`{id}`表示在 `url` 的动态参数，比如数字 `5`；
13. 那么闭包的`$id`，可以接受 `url` 传递过来的 `5`，最终输出 `5`；

## 二. 创建控制器

1. MVC 模式中 C 代表控制器，用于接收 HTTP 请求，从而进行逻辑处理；

2. 有两种方式可以创建控制器，IDE 直接创建，或使用命令生成一个；

```
php artisan make:controller TaskController
```

3. 控制器目录在 app\Http\Controllers 下，使用 IDE 会自动生成命名空间；

```
namespace App\Http\Controllers;
```

```
class TaskController extends Controller
{
    public function index()
    {
        return 'index';
    }

    public function read($id)
    {
        return 'id:' . $id;
    }
}
```

4. 通过设置路由来访问创建好的控制器，参数二：控制器@方法名；

```
Route::get('task', 'TaskController@index');
```

```
Route::get('task/read/{id}', 'TaskController@read');
```

## 03. 路由参数.重定向.视图

学习要点：

1. 路由参数
2. 路由重定向
3. 路由视图

本节课我们来开始进入学习路由的参数设置、重定向和路由的视图。

### 一. 路由参数

1. 上一节课，我们已经学习了部分路由参数的功能，比如动态传递{id}；
2. 那么，有时这个参数需要进行约束，我们可以使用正则来限定必须是数字；

```
Route::get('task/read/{id}', 'TaskController@read')
    ->where('id', '[0-9]+'); //单个参数
    ->where(['id'=>'[0-9]+', 'name'=>'[a-z]+']); //多个参数
```

3. 如果想让约束 id 只能在 0-9 之间作用域全局范围，可以在模型绑定器里设置；
4. 模型绑定器路径为：app\Providers\RouteServiceProvider 的 boot()方法；

```
public function boot()
{
    Route::pattern('id', '[0-9]+');
    parent::boot();
}
```

5. 如果 id 已经被全局约束，在某个局部你想让它脱离约束，可以如下操作：

```
...->where('id', '.*');
```

### 二. 路由重定向

1. 可以设置访问一个路由的 URI，跳转到另一个路由的 URI，具体如下：

```
Route::redirect('index', 'task');
Route::redirect('index', 'task', 301); //状态码
```

2. 还有一个方法，可以直接让路由跳转返回 301 状态码而不用设置：

```
Route::permanentRedirect('index', 'task');
```

### 三. 视图路由

1. 在使用视图路由之前，我们先要创建一个视图(MVC)中的 V 部分；
2. 使用视图路由，有三个参数：1.URI(必)；2.名称(必)；3.参数(选)；

```
//参数1：URI，localhost:8000/task
//参数2：view，resources/views/task.blade.php
//参数3：传参，{$id}
Route::view('task', 'task', ['id'=>10]);
```

## 04. 路由命名和分组

学习要点：

1. 路由命名

2. 路由分组

本节课我们来学习一下路由模块中的命名功能和分组功能。

### 一. 路由命名

1. 给一个制定好的路由进行命名，可以生成 URL 地址或进行重定向；

```
Route::get('task', 'TaskController@index')
    ->name('task.index');
```

2. 在控制器区域，使用助手函数 `route()` 来获取路由生成的 URL 地址；

```
//生成url地址, http://localhost:8000/task
$route('task.index');      PS: URL 是 URI 的子集, 更多区别请百度;
```

3. `route()` 助手的第二参数为参数，第三参数为是否包含域名 URL；

```
// http://localhost:8000/task?id=10
$route('task.index', ['id'=>10]);
```

```
// /task?id=10
$url = route('task.index', ['id'=>10], false);
```

PS：如果需要更改成 /task/10 模式，路由需要相应更改 `task/{id}`

4. 使用 `redirect()` 助手结合 `route()` 生成一个重定向跳转，注意不要自我死跳；

```
//生成重定向
return redirect()->route('task.index', ['id'=>10]);
```

### 二. 路由分组

1. 路由分组功能是为了让大量路由共享路由属性，包括中间件、命名空间等；

```
//一个空的分组路由
Route::group([], function () {
    Route::get('index/{id}', function ($id) {
        return 'index' . $id;
    });
    Route::get('task/{id}', function ($id) {
        return 'task' . $id;
    });
});
```

2. 可以将中间件作用域路由分组中，有两种写法，至于中间件？后续章节讲解；

//引入中间件，方法一

```
Route::group(['middleware'=>'中间名'], function () {});
```

//引入中间件，方法二

```
Route::middleware(['中间件'])->group(function () {});
```

3. 可以设置路由路径前缀，通过 `prefix` 来设置，也有两种方法，具体如下：

//引入路由前缀，方法一

```
Route::group(['prefix'=>'api'], function () {});
```

//引入路由前缀，方法二

```
Route::prefix('api')->group(function () {});
```

4. 可以设置子域名，从而限定路由可执行的域名，有两种方法，具体如下：

//引入子域名，方法一

```
Route::group(['domain'=>'127.0.0.1'], function () {});
```

//引入子域名，方法二

```
Route::domain('127.0.0.1')->group(function () {});
```

5. 可以设置命名空间，让命名空间分配给控制器，让其得以访问，具体如下：

//命名空间，方法一

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

//命名空间，方法二

```
Route::namespace('Admin')->group(function () {});
```

PS：在 `Controller` 目录下创建 `Admin` 目录，再其目录下创建的控制器命名空间如下：

```
namespace App\Http\Controllers\Admin;
```

6. 可以设置名称前缀，方式两种，也可以嵌套，具体如下：

//名称前缀，方式一

```
Route::group(['as'=>'task.'], function () {
    Route::get('task', 'TaskController@index')->name('index');
    Route::get('task/url', 'TaskController@url');
});
```

//名称前缀，方式二

```
Route::name('task.')->group(function () {});
```

//生成 URL

```
$url = route('task.index');
return $url;
```

//嵌套方式命名前缀

```
Route::name('task.')->group(function () {
    Route::name('abc.')->group(function () {
```

```
Route::get('task', 'TaskController@index')->name('index');
});

Route::get('task/url', 'TaskController@url');
};

//生成 URL
$url = route('task.abc.index');
return $url;
```

## 05. 回退. 当前路由. 单行为

学习要点：

1. 单行为控制器
2. 路由回退
3. 当前路由

本节课我们来开始学习路由和控制器的一些方法：回退、当前路由、单行为；

### 一. 单行为控制器

1. 之前的课程，我们简单的创建和定义了控制器，并继承了控制器基类；
2. 为何要继承基类？因为继承基类后，可以使用基类的方法，比如中间件等；
3. 继承基类后除了支持中间件快捷使用，还支持验证、列队等快捷方法；

```
public function __construct()  
{  
    $this->middleware('中间件');  
}
```

4. 如果你想要定义一个只执行一个方法的控制器，可以使用单行为控制器；
5. 单行为控制器使用`__invoke()`方法，可以使用命令行创建；

```
php artisan make:controller OneController --invokable
```

```
//手工创建  
class OneController extends Controller  
{  
    public function __invoke()  
    {  
        return '单行为控制器';  
    }  
}
```

6. 单行为控制器，路由定义就不需要指定特定的方法，指定控制器即可；
7. 单行为控制器只是语义上的单行为，并没有限制创建更多方法访问；

```
Route::get('one', 'OneController');
```

### 二. 路由回退

1. 如果我们跳转到了一个不存在路由时，会产生 404 错误，体验不佳；
2. 可以使用回退路由，让不存在的路由自动跳转到你指定的页面去；
3. 注意：由于执行顺序问题，必须把回退路由放在所有路由的最底部；

```
Route::fallback(function () {  
    return redirect('/');  
});
```

4. 当然，你也可以制作一个自己的 404 页面，用回退路由加载这个页面；

```
Route::fallback(function () {
    return view('404');
});
```

### 三. 当前路由

1. 我们可以通过使用`::current()`系列方法，来获取当前路由的访问信息；

```
Route::get('index', function () {
    //当前路由信息
    dump(Route::current());
    //返回当前路由的名称
    return Route::currentRouteName();
    //返回当前路由指向的方法
    return Route::currentRouteAction();
})->name('localhost.index');
```

## 06. 响应设置和重定向

学习要点：

1. 响应设置
2. 重定向

本节课我们来开始进入学习响应设置，和跳转重定向。

### 一. 响应设置

1. 路由和控制器处理完业务都会返回一个发送到浏览器的响应：`return;`
2. 比如字符串会直接输出，而数组则会输出 `json` 格式，本身是 `Response` 对象；

```
return [1, 2, 3];           //输出 json 格式  
return response([1, 2, 3]); //同上  
return response()->json([1, 2, 3]); //同上
```

3. 如果使用 `response()` 输出的话，可以设置状态码和响应头信息；

```
return response('index', 201); //可以设置 HTTP 请求状态码
```

4. 也可以给 `HTTP` 添加或修改标头，比如将 `html` 解析模式改成文本 `plain` 模式；

```
return response('<b>index</b>')  
->header('Content-Type', 'text/plain'); //文本解析模式
```

5. 结合上面的响应操作，再结合 `view()` 视图功能，显示纯 `HTML` 代码页面；

```
return response()->view('task', ['id'=>10], 201)  
->header('Content-Type', 'text/plain');
```

### 二. 路由重定向

1. 重定向使用助手函数 `redirect()` 的 `to()` 方法，注意需要 `return` 才能跳转；

```
return redirect()->to('/'); //跳到首页  
return redirect()->to('task'); //跳转到 task  
return redirect()->to('task/url'); //跳转到 task/url
```

2. 也可以直接使用快捷方式直接进行跳转；

```
return redirect('/'); //跳到首页  
return redirect('task'); //跳转到 task  
return redirect('task/url'); //跳转到 task/url
```

3. `redirect()` 助手有一个对应的 `Facade` 模式对象；

```
return Redirect::to('/'); //Facade 模式，但需要 use 引入
```

4. 使用 `redirect()` 的 `route()` 方法，可以跳转到指定的命名路由 `URI`；

```
return redirect()->route('task.index'); //注意和 route() 方法区别
```

5. 使用 `redirect()` 的 `back()` 方法，可以重定向到上一个页面中；

```
return redirect()->back();  
return back(); //快捷方式
```

6. 使用 `redirect()` 的 `action()` 方法，可以直接重定向到控制器方法；

```
return redirect()->action('TaskController@index'); //需注册路由  
return redirect()->action('TaskController@index', ['id'=>10]);
```

7. 使用 `redirect()` 的 `away()` 方法，跳转到外部链接；

```
return redirect()->away('http://www.baidu.com'); //不带任何编码
```

## 07. 资源控制器

学习要点：

### 1. 资源控制器

本节课我们来开始学习控制器的快捷方法：资源控制器。

#### 一. 资源控制器

1. 声明：资源控制器是某个特定场景下的产物，完全理解需要 PHP 项目基础；
  2. 比如开发过博客系统，留言帖子系统之类，具有类似思维，否则你懂的...；
  3. 只是学习了 PHP 基础，就立刻学习框架的同学，可以过一遍即可(不影响后续)...；
  4. 有一种控制器专门处理 CURD(增删改查)，方法很多且方法名基本固定；
  5. 对于这种控制器，我们可以将它设置为资源型控制器，不要大量设置路由；
  6. 这里推荐直接使用命令行生成资源路由，比如：BlogController；
- ```
php artisan make:controller BlogController --resource
```
7. 生成了的资源控制器会产生 7 个方法，配置好路由后会自动生成相关内容

```
Route::resource('blogs', 'BlogController'); //单个资源路由
```

```
//批量定义资源路由
```

```
Route::resources([
    'blogs' => 'BlogController'
]);
```

| HTTP 类型   | 路由 URI            | 控制器方法     | 路由命名          | 描述        |
|-----------|-------------------|-----------|---------------|-----------|
| GET       | blogs             | index()   | blogs.index   | 获得数据列表    |
| GET       | blogs/create      | create()  | blogs.create  | 创建页面(表单页) |
| POST      | blogs             | store()   | blogs.store   | 创建页的接受处理  |
| GET       | blogs/{blog}      | show()    | blogs.show    | 获得一条数据    |
| GET       | blogs/{blog}/edit | edit()    | blogs.edit    | 编辑(表单页)   |
| PUT/PATCH | blogs/{blog}      | update()  | blogs.update  | 从编辑页中接受处理 |
| DELETE    | blogs/{blog}      | destroy() | blogs.destroy | 删除一条数据    |

8. 如果我们注册了资源路由，那么如上图的资源路由 URI 和名称均自动创建生效；

```
http://localhost:8000/blogs/10/edit //可以访问到 edit 方法
```

```
return route('blogs.store'); //可以通过助手 route() 了解是否注册
```

9. 还有一条命令可以直接查看目前可用的路由以及命名；

```
php artisan route:list
```

10. 我们也可以限制资源路由只开放部分方法或排除部分方法，可以用命令查看；

```
//只有 index(),show()可访问
Route::resource('blogs', 'BlogController')
    ->only(['index', 'show']);
```

```
//排除 index(),show()的其它方法可访问
Route::resource('blogs', 'BlogController')
    ->except(['index', 'show']);
```

11. 资源控制器还有一种不需要 HTML 页面方法的 API 路由，只提供数据接口；

```
//API 资源，并不需要 HTML 页面(create,edit)，会排除
Route::apiResource('blogs', 'BlogController');
```

```
//批量方式
Route::apiResources([
    'blogs' => 'BlogController'
]);
```

12. 当然，也支持一开始就生成一个不包含 HTML 页面方法的资源控制器；

13. 要注意的是，对应的资源路由，直接使用 api 资源路由即可；

```
php artisan make:controller CommentController --api
Route::apiResource('comments', 'CommentController');
```

## 08. 资源嵌套·浅嵌套·自定义

学习要点：

1. 嵌套资源
2. 资源自定义

本节课我们来开始学习控制器的资源嵌套功能、浅嵌套以及资源自定义。

### 一. 嵌套资源

1. 声明：资源控制器是某个特定场景下的产物，完全理解需要 PHP 项目基础；
2. 比如开发过博客系统，留言帖子系统之类，具有类似思维，否则你懂的...；
3. 只是学习了 PHP 基础，就立刻学习框架的同学，可以过一遍即可(不影响后续)...；
4. 在一篇博文(Blog)下有多条评论(Comment)，编辑某条博文下的一条评论；
5. 以上需求，可以通过嵌套资源路由来实现这个功能；

```
php artisan make:controller CommentController --resource
//嵌套资源路由
Route::resource('blogs.comments', 'CommentController');
```

| HTTP 类型   | 路由 URI                               | 控制器方法     | 路由命名                   |
|-----------|--------------------------------------|-----------|------------------------|
| GET       | blogs/{blog}/comments                | index()   | blogs.comments.index   |
| GET       | blogs/{blog}/comments/create         | create()  | blogs.comments.create  |
| POST      | blogs/{blog}/comments                | store()   | blogs.comments.store   |
| GET       | blogs/{blog}/comments/{comment}      | show()    | blogs.comments.show    |
| GET       | blogs/{blog}/comments/{comment}/edit | edit()    | blogs.comments.edit    |
| PUT/PATCH | blogs/{blog}/comments/{comment}      | update()  | blogs.comments.update  |
| DELETE    | blogs/{blog}/comments/{comment}      | destroy() | blogs.comments.destroy |

6. 以上需求，可以通过嵌套资源路由来实现这个功能，编辑方法以及传参如下：

```
public function edit($blog_id, $comment_id)
{
    return '编辑博文下的评论，博文 id : '.$blog_id.'，评论 id : '.$comment_id;
}
```

7. 而实际上，每个 id 都是独立唯一的，并不需要父 id 和子 id 同时存在；
8. 为了优化资源嵌套，通过路由方法->shallow() 实现浅层嵌套方法；

```
//浅层嵌套
Route::resource('blogs.comments', 'CommentController')->shallow();
```

9. 实现后的路由，在传递参数方法也比较精准，具体如下：

| HTTP 类型   | 路由 URI                       | 控制器方法     | 路由命名                   |
|-----------|------------------------------|-----------|------------------------|
| GET       | blogs/{blog}/comments        | index()   | blogs.comments.index   |
| GET       | blogs/{blog}/comments/create | create()  | blogs.comments.create  |
| POST      | blogs/{blog}/comments        | store()   | blogs.comments.store   |
| GET       | comments/{comment}           | show()    | blogs.comments.show    |
| GET       | comments/{comment}/edit      | edit()    | blogs.comments.edit    |
| PUT/PATCH | comments/{comment}           | update()  | blogs.comments.update  |
| DELETE    | comments/{comment}           | destroy() | blogs.comments.destroy |

```
public function edit($id)
{
    return '评论 id.' . $id;
}
```

10. 如果觉得资源路由命名过长，可以自己自定义，有两种方式：

```
->name('index', 'b.c.i');
->names([
    'index' => 'b.c.i'
]);
```

11. 如果觉得资源路由的参数不符合你的心意，也可以改变：

```
->parameter('blogs', 'id');
->parameters([
    'blogs' => 'blog_id',
    'comments' => 'comment_id'
]);
```

## 09. 表单伪造和 CSRF 保护

学习要点：

### 1. 表单伪造

本节课我们来开始学习表单伪造和 CSRF 保护的功能。

#### 一. 表单伪造

- 之前一直用的 GET 请求方式，而表单可以实现 POST 方式，我们来实验下；
- 先在 TaskController 创建两个方法，一个表单页，一个接受表单数据路由；

```
public function form()
{
    return view('form');
}

//表单页
Route::get('task/form', 'TaskController@form');

//接受表单数据
Route::any('task/getform', function () {
    return \Illuminate\Support\Facades\Request::method();
});
```

- 表单页以 post 发送，路由也使用 post 接受，以下表单提交会出现 419 错误；

```
<form action="/task/getform" method="post">
    用户名：<input type="text" name="user">
    <button type="submit">提交</button>
</form>
```

- 这是为了避免被跨站请求伪造攻击，框架提供了 CSRF 令牌保护，请求时验证；

```
<input type="hidden" name="_token" value="{{csrf_token()}}>
```

- 表单可以实现 POST 提交方式，那其它提交方式该如何实现呢？可以采用伪造技术；

```
<input type="hidden" name="_method" value="PUT">
```

- 对于 CSRF 令牌保护和表单伪造提交方式，也支持快捷方式的声明，如下：

```
@csrf
@method('PUT')
```

- 如果我们想让某些 URL 关闭 csrf 验证，可以设置 csrf 白名单；

- 白名单具体设置位置在：中间件目录下的 VerifyCsrfToken.php 文件；

- 当然，不建议直接注释掉这个验证 csrf 功能的中间件；

```
protected $except = [
    //
    'api/*',
];
```

## 10. 数据库配置入门

学习要点：

### 1. 配置数据库

本节课我们来开始学习数据库的配置方法，以及连接数据库。

#### 一. 配置数据库

1. 框架支持原生、查询构造器和 Eloquent ORM(关系型对象映射器)来操作数据库；
2. 数据库的配置在 config/database.php，如果是本地可以直接配置.env 文件；
3. 我们通过.env 文件配置数据库连接的相关信息，以提供给 database 读取；

```
DB_CONNECTION=mysql          // .env
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=grade
DB_USERNAME=root
DB_PASSWORD=123456

'mysql' => [
    'driver' => 'mysql',
    ...
]
```

4. 我们可以直接创建一个新的控制器 DataController 来测试数据库部分；
5. 数据库有一个专用类 DB，可以用它来实现原生查询和构造器查询；

```
// 使用 DB 类的 select() 方法执行原生 SQL
$user = DB::select('select * from laravel_user');
return $user;
```

6. 查询构造器主要通过 DB 类的各种数据库操作方法来实现，比如选定一条；

```
// 这里省去了 laravel_，需要在 database.php 配置
$user = DB::table('user')->find(19);
return [$user];
```

7. 由于火狐浏览器自动将 JSON 显示的很美化，而 find() 只返回对象；  
return response()->json(\$user); // 这种写法去掉了外围的数组下标 0

8. 使用 Eloquent ORM 模型来操作数据库，使用命令在 Http 目录下创建模型；

```
php artisan make:model Http/Models/User      // 默认在 app 目录
```

```
// 使用 Eloquent ORM 构建
$user = User::all();
return $user;
```

9. 上面使用模型来操作数据后，报错提示数据表是复数：`users`；
10. 而我们真实的数据库表为：`laravel_user`，为何会这样？？？
11. 前缀可以在 `database.php` 修改添加：`laravel_`，最终变为：`laravel_users`；
12. 由于模型编码规范要求数据表是复数，这里的复数并不是单纯加 `s`；
13. 可能会加 `es`，可能会加 `ies`，也可能是 `child` 编程 `children` 之类的；
14. 可以使用字符串助手：`Str::plural()` 来判断英文单词的复数情况：

```
return Str::plural('bus');           //buses
return Str::plural('user');          //users
return Str::plural('child');         //children
```

15. 你可以根据规范去更改数据表名称，或者强制使用现有的数据表名；

```
protected $table = 'user';
```

## 11. 构造器的查询. 分块. 聚合

学习要点：

1. 构造器查询
2. 分块. 聚合

本节课我们来开始学习数据库的构造器查询以及分块和聚合查询。

### 一. 构造器查询

1. `table()`方法引入相应的表，`get()`方法可以查询当前表的所有数据；

```
//获取全部结果  
$users = DB::table('users')->get();
```

2. `first()`方法，可以获取到第一条数据；

```
//获取第一条数据  
$users = DB::table('users')->first();
```

3. `value(字段名)`方法，可以获取到第一条数据的指定字段的值；

```
//获取第一条数据的 email 字段值  
$users = DB::table('users')->value('email');
```

4. `find(id)`方法，可以获取指定 `id` 的一条数据；

```
//通过 id 获取指定一条数据  
$users = DB::table('users')->find(20);
```

5. `pluck(字段名)`可以获取所有数据单列值的集合；

```
//获取单列值的集合  
$users = DB::table('users')->pluck('username');  
$users = DB::table('users')->pluck('username', 'email');
```

### 二. 分块. 聚合

1. 如果你一次性处理成千上万条记录，防止读取出错，可以使用 `chunk()` 方法；

```
//切割分块执行，每次读取 3 条，id 排序；  
DB::table('users')->orderBy('id')->chunk(3, function ($users) {  
    foreach ($users as $user) {  
        echo $user->username;  
    }  
    echo '-----<br>';  
});
```

2. 构造器查询提供了：`count()`、`max()`、`min()`、`avg()`和`sum()`聚合查询；

```
//聚合查询  
return DB::table('users')->count();  
return DB::table('users')->max('price');  
return DB::table('users')->avg('price');
```

3. 构造器查询两个判断记录是否存在方法：`exists()`和`doesntexists()`方法；

```
//判断是否存在  
return DB::table('users')->where('id', 19)->exists();  
return DB::table('users')->where('id', 18)->doesntExist();
```

PS：这里`DB::`第一个使用静态，返回查询对象，然后使用`->where`等各种查询方法，这些查询方法返回的还是查询对象，所以可以继续连缀操作。最后当遇到比如`get()`返回结果等方法时，停止连缀。所以，返回结果必须放在最后。

## 12. 构造器的查询表达式

学习要点：

1. **select** 查询
2. **where** 查询

本节课我们来开始学习数据库的构造器查询 **select** 和 **where** 查询：

### 一. **select** 查询

1. **select()**方法可以制定你想要的列，而不是所有列；

```
//设置显示的列，设置列别名
$users = DB::table('users')->select('username as name', 'email')->get();
```

2. **addSelect()**方法，可以在你基础的查询构造器上再增加想要显示的字段；

```
//给已经构建好的查询添加更多字段
$base = DB::table('users')->select('username as name', 'email');
$users = $base->addSelect('gender')->get();
```

3. **DB::raw()**方法可以在 **select()** 内部实现原生表达式，否则解析错误；

```
//结合原生 SQL 实现复杂查询
$users = DB::table('users')->select(DB::raw('COUNT(*) AS id, gender'))
    ->groupBy('gender')
    ->get();
```

4. 也可以直接使用 **selectRaw()** 方法实现内部原生；

```
//或者直接使用 selectRaw() 方法实现原生
$users = DB::table('users')->selectRaw('COUNT(*) AS count, gender')
    ->groupBy('gender')
    ->get();
```

5. 还可以通过 **havingRaw()** 方法实现更精准的分组筛选；

```
//使用 havingRaw 方法实现分组筛选
$users = DB::table('users')->selectRaw('COUNT(*) AS count, gender')
    ->groupBy('gender')
    ->havingRaw('count>5')
    ->get();
```

### 二. **where** 查询

1. **where()**查询，即条件查询，完整形式需要字段表达式和值三个；

```
//where 查询完整形式
$users = DB::table('users')->where('id', '=', 19)->get();
```

2. 大部分情况下，是等于用的比较多，就可以省略掉=号参数；

```
//where 查询完整形式  
$users = DB::table('users')->where('id', 19)->get();
```

3. 当然，还有>、<、>=、<=、<>、like 等操作符；

```
users = DB::table('users')->where('price', '>=', 95)->get();  
$users = DB::table('users')->where('username', 'like', '%\j%')->get();
```

4. 如果条件较多，可以用数组来分别添加条件，具体如下：

```
//如果条件都是等于，查看 SQL 语句用->toSql()替换->get()
```

```
$users = DB::table('users')->where([  
    'price'      => 90,  
    'gender'     => '男'  
)->get();
```

//如果条件非等于

```
$users = DB::table('users')->where([  
    ['price', '>=', 90],  
    ['gender', '=', '男']  
)->get();
```

## 13. 构造器的 where 派生查询

学习要点：

### 1. where 派生查询

本节课我们来开始学习数据库的构造器查询中 where 派生查询。

#### 一. where 派生查询

1. orWhere()方法，可以通过连缀实现两个或以上的 or 条件查询；

```
//where() + orWhere 实现 or 条件查询
$users = DB::table('users')
    ->where('price', '>', 95)
    ->orWhere('gender', '女')
    ->toSql();
```

2. 通过闭包，我们还可以构建更加复杂的 orWhere 查询；

```
//orWhere()结合闭包查询
$users = DB::table('users')
    ->where('price', '>', '95')
    ->orWhere(function ($query) {
        $query->where('gender', '女')
        ->where('username', 'like', '%小%');
    })->toSql();
```

3. whereBetween()可以实现区间查询，比如价格在一个区间内的用户；

```
//whereBetween 查询区间价格 60~90 之间
$users = DB::table('users')->whereBetween('price', [60, 90])->toSql();
```

PS：这里还支持相关三种：whereNotBetween/orWhereBetween/orWhereNotBetween；

4.whereIn()可以实现数组匹配查询，比如匹配出数组里指定的数据；

```
//whereIn 查询数组里匹配的数值
$users = DB::table('users')->whereIn('id', [20, 30, 50])->toSql();
```

PS：这里还支持相关三种：whereNotIn/orWhereIn/orWhereNotIn；

5. whereNull()可以查询字段为 Null 的记录；

```
//whereNull 查询字段值为 Null 的记录
$users = DB::table('users')->whereNull('uid')->toSql();
```

PS：这里还支持相关三种：whereNotNull/orWhereNull/orWhereNotNull；

6. `whereDate()`可以查询指定日期的记录；

```
//whereYear 查询指定日期的记录，或大于  
$users = DB::table('users')->whereDate('create_time', '2018-12-11')->toSql();
```

PS: 这里还支持相关四种：`whereYear`/`whereMonth`/`whereDay`/`whereTime`, 支持 `or` 前缀；

PS: 三个参数支持大于小于之类的操作 `orWhereDate('create_time', '>', '2018-12-11')`

## 14. 构造器的排序分组.子查询

学习要点：

1. 排序分组
2. 子查询

本节课我们来开始学习数据库的构造器查询中的子查询、排序、分组等。

### 一. 排序分组

1. 使用 `whereColumn()` 方法实现两个字段相等的查询结果；

```
//判断两个相等的字段，同样支持 orWhereColumn()
//支持符号'create_time','>', 'update_time'
//支持符号支持数组多个字段格式['create_time','>', 'update_time']
$users = DB::table('users')
    ->whereColumn('create_time', 'update_time')
    ->get();
```

2. 使用 `orderBy()` 方法实现 `desc` 或 `asc` 排序功能。

```
//支持 orderByRaw 和 orderByDesc 倒序方法
$users = DB::table('users')
    ->orderBy('id', 'desc')
    ->get();
```

3. 使用 `latest()` 方法设置时间倒序来排，默认时间字段是 `created_at`；

```
//按照创建时间倒序排，默认字段 created_at
$users = DB::table('users')->latest('create_time')->toSql();
```

4. 使用 `inRandomOrder()` 方法来随机排序，得到一个随机列表；

```
//随机排序
$users = DB::table('users')->inRandomOrder()->get();
```

5. 使用 `skip()` 和 `take()` 限制结果集，或使用 `offset()` 和 `limit()`；

```
//从第 3 条开始，显示 3 条
$users = DB::table('users')->skip(2)->take(3)->toSql();
$users = DB::table('users')->offset(2)->limit(3)->get();
```

6. 使用 `when()` 方法可以设置条件选择，执行相应的 SQL 语句；

```
//when 实现条件选择
$users = DB::table('users')->when(true, function ($query) {
    $query->where('id', 19);
}, function ($query) {
    $query->where('username', '辉夜');
})->get();
```

7. 如果 MySQL 在 5.7+，有支持 JSON 数据的新特性；

```
$users = DB::table('users')->where('list->id', 19)->first();
```

## 二. 子查询

1. 使用 `whereExists()` 方法实现一个子查询结果，返回相应的主查询；

//通过 books 表数据，查询到 users 表关联的所有用户

```
$users = DB::table('users')->whereExists(function ($query) {
    $query->selectRaw(1)
        ->from('books')
        ->whereRaw('laravel_books.user_id = laravel_users.id');
})->toSql();
```

//`whereRaw` 这句也可以替代为：`whereColumn('books.user_id', 'users.id')`；

PS: `select 1 from`，一般用于子查询的手段，目的是减少开销，提升效率，深入请搜索；

2. 也可以使用 `where(字段, function())` 闭包，来实现一个子查询；

//`id=子查询返回的 user_id`

```
$users = DB::table('users')->where('id', function ($query) {
    $query->select('user_id')
        ->from('books')
        ->whereColumn('books.user_id', 'users.id');
})->toSql();
```

## 15. 构造器的 `join` 查询

学习要点：

### 1. `join` 查询

本节课我们来开始学习数据库的构造器查询中 `join` 查询。

#### 一. `join` 查询

- 使用 `join` 实现内联接的多表查询，比如三张表进行 `inner join` 查询；

```
$users = DB::table('users')
    ->join('books', 'users.id', '=', 'books.user_id')
    ->join('profiles', 'users.id', '=', 'profiles.user_id')
    ->select('users.id', 'users.username', 'users.email',
              'books.title', 'profiles.hobby')
    ->get();
```

- 也可以使用 `leftJoin` 左连接或 `rightJoin` 右连接实现多表查询；

```
$users = DB::table('users')
    ->leftJoin('books', 'users.id', '=', 'books.user_id')
    ->rightJoin('profiles', 'users.id', '=', 'profiles.user_id')
    ->get();
```

- 使用 `crossJoin` 交叉连接查询，会生成笛卡尔积，再用 `distinct()` 取消重复；

```
$users = DB::table('users')
    ->crossJoin('books')
    ->select('username', 'email')
    ->distinct()
    ->get();
```

- 如果你想要实现闭包查询，和 `where` 类似，只不过要用 `on` 和 `orOn` 方法；

```
$users = DB::table('users')
    ->join('books', function ($join) {
        //支持 orOn 连缀
        $join->on('users.id', '=', 'books.user_id');
    })->toSql();
```

PS: `on()` 方法后面如果想要再增加筛选条件，可以追加 `where()`；

- 使用 `joinSub` 实现子连接查询，将对应的内容合并在一起输出；

```
//子连接查询
$query = DB::table('books')->selectRaw('user_id,title');
$users = DB::table('users')->joinSub($query, 'books', function ($join) {
```

```
$join->on('users.id', '=', 'books.user_id');
})->get();
```

6. 使用 `union()` 或 `unionAll()` 方法实现两个查询的合并操作；

```
//union 取消重复，unionAll 不取消重复
$query = DB::table('users');
$users = DB::table('users')
    ->union($query)
    ->get();
```

## 16. 构造器的增删改

学习要点：

### 1. 增删改操作

本节课我们来开始学习数据库的构造器查询中增删改操作。

#### 一. 增删改操作

- 使用 `insert()` 方法可以新增一条或多条记录；

```
//新增一条记录
DB::table('users')->insert([
    'username' => '李白',
    'password' => '123456',
    'email'     => 'libai@163.com',
    'details'   => '123'
]);
```

```
//新增多条记录
DB::table('users')->insert([
    [...],
    [...]
]);
```

- 使用 `insertOrIgnore()` 方法，可以忽略重复插入数据的错误；

```
//忽略重复新增数据的错误
DB::table('users')->insertOrIgnore([
    'id'        => 304,
    'username'  => '李白',
    'password'  => '123456',
    'email'     => 'libai@163.com',
    'details'   => '123'
]);
```

- 使用 `insertGetId()` 方法，获取新增后的自增 ID；

```
//获取新增后返回的 ID
$id = DB::table('users')->insertGetId([
    'username'  => '李白',
    'password'  => '123456',
    'email'     => 'libai@163.com',
    'details'   => '123'
]);

return $id;
```

4. 使用 `update()`方法，可以通过条件更新一条数据内容；

```
//更新修改一条数据
DB::table('users')->where('id', 304)
    ->update([
        'username' => '李红',
        'email'     => 'lihong@163.com'
    ]);
```

5. 使用 `updateOrInsert()`方法，可以先进行查找修改，如不存在，则新增；

```
//参数1：修改的条件
//参数2：修改的内容(新增的内容)
DB::table('users')->updateOrInsert(
    ['id'=>307],
    ['username'=>'李黑', 'password'=>'654321', 'details'=>'123']
);
```

6. 对于 `json` 数据，新增和修改的方法和正常数据类似；

```
//新增时，转换为 json 数据
'list'      => json_encode(['id'=>19])

//修改时，使用 list->id 指定
DB::table('users')->where('id', 306)
    ->update([
        'list->id' => 20
    ]);
```

7. 更新数据时，可以使用自增 `increment()` 和自减 `decrement()` 方法；

```
//默认自增/自减为1，可设置
DB::table('users')->where('id', 306)->increment('price');
DB::table('users')->where('id', 306)->increment('price', 2);
```

8. 使用 `delete()` 删除数据，一般来说要加上 `where` 条件，否则清空；

```
//删除一条数据
DB::table('users')->delete(307);
DB::table('users')->where('id', 307)->delete();

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

## 17. 模型的定义

学习要点：

1. 默认设置
2. 模型定义

本节课我们来开始学习数据库的模型部分的定义和默认值的设置。

### 一. 默认设置

1. 框架可以使用 Eloquent ORM 进行数据库交互，也就是关系对象模型；
2. 在数据库入门阶段，我们已经创建了一个 `User.php` 模型，如下：

```
php artisan make:model Http\Models\User           //默认在 app 目录
```

3. 而调用的时候，我们也知道表名要遵循它默认规则，修改为复数，或特定；

```
class User extends Model
{
    protected $table = 'user';
}
```

4. 系统假定你的主键为 `id`，如果你要修改默认主键，可以特定；

```
protected $primaryKey = 'xid';
```

5. 系统假定你的主键 `id` 为自增性，意味着是主键会自动转换 `int` 类型；

6. 如果你希望不是自增，非数值类型主键，可以设置取消；

```
public $incrementing = false;
```

7. 如果你主键不是一个整数，那么需要`$keyType` 设置为 `string`；

```
protected $keyType = 'string';
```

8. 系统默认情况下会接管 `created_at` 和 `updated_at` 两个时间戳列；

9. 如果不想让系统干涉这两个列，可以设置 `false` 取消；

```
public $timestamps = false;
```

10. 如果你想自定义时间戳的格式，可以设置；

```
protected $dateFormat = 'U';
```

11. 可以更改创建时间 `created_at` 和更新时间 `updated_at` 字段名；

```
const CREATED_AT = 'create_time';
const UPDATED_AT = 'update_time';
```

12. 默认读取 `database.php` 配置的数据库连接，也可以在模型端局部更改；

```
protected $connection = 'mysql';
```

## 二. 模型定义

1. 之前在查询构造器部分，把常用的数据库操作基本讲完，模型大体相同；

2. 比如，我们要查询所有数据，直接使用模型::all()即可；

```
//查询所有记录
$users = User::get();           //或 all()
return [$users];
```

3. 也可以像查询构造器一样，添加各种各样的条件，写法一样；

```
//查询性别为男，价格大于 90，限制显示 2 条
```

```
$users = User::where([
    ['gender', '=', '男'],
    ['price', '>', 95]
])->limit(2)->get();
```

4. 虽然安装了插件，但模型还是没有代码提示，可以通过安装插件解决；

```
composer require barryvdh/laravel-ide-helper
```

```
php artisan ide-helper:generate - 为 Facades 生成注释
```

```
php artisan ide-helper:models - 为数据模型生成注释
```

```
php artisan ide-helper:meta - 生成 PhpStorm Meta file
```

5. 其它查询方法基本和查询构造器一样，如果有不一样，参考错误提示；

6. 这里列出官网给出示例的方法，对照实验(结合详细文档，重复较多)；

- (1) .find(1) //通过主键查找
- (2) .first() //查找第一个
- (3) .firstWhere() //找到查询中的首个
- (4) .find([1,2,3]) //通过数组查找
- (5) .firstOr() //查找首个返回，支持闭包
- (6) .firstOrFail() //找不到时返回异常
- (7) .count()、max()等集合 //集合操作

PS：还有很多在查询构造器中的方法，比如排序、分组子查询等等都可以使用(并未一一验证)。

## 18. 模型的增删改

学习要点：

### 1. 增删改操作

本节课我们来开始学习数据库的模型的增删改的操作。

#### 一. 增删改操作

- 新增方法如下，注意：默认模型接管 `created_at` 和 `updated_at`；

```
$users = new User();
$users->username = '辉夜';
$users->password = '123';
$users->email = 'huiye@163.com';
$users->details = '123';
$users->save();
```

- 更新，只要是查找到一条数据的情况下使用 `save()` 就是更新；

```
$users = User::find(321);
$users->username = '夜辉';
$users->save();
```

- 使用 `update()` 方法实现批量更新；

```
User::where('username', '夜辉')
    ->update([
        'username' => '辉夜'
    ]);
```

- 使用 `create()` 方法实现新增，但需要在模型端设置批量赋值的许可；

```
User::create([
    'username' => '辉夜',
    'password' => '123',
    'email'     => 'huiye@163.com',
    'details'   => '123',
]);
```

// 许可批量赋值， 默认不可

```
protected $fillable = [
    'username',
    'password',
    'email',
    'details'
];
```

```
//不许可的批量赋值，不可和$fillable 同时使用  
//protected $guarded = ['uid'];  
  
//如果取消批量赋值限制，直接如下  
protected $guarded = [];
```

PS：必须在模型中定义批量赋值的可填充字段，否则无法生效；防止用户不小心设置新值；

5. 使用 `delete()`方法，可以删除数据；

```
$users = User::find(332);  
$users->delete();  
  
//批量删除  
$users = User::where('username', '夜辉');  
$users->delete();
```

6. 如果你是通过主键 `id` 删除，那使用 `destroy(id)`方法，免去查询操作；

```
//通过主键删除  
User::destroy(328);
```

## 19. 批量赋值和软删除

学习要点：

1. 批量赋值
2. 软删除

本节课我们来开始学习数据库的批量赋值的原因和软删除的方法。

### 一. 批量赋值

1. 上一节增删改中，新增中我们发现需要进行批量赋值的许可；
2. 一般情况下，是为了防止提交过来的字段在部分场景中不需要或不能；
3. 所以，我们需要通过黑白名单机制进行过滤掉必要的字段；

```
//通过提交过来的数据一次性新增  
User::create(\Request::all());
```

### 二. 软删除

1. 什么叫软删除？它相对于真实的删除，而并非真正的删除，只是隐藏了；
2. 首先，需要在数据库创建一个字段 `deleted_at`(默认)，用于判断是否被软删除；
3. 默认设置这个字段为空(`null`)，如果写入数据，成为非空状态，则说明被删除；
4. 开启软删除的功能，需要在模型端设置一下：

```
//开启软删除功能  
use SoftDeletes;
```

5. 当开启了软删除功能，之前的删除操作，都会变成更新操作，给 `deleted_at` 赋值；

```
//删除一  
$users = User::find(82);  
$users->delete();
```

```
//删除二  
User::destroy(81);
```

6. 而当我们开启了软删除的功能后，此时通过正常的数据获取列表，会自动隐藏；

```
//软删除的数据不可见  
$users = User::get();  
return [$users];
```

```
//单独获取被软删除的数据不行  
$users = User::find(82);  
return [$users];
```

7. 如果需要查询包含软删除的数据，通过 `withTrashed()`方法实现；

```
//获取包含软删除的数据
$users = User::withTrashed()->get();
return [$users];

//获取某个被软删除的数据(即使不是软删除的也可以搜索到)
$users = User::withTrashed()->find(82);
return [$users];
```

8. 如果只想搜索出被软删除的数据，通过 `onlyTrashed()`方法实现；

```
//获取所有软删除的数据
$users = User::onlyTrashed()->get();
return [$users];

//获取某个被软删除的数据(只有软删除的数据才可以被搜索到)
$users = User::onlyTrashed()->find(82);
return [$users];
```

9. 如果要对这个数据进行软删除的判断，是否是被软删除的，可以使用 `trashed()`；

```
//判断是否是被软删除的数据
$users = User::withTrashed()->find(81);
return $users->trashed();
```

10. 如果想将软删除的数据恢复正常(类似从回收站还原)，使用 `restore()`方法；

```
//将被软删除的数据回复正常
$users = User::onlyTrashed()->find(82);
$users->restore();
```

11. 如果开启了软删除，还需要强行真实的永久删除，可以使用 `forceDelete()`方法；

```
//开启软删除时的真实永久删除
$users = User::onlyTrashed()->find(82);
$users->forceDelete();
```

## 20. 模型的作用域

学习要点：

1. 本地作用域
2. 全局作用域

本节课我们来开始学习数据库模型的本地作用域和全局作用域的设置方法。

### 一. 本地作用域

1. 很多情况下，我们在数据查找时有一部分条件会被重复且大量使用；
2. 而这个条件，可能只是在这个模型对应的数据表使用，别的表并不使用；
3. 那么这种情况，可以使用本地作用域的方式，将常用的 SQL 封装起来；
4. 比如：用户模块中，我们大量查询需要查询性别为男，且其它条件的 SQL；

```
$users = User::where('gender', '男')
    ->where('price', '>', 90)
    ->get();
```

PS：我们可以将性别为男这个片段，封装成一个单独的方法，然后统一在这个模型下调用；

```
//App\Http\Models;
//本地作用域，搜索自动添加为“男”的条件
//语法：scope 开头，后面名称尽可能包含语义
public function scopeGenderMale($query)
{
    return $query->where('gender', '男');
}

//当然，如果赶紧单词太长，直接 gm()也行
$users = User::genderMale()
    ->where('price', '>', 90)
    ->get();
```

5. 上面的方法比较死板，适合简单粗暴，如果想要灵活多变，支持传递参数；

```
//参数可以是 1 个或多个
$users = User::gender('女', -3)
    ->where('price', '>', 90)
    ->get();

//参数 2 和 3，接受控制器传递过来的 1, 2
public function scopeGender($query, $value, $value2)
{
    return $query->where('gender', $value)->where('status', $value2);
}
```

## 二. 全局作用域

1. 全局作用域，顾名思义就是在任意地方都可以有效的封装条件；
2. 比如有个需求，不管在哪里操作，总是显示 `status` 为 1 的用户；
3. 首先在 `app` 目录下创建一个用于全局作用域的目录：`Scopes`；
4. 创建一个用于设置 `status` 为 1 的全局作用域的类，它需要实现 `scope` 接口；

```
namespace App\Scopes;

//这里引用代码自动生成
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Scope;

class StatusScope implements Scope
{
    /**
     * @inheritDoc
     */
    public function apply(Builder $builder, Model $model)
    {
        // TODO: Implement apply() method.
        return $builder->where('status', 1);
    }
}
```

5. 此时，还不能实现全局，因为需要在模型设置个开关，让其富有灵活性；

```
//启用全局作用域
protected static function booted()
{
    parent::booted(); // TODO: Change the autogenerated stub
    static::addGlobalScope(new StatusScope());
}
```

PS: 而在控制器端，并不需要做任何设置，即可自动添加 `status=1` 的条件；

6. 当然，如果这个全局只是针对某个模块，并不需要创建一个全局类，直接闭包即可；

```
static::addGlobalScope('status', function (Builder $builder) {
    return $builder->where('status', 1);
});
```

PS: 注意 `Builder` 引入的文件和全局类引入的文件一致，如果引入别的同名类会错；

7. 如果某个查询，并不需要这个全局条件，可以单独移出掉；

```
//取消名称为 status 的全局  
$users = User::withoutGlobalScope('status')->get();  
//取消全局类的条件  
$users = User::withoutGlobalScope(StatusScope::class)->get();
```

PS：还有 `withoutGlobalScopes([])` 方法，传递参数取消多个全局；

## 21. 模型的访问器和修改器

学习要点：

1. 访问器
2. 修改器

本节课我们来开始学习数据库模型的访问器和修改器的使用。

### 一. 访问器

1. 访问器：就是在获取数据列表时，拦截属性并对属性进行修改的过程；
2. 比如，我们在输出性别时，在性别左右加上括号，或给邮件转换为大写；

```
//访问器，前固定 get，后固定 Attribute，Gender 是字段名
//参数$value 是源字段值，可修改返回
public function getGenderAttribute($value)
{
    return '【.$value.】';
}
```

PS：如果字段名是两个单词中间是下划线：user\_name，那么方法名：getUserAttributeName()

3. 我们也可以创建一个虚拟字段，用已有的数据字段进行整合，不过要进行数据追加；

```
//将虚拟字段追加到数据对象列表里去
protected $appends = ['info'];

//创建一个虚拟字段
public function getInfoAttribute()
{
    return $this->username . '-' . $this->gender;
}
```

PS：注意，如果 gender 之前已经有访问器修改过，上面的方法会得到修改过的结果；

PS：如果要使用源字段进行创建虚拟字段，需要使用下面这种：

```
return $this->attributes['username'] . '-' . $this->attributes['gender'];
```

### 二. 修改器

1. 修改器，相对于访问器，是在写入的时候拦截，进行修改再写入；

```
//修改器，写入数据时，将邮箱转换为大写
public function setEmailAttribute($value)
{
    $this->attributes['email'] = strtoupper($value);
}
```

2. 可以添加默认的日期列， 默认 `created_at` 和 `updated_at`；  
`//设置可以自动写入日期的列  
protected $dates = [  
 'details'  
];`

3. 可以设置字段输出的类型， 比如设置一个布尔型， 输出时就是 `true` 和 `false`；  
`//设置字段类型  
protected $casts = [  
 'details' => 'boolean'  
];`

## 22. 集合的使用

学习要点：

1. 创建集合
2. 集合方法

本节课我们来开始学习数据集合的创建和使用方法。

### 一. 创建集合

1. 什么是集合？即：它是一种更具读取性和处理能力的数组封装；
2. 比如，我们从数据库得到的数据列表，它就是一种集合；
3. 数据集合，提供了大量的方法方便我们进行各种操作；
4. 除了数据库对象返回的数据集合之外，我们还可以自行创建数据集合；

```
//创建一个数据集合
$collection = collect(['张三', '李四', '王五', null]);
```

```
//使用 dd 查看它的类型
dd($collection);
```

```
//直接 return 可以返回
return $collection;
```

5. 数据集合提供了大概有三十多个(31?)处理数据集合的方法，可链式调用；
6. 这里我们把最常用的演示一遍，剩下的所有，可以自行参考手册方法列表；

```
//以底层数组形式输出
return $collection->all();
```

```
//map 方法，类似访问器，可修改输出
return $collection->map(function ($value, $key) {
    return $key.'.'.$value.'';
});
```

```
//支持链式，reject 移出非 true 的值
return $collection->reject(function ($value, $key) {
    return $value === null;
})->map(function ($value, $key) {
    return $key.'.'.$value.'';
});
```

```
//filter 筛选为 true 的值，和 reject 相反
return $collection->filter(function ($value, $key) {
    return $value === null;
});
```

```
//search 找到后返回 key , 找不到返回 false
return $collection->search('王五');

//集合的分割
return $collection->chunk(2);

//迭代输出
$collection->each(function ($item, $key) {
    echo $item;
});
```

PS：这里就介绍这么多，更多的我们去手册扫一遍。做项目时，凭着记忆回头查，慢慢就熟了；

PS：下一节，我们会花一节课，把最常用的方法再运行一遍加深印象；

7. 如果三十多个方法都没有你要的，还可以自定义方法，比如说所有英文大写；

```
$collection = collect(['Mr.Zhang', '李四', '王五', null]);
```

```
Collection::macro('toUpper', function () {
    //dd($this);
    return $this->map(function ($value) {
        return strtoupper($value);
    });
});

return $collection->toUpper();
```

## 23. 集合的常用方法

学习要点：

### 1. 常用方法

本节课我们来开始学习数据集合的常用方法，数了下（119个）。

#### 一. 常用方法

1. `all()`方法，转换为属性形式输出，使用 `dd` 方法看类型；

```
$collection = collect([1, 2, 2, 3, 4, 4, 4]);
dd($collection->all());
```

PS: `$collection->dd()`方法可以以 `dd()` 模式输出，还有 `dump()` 模式；

2. `avg()`方法返回平均值；

```
//返回平均值
$collection = collect([1, 2, 3, 4]);
return $collection->avg();
```

```
//返回分组平均值
$collection = collect([('男'=>1], ['女'=>1], ['男'=>3]));
return $collection->avg('男');
```

3. `count()`方法返回集合总数；

```
return $collection->count();
```

PS: 相关的还有 `sum()`、`min()`、`max()` 等统计；

4. `countBy()`方法返回数值出现的次数或回调函数指定值出现的次数；

```
//值出现的次数
$collection = collect([1, 2, 2, 3, 4, 4, 4]);
return $collection->countBy();
```

```
//回调搜索相同指定片段的值的次数
$collection = collect(['xiaoxin@163.com', 'yihu@163.com', 'xiaoying@qq.com']);

return $collection->countBy(function ($value) {
    return substr(strrchr($value, '@'), 1);
});
```

PS: 相关的还有 `groupBy()`、`keyBy()` 方法；

5. `diff()`方法返回集合数组之间不相同的部分，组合新的集合；

```
//diff 返回两个集合中不相同的
$collection = collect([1, 2, 3, 4, 5]);
return $collection->diff([3, 5]);
```

PS: 其中还有 `diffAssoc()`、`diffKeys()` 派生方法;

6. `duplicates()` 返回重复的值;

```
$collection = collect([1, 2, 2, 3, 4, 5, 5, 6]);
return $collection->duplicates(); //严格派生方法: duplicatesStrict()
```

7. `first()` 返回成立后的第一个值;

```
//返回判断成立的第一条数值
$collection = collect([1, 2, 3, 4]);
return $collection->first(function ($value) {
    return $value > 2;
});
```

PS: 相关的还有 `every()`、`except()`、`only()`、`firstWhere()`、`last()` 等方法;

8. `flatten()` 将多维数组转换为一维;

```
$collection = collect(['name'=>'Mr.Lee',
                      'details'=>['gender'=>'男', 'age'=>100]]);
return $collection->flatten();
```

9. `get()` 通过键名找值;

```
$collection = collect(['name'=>'Mr.Lee', 'gender'=>'男']);
return $collection->get('name');
```

PS: 相关的还有 `pluck()` 等;

10. `has()` 判断集合中是否存在指定键;

```
return $collection->has('name');
```

11. `pop()` 移出集合中最后一个值;

```
$collection = collect([1, 2, 3, 4, 5]);
//$/collection->pop();
return $collection;
```

PS: 相关的还有 `pull()`、`push()`、`put()` 方法;

12. `slice()` 返回指定值后续的集合;

```
$collection = collect([1, 2, 3, 4, 5]);
return $collection->slice(3);
```

PS: 相关的还有 `splice()` 等方法;

13. `sort()` 返回指定值后续的集合;

```
$collection = collect([3, 1, 5, 2, 7]);
return $collection->sort()->values();           //需要配合 values()方法
```

PS: 相关的有 `sortBy()`、`sortByDesc()`、`sortKeys()` 等;

14. `where()` 系列方法，和数据库条件一样;

```
$collection = collect([
    ['name'=>'Mr.Lee', 'gender'=>'男'],
    ['name'=>'Miss.Zhang', 'gender'=>'女']
]);
return $collection->where('name', 'Mr.Lee');
```

## 24. 模型的数据集合

学习要点：

### 1. 数据集合

本节课我们来开始学习数据库模型的数据集合的用法。

#### 一. 数据集合

1. 数据集合，就是已经将模型方法 `get()` 获取到的数据再进行处理；
2. 比如：`map()` 方法，通过它可以实现类似访问器一样对字段进行处理的效果；

```
$users = User::get();

//使用集合方法 map 可以对输出的字段进行过滤
$women = $users->map(function ($user) {
    $user->email = strtoupper($user->email);
    return $user;
});

return [$women];
```

PS：数据集合支持连缀操作，和数据库连缀一样；

3. 使用 `reject()` 方法，可以获取条件之外的数据内容；

```
$women = $users->reject(function ($user) {
    return $user->gender != '女';
})->map(function ($user) {
    return $user;
});
```

4. 下面列出常用的集合方法列表：

```
//判断集合中是否包含指定的模型实例
return $users->contains(19);
return $users->contains(User::find(19));
//返回不在集合中的所有模型
return $users->diff(User::whereIn('id', [19,20,21])->get());
//返回给定主键外的所有模型
return $users->except([19,20,21]);
//集合也有 find 方法
return $users->find(19);
//返回集合的数量
return $users->count();
//返回所有模型的主键
return $users->modelKeys();
```

```
//返回主键的所有模型  
return $users->only([19,20,21]);  
//返回集合中的唯一模型  
return $users->unique();
```

## 25. 模型的一对一关联

学习要点：

1. 关联概念
2. 一对关联

本节课我们来开始学习数据库模型的关联概念和一对一关联的方法。

### 一. 关联概念

1. 关联模型，即：两张或以上的表进行一定规则的绑定关联；
2. 比如：一个学生(学生表)对应一张个人信息卡(信息表)，这种就是一对一；
3. 再比如：一篇博文(帖子表)对应多个评论(评论表)，这种就是一对多；
4. 再再比如：一个用户(用户表)对应多个职位(权限表)，而一个职位又可以有多个用户；那么，这种就是多对多关联；
5. 自然，还有更多更复杂的关联，都是基于此的，我们只探讨这三种；
6. 既然是关联，当然会有绑定的概念，当有数据库操作，关联表也会跟着变动；
7. 这就是关联模型的意义；

### 二. 一对关联

1. 一对关联，我们选择两张表来做演示，首先看下两张表的结果对比；

laravel_users @grade (本地) ...			
id	username	password	gender
19	蜡笔小新	123	男
20	路飞	123	男
21	黑崎一护	456	男
24	小明	123	男
25	孙悟饭	123	男
26	孙悟天	123	男
27	樱桃小丸子	123	女
29	孙悟空	123	男
76	李白	123	男
79	辉夜	123	女

laravel_profiles @grade (本地) ...			
id	user_id	hobby	status
1	19	喜欢大姐姐	1
2	20	特爱吃肉	0
3	21	朽木露琪亚	-1
4	24	暗恋小红	2
5	25	毕迪丽	1
6	26	特兰克斯	-1
7	27	琦玉爷爷	1
8	29	有空就修行	1
9	79	会长大人	0

PS：主表主键设为 `id`，关联主键默认就是 `id`，可以默认不写；

PS：附表的外键设置为 `user_id`，即：主表名\_主键，吻合可默认不写；

3. 由于之前 Models 下的 `User.php` 模型代码很多了，改成 `.bak` 后重建；
4. 创建两个 `model`，`User.php` 和 `Profile.php`，并使用命令实现提示；

```
php artisan ide-helper:models
```

```
//User.php , 一对关联 Profile 表
public function profile()
{
    //参数 1 或 : 'App\Http\Models\Profile'
    //参数 2：默认为 user_id , 如不是需要指明
```

```
//参数3：默认id，如不是需要指明  
return $this->hasOne(Profile::class, 'user_id', 'id');  
}
```

5. 注意：`Profile.php` 只要建立空类即可，不需要其它，一对一调用如下：

```
//注意：->profile不要加括号，以属性方式访问  
$profiles = User::find(19)->profile;  
return $profiles;
```

6. 一对一也支持反向关联：定向反向关联；具体通过在 `Profile` 设置即可；

```
//profile.php  
//参数1为主表类  
//参数2，3和正向一致，默认对应可以不写  
public function user()  
{  
    return $this->belongsTo(User::class, 'user_id', 'id');  
}  
  
$users = Profile::find(1)->user;  
return $users;
```

## 26. 模型的一对多关联

学习要点：

### 1. 一对多关联

本节课我们来开始学习数据库模型一对多关联的方法。

## 二. 一对多关联

1. 一对多关联，本质上使用方法和一对一关联类似，内部实现略有不同；
2. 创建另一个模型：book.php，我们看下这个表数据；

id	user_id	title
1	19	《莎士比亚》
2	20	《悲惨世界》
3	21	《西游记》
4	24	《红楼梦》
5	25	《圣经》
6	26	《三个代表》
7	27	《国富论》
8	29	《道德情操论》
9	79	《资本论》
10	19	《热情天堂》
11	19	《完美人生》

PS：这里 user\_id=19 有三个，也就是蜡笔小新有三本书，三个关联数据记录；

3. 正向关联：创建一个空的 book.php，在 User.php 进行对其关联；

```
//正向，一对多关联 Book 表
public function book()
{
    return $this->hasMany(Book::class, 'user_id', 'id');
}

//得到蜡笔小新所有关联的书籍列表
$books = User::find(19)->book;
return $books;
```

4. 获取一对多关联的数据，如果再进行筛选，可以使用下面方法：

```
$books = User::find(19)->book()->where('id', 11)->get();
return $books;
```

5. 一对多的反向关联和一对一反向一样，具体如下：

```
//一对多反向关联
$users = Book::find(1)->user;
return $users;
```

## 27. 模型的多对多关联

学习要点：

### 1. 多对多关联

本节课我们来开始学习数据库模型多对多关联的方法。

## 二. 多对多关联

1. 多对多关联，比前面两种要复杂一些，需要一张中间表，共三张；

users					role_user				
id	username	password	gender	id	type	id	user_id	role_id	details
19	蜡笔小新	123	男	1	超级管理员	1	19	2 (Null)	
20	路飞	123	男	2	评论审核专员	2	20	4 (Null)	
21	黑崎一护	456	男	3	图片监察员	3	21	5 (Null)	
24	小明	123	男	4	帐户处理专员	4	24	1 (Null)	
25	孙悟饭	123	男	5	广告投放专员	5	25	3 (Null)	

- (1) .users: 用户表；
- (2) .roles: 权限表；
- (3) .role\_user: 中间表：默认表名，user\_id, role\_id，默认外键可不指明；

2. 创建权限表：Role.php，留空；在User.php设置多对多关联；

```
//多对多关联
public function role()
{
    return $this->belongsToMany(Role::class, 'role_user', 'user_id', 'role_id');
}
```

PS：参数 2 传中间表名，参数 3, 4 如果是默认值，则可不传；

3. 多对多关联输出，查看用户都拥有哪些权限；

```
$roles = User::find(19)->role;
return $roles;
```

4. 获取权限列表中某一个数据，和一对多操作方法一样，但注意返回的表名称；

```
//注意，多对多这里 role() 返回的是 role_user 表
//可以通过 dd($roles) 查看，所以，where 需要用 role_id 来指明
$roles = User::find(19)->role()->where('role_id', 1)->get();
return $roles;
```

```
//当然，你也可以使用集合的方式去实现筛选
$roles = User::find(19)->role;
return $roles->where('id', 1);
```

5. 多对多的反向关联和其它两种方式也差不多；

```
//反向多对多关联，后面 id 是反的
public function user()
{
    return $this->belongsToMany(User::class, 'role_user', 'role_id',
    'user_id');
}

$users = Role::find(1)->user;
return $users;
```

6. 多对多会生成一个中间字段：**pivot**，里面包含多对多的双 **id**；

7. 如果想要 **pivot** 字段包含更多的中间表字段，可以自行添加，还可以修改字段名；

```
return $this->belongsToMany(Role::class, 'role_user', 'user_id', 'role_id')
    ->withPivot('details', 'id')
    ->as('pivot_name');
```

8. 定义多对多绑定时，可以在绑定方法内筛选数据；

```
$this->belongsToMany(Role::class) ... ->wherePivot('id', 1);
```

PS：还有 **wherePivotIn**，以及派生的四种方法；

PS：除了一对一，一对多，多对多，还有派生的远程一对一，远程一对多，以及多态一对一，多态一对多，多态多对多。这些更多的扩展，暂时不纳入基础的核心课程，防止过于繁杂冗余导致劝退。

## 28. 模型的关联查询

学习要点：

### 1. 关联查询

本节课我们来开始学习数据库模型关联查询的方法。

### 二. 关联查询

1. 前几节课，了解了三种基础的关联模型，并简单的进行查询；
2. 本节课，我们详细的了解更多的查询方案；

```
//下面两种查询是一样的；  
$books = User::find(19)->book;  
$books = User::find(19)->book()->get();  
  
//可以采用 where 筛选或闭包  
$books = User::find(19)->book()  
    ->where('id', 1)->orWhere('id', 11)->get();  
  
$books = User::find(10)->book()->where(function ($query) {  
    $query->where('id', 1)->orWhere('id', 11);  
})->get();
```

3. 使用 `has()`方法，可以查询某些条件下的关联查询数据；

```
//获取存在关联书籍的用户列表(言下之意：至少一本书)  
$users = User::has('book')->get();  
return $users;  
  
//获取存在关联书籍(并超过 3 条)的用户列表  
$users = User::has('book', '>=', 3)->get();  
return $users;
```

4. 使用 `whereHas()`方法，创建闭包查询；

```
//whereHas 闭包用法  
$users = User::whereHas('book', function ($query) {  
    //这里$query 是 book 表，通过 user_id 查询，返回 user 表数据  
    $query->where('user_id', 19);  
})->get();  
return $users;
```

5. 使用 `doesntHave()`方法，即 `has()`的反向操作；

```
//获取不存在关联书籍的用户列表，闭包用法：whereDoesntHave()  
$users = User::doesntHave('book')->get();  
return $users;
```

6. 使用 `withCount()`方法，可以进行关联统计；

```
//关联统计，会自动给一个 book_count 字段
//统计每个用户有多少本书
$users = User::withCount('book')->get();
return $users;

//给多个关系添加统计：profile_count, book_count
$users = User::withCount(['profile', 'book'])->get();
return $users;

//关联统计再结合闭包进行筛选，还可以设置别名
$users = User::withCount(['profile', 'book' => function ($query) {
    //这里限制被统计的记录
    $query->where('user_id', 19);
}])->get();
return $users;
```

## 29. Debugbar 调试器

学习要点：

### 1. 安装使用

本节课我们来开始学习 Debugbar 调试器的安装和简单使用。

### 二. 安装使用

1. 通过 composer 在项目中安装 Debugbar，命令如下：

```
composer require barryvdh/laravel-debugbar
```

2. 生成一个配置文件，给用户配置，可以根据需求进行配置；

```
php artisan vendor:publish --provider="Barryvdh\Debugbar\ServiceProvider"
```

3. 刷新页面，即可看到底部的调试工具，message 信息还需要引入；

```
use Barryvdh\Debugbar\Facade as DebugBar;  
DebugBar::info('信息！');
```

4. 如果想要关闭调试工具，可以设置 config/debugbar.php；

```
'enabled' => env('DEBUGBAR_ENABLED', false),
```

```
//手工开启或关闭  
DebugBar::enable();  
DebugBar::disable();
```

## 30. 模型的预加载

学习要点：

### 1. 预加载

本节课我们来开始学习数据库模型的预加载功能。

#### 一. 预加载

1. 预加载，就是解决关联查询中产生的 N+1 次查询带来的资源消耗
2. 我们要获取所有书籍的作者(或拥有者)，普通查询方案如下：

```
//获取所有书籍列表
$books = Book::all();

//遍历每一本书
foreach ($books as $book) {
    //每一本书的关联用户的姓名
    DebugBar::info($book->user->username);
}
```

PS：通过调试器 Debugbar 中 SQL 语句的分析，发现包含十多条 SQL 语句；

PS：原因是关联查询时，每遍历一次就会执行一遍 SQL 语句，导致性能欠佳；

PS：所谓 N+1 条，就是起初获取全部数据的 1 条和，遍历的 N 条；

3. 使用 `with()` 关键字，进行预载入设置，提前将 SQL 整合；

```
//with 关键字预载入
$books = Book::with('user')->get();

foreach ($books as $book) {
    DebugBar::info($book->user->username);
}
```

PS：此时的 SQL 执行数目为：1+1 条；也支持数组多个关联 `with(['book', 'profile'])`；

PS：预加载也可以设置显示的列；

`//预载入设置指定的列`

```
$books = Book::with('user:id,username')->get();
```

4. 如果每次都必须使用预载入进行关联查询，可以在模型中定义；

```
protected $with = ['user'];
```

PS：此时就可以像最初那样写代码，而不需要使用 `with()` 方法了；

5. 为了演示方便，暂时取消模型`$with`，再看下预载入结合筛选；

```
$books = Book::with(['user' => function ($query) {
    $query->where('id', 19);
}])->get();
```

PS：预载入筛选不可以使用`limit`、`take`方法；

6. 有时，可能会产生逻辑判断是否查询数据，但预加载会提前关联执行；

7. 这样，会导致资源性能的浪费，这时，可以采用延迟预载入；

```
$books = Book::all();
```

```
if (true) {
    $books = $books->load('user'); //load(['user' => function () {}])
    foreach ($books as $book) {
        DebugBar::info($book->user->username);
    }
}
```

8. 使用`loadCount()`方法，可以实现延迟关联统计；

```
$users = User::all();
if (true) {
    return $users->loadCount('book');
}
```

## 31. 模型的关联写入

学习要点：

### 1. 关联写入

本节课我们来开始学习数据库模型的关联写入等操作。

#### 一. 关联写入

- 新增有三种方式，我们一一来看，比如给一个用户增加关联书籍；

```
//先限定用户
$user = User::find(19);
//给这个用户关联的 book 新增一条记录
//user_id 会自动写入 19 , title 自定义
$user->book()->save(new Book(['title' => '《哈利波特》']));
```

- 需要设置批量赋值，我们取消掉，book 表没有时间字段，也要取消自动写入；

```
//取消批量赋值
protected $guarded = [];
//取消自动时间字段
public $timestamps = false;

//批量新增
$user->book()->saveMany([
    new Book(['title' => '《哈利波特》']),
    new Book(['title' => '《指环王》'])
]);
```

- create 和 createMany 只需要插入数组即可完成关联新增；

```
$user->book()->create([
    'title' => '《哈利波特》'
]);

$user->book()->createMany([
    ['title' => '《哈利波特》'],
    ['title' => '《指环王》']
]);
```

PS：还有 findOrNew、firstOrNew、firstOrCreate 和 updateOrCreate 方法；

- 有新增，自然有修改删除，直接使用 delete() 和 update() 方法即可；

```
//关联删除，删除 user_id=19 的书
$user = User::find(99);
$user->book()->delete();
```

```
//关联修改，修改 user_id=19 的书
$user = User::find(99);
$user->book()->update(['title' => '《修改书籍》']);
```

5. 使用 `associate()`方法来修改掉书籍关联的用户，即修改 `user_id`;

```
//修改关联的外键，即：user_id 修改，换用户
$user = User::find(20);
$book = Book::find(11);
$book->user()->associate($user);
$book->save();
```

PS：如果想取消一本书的拥有者，比如将 `user_id` 设置为 `null`，字段要设置可以 `null`；

```
$book = Book::find(11);
$book->user()->dissociate();
$book->save();
```

6. 在搜索书籍的对应用户的时候，空 `null` 字段会导致用户出现 `null` 数据；

7. 我们可以采用空对象默认模型的方式，去解决这个问题；

```
//Book.php
public function user()
{
    return $this->belongsTo(User::class, 'user_id', 'id')
        ->withDefault([
            'id' => 0,
            'username' => '游客用户'
        ]);
}
```

## 32. 多对多的关联写入

学习要点：

### 1. 关联写入

本节课我们来开始学习数据库模型的关联写入等操作。

#### 一. 关联写入

1. 多对多的新增：比如，给用户增加一个角色权限，具体如下：

```
//得到要添加权限的用户
$user = User::find(99);
//得到权限的 id,比如超级管理员
$roleId = 1;
//给辉夜设置成超级管理员
$user->role()->attach($roleId);
```

2. 如果你想给中间表附加 `details` 字段的数据，可以使用第二参数；

```
$user->role()->attach($roleId, ['details' => '喀']);
```

3. 如果想移出某个用户的角色权限，可以使用 `detach()`方法；

```
//删除一个角色权限
$user->role()->detach($roleId);
```

PS：如果不指定中间表 `id`，那么就移出这个用户的所有权限角色；

4. 也支持批量处理，直接用数组传递参数即可；

```
//这里传递的是角色权限表的 ID
$user->role()->attach([1,2,3]); //附加值 1 => ['detail' => 'xxx']
//删除指定的 user_id
$user->role()->detach([1,2,3]);
```

5. 使用 `sync()`方法，可以新增角色权限，且可以判断已存在而不再新增；

```
//同步关联，已存在不在新增
return $user->role()->sync([1,2,3]); //附加值 1 => ['detail' => 'xxx']
```

6. 使用 `updateExistingPivot()`可更新指定 `roleId` 的额外字段；

```
//更新中间表的额外字段
$user->role()->updateExistingPivot($roleId, ['details'=>'喀']);
```

PS：直接使用 `update()`是更新所有；

PS：通过查看源码或 IDE 代码提示的方法，有更多的操作；可自行阅读扩展；

## 33. 请求和依赖注入

学习要点：

1. Request 请求
2. 依赖注入

本节课我们来开始学习 Http 的 Request 请求，然后简单了解下依赖注入。

### 一. Request 请求

1. 创建 UserController 控制器，在方法中注入 Request 对象；

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

class UserController
{
    //方法会自动得到 Request 对象，这个就是注入
    public function index(Request $request)
    {
        //input 的方法可以获取 http 请求的数据
        //localhost:8000/user?name=Mr.Lee
        return $request->input('name');
    }
}
```

3. 使用 all() 方法，可以获取所有 url 传递过来的参数；

```
//http://localhost:8000/user?name=Mr.Lee&gender=男&age=100
return $request->all();
```

4. 路由自带参数时，可以通过第二，第三参数接受；

```
Route::get('/user/{id}/{uid}', 'UserController@index');
public function index(Request $request, $id, $uid)
```

5. 路由区域，也支持闭包注入 Request；

```
Route::get('/user', function (Request $request) {
    return $request->all();
});
```

6. 使用 path() 方法，可以得到当前的 uri 路径；

```
return $request->path();
```

7. 使用 is() 方法，可以判断当前的 uri 是否匹配；

```
//是否是 user/* 的 uri，返回布尔值
return $request->is('user/*');
```

8. 使用 `url()` 和 `fullUrl()` 得到 `url` 地址，带参数和不带参数区别；

```
return $request->url();
return $request->fullUrl();
```

9. 使用 `isMethod()` 来判断 HTTP 请求的方式，`get`、`post` 等等；

```
return $request->isMethod('post');
```

## 二. 依赖注入

1. 我们发现在控制器方法，可以直接拿到 `Request` 对象，而不需要 `new`；
2. 这种方法是系统的“服务容器”给你自动注入的；
3. 比如，我们想使用另一个控制器的方法，那原始做法如下：

```
$task = new TaskController();
return $task->read(10);
```

4. 直接使用依赖注入，也可以直接使用；

```
public function index(Request $request, TaskController $task)
{
    return $task->read(10);
}
```

## 34. 请求的常用方法

学习要点：

### 1. 常用方法

本节课我们来开始学习 `Request` 对象的常用方法。

#### 一. 常用方法

- 上一节了解过 `all()` 获取全部, `input()` 获取指定, 还有其它方式;

```
//参数2为默认值  
$request->input('name', 'no name');
```

```
//空参数和all()效果一样  
$request->input();
```

```
//动态方式获取  
$request->name;
```

- 数组的接受方式, 如下:

```
<form action="/user/10" method="get">  
    <input type="checkbox" name="select[][a]" value="1">  
    <input type="checkbox" name="select[][b]" value="2">  
    <input type="checkbox" name="select[][c]" value="3">  
    <button type="submit">发送</button>  
</form>  
  
return $request->input('select.1.b');
```

PS: 如果是 JSON 数据, 也是这么弄;

- `Request` 对象还有一些实用方法, 具体如下:

```
//返回布尔值  
$request->boolean('name');
```

```
//返回IP  
$request->ip();
```

```
//只接受固定参数  
$request->only(['age', 'gender']);
```

```
//排除不要的参数  
$request->except(['name']);
```

```
//判断参数是否存在  
return $request->has('name');  
  
//判断参数是否全部存在  
return $request->has(['name', 'age']);  
  
//判断参数只有一个存在，就返回 true  
return $request->hasAny(['name', 'age']);  
  
//判断参数存在，并且不为空  
return $request->filled('name');  
  
//判断参数不存在(为空也不行)  
return $request->missing('name');
```

4. `request()`助手函数，使用如下：

```
return request()->input();
```

## 35.生成 URL

学习要点：

### 1.生成 URL

本节课我们来开始学习 URL 和 url()助手函数，用于生成 url 地址。

#### 一. 生成 URL

1. 框架提供了 url()助手函数，方便我们生成各种想要 url 地址；

```
//生成指定的 url  
$user = User::find(19);  
return url('/user/' . $user->id);
```

2. 如果 url()不给参数，可以当作对象执行更多的方法；

```
//得到当前 url , 不带参数  
return url()->current();  
//得到当前 url , 带参数  
return url()->full();  
//得到上一个 url  
return url()->previous();           //URL::previous()
```

3. 使用 route()方法，生成命名路由的 url，貌似讲过？

```
Route::any('/url/{id}', 'UserController@url')->name('url.id');  
return route('url.id', ['id' => 5]);
```

4. 也可以直接使用控制器，也可以返回 url；

```
//使用控制器返回 url  
return action('UserController@index', ['id' => 5]);
```

5. 生成一个签名 URL，在 URL 后面追加一个哈希签名字符串，用于验证；

```
return url()->signedRoute('url.id', ['id' => 5]);  
  
//验证哈希签名  
return request()->hasValidSignature();
```

## 36. Cookie

学习要点：

### 1. 使用 Cookie

本节课我们来开始学习 **Cookie** 的使用方法。

#### 一. 使用 **Cookie**

- 首先，获取 **Cookie** 有两种方法，具体如下：

```
//注意 Laravel 中 cookie 都是加密的，原生 cookie 只能获取加密信息  
//使用 request()->cookie 获取解密后的 cookie 信息  
return request()->cookie('laravel_session');  
  
//使用 Cookie::也可以获取，引入 Illuminate\Support\Facades\Cookie;  
return Cookie::get('laravel_session');
```

- 使用 **response()** 方法，可以创建 **cookie**；

```
//response()方法写入一个 cookie  
//参数 3，是过期时间，已分钟为单位  
//这里必须有 return，否则无法写入  
return response('Hello Cookie')->cookie('name', 'Mr.Lee', 10);
```

- 使用 **Cookie::queue()** 方法来写入 **cookie**；

```
//推荐这个，清爽很多  
Cookie::queue('age', 100, 10);
```

- 使用助手函数 **cookie()** 来创建 **cookie** 实例，然后再写入，更加灵活；

```
//助手函数，创建一个实例，让写入可以更加灵活  
$cookie = cookie('gender', '男', 10);  
Cookie::queue($cookie);
```

```
//完整版，后面四种：路径，域名，https，仅 http  
cookie($name, $value, $minutes, $path, $domain, $secure, $httpOnly)
```

- 默认 **cookie** 是加密存放的，如果想某个 **cookie** 不加密，在中间件文件夹设置；

`Http\Middleware/middlewareEncryptCookies.php`

```
protected $except = [  
    //  
    'name'  
];
```

## 37.Session

学习要点：

### 1. 使用 Session

本节课我们来开始学习 Session 的使用方法。

#### 一. 使用 Session

- 首先，启动 Web 后，默认会有 session，通过下面代码获取所有；

```
//获取所有 session  
return request()->session()->all();
```

- 使用 get() 获取某一个 session；

```
//获取其中一个 session  
return request()->session()->get('_token');  
  
//参数 2，闭包设置默认值  
return request()->session()->get('name', function () {  
    return 'no session name';  
});
```

- 当然，也可以采用 Session::get() 来获取 session；

```
return Session::get('_token');
```

- 助手函数 session()，可以获取并可以设置默认值；

```
//获取 session 值  
return session('_token');  
//获取 session 值并设置默认值  
return session('name', 'no session name');
```

- 判断是否存在 session 有两种方案，都支持 request() 方式；

```
return Session::has('name');           //判断是否存在且不为 null  
return Session::exists('name');        //判断是否存在，即使是 null
```

- 也可以使用助手函数传递数组的方式或 put() 方法，来存储 session 值；

```
//设置 session 值  
session(['name' => 'Mr.Lee']);  
  
//也支持 request() 存储  
Session::put('name', 'MrWang');
```

7. 使用 `push()`方法，可以存储数组，支持 `request()`方式；

```
//session 数组方式
Session::push('info.name', 'Mr.Lee');
Session::push('info.name', 'Mr.Wang');
Session::push('info.name', 'Mr.Zhang');
return Session::get('info');
```

8. 使用 `flash()`方法，获取后自动删除，支持 `request()`方式；

```
//存储的 session 只能被获取一次，然后自动删除，flash 也称为闪存数据
Session::flash('name', 'Mr.Lee');
```

9. 如果使用闪存数据，本次请求不要立刻自行删除，可以使用 `reflash()`；

```
//本次请求获取，不要删除数据，给下一次请求时再自行删除，这是保存所有闪存数据
Session::reflash();           //Session::keep(['name']);保存单独的删除数据
return Session::get('name');
```

10. 如果 `forget()`可以删除一条或多条 `session` 数据，支持 `request()`方式；

```
//删除一条数据
Session::forget('name');      //Session::forget(['name'])
return Session::get('name');

//删除一条数据，并返回
Session::pull('info');
//删除所有数据
Session::flush();
```

11. 如果 `regenerate()`可以重新生成 `SessionID`；

```
//重新生成 SessionID
Session::regenerate();
//获取 SessionID
return Cookie::get('laravel_session');
```

## 38. 理解中间件

学习要点：

### 1. 路由中间件

本节课我们来开始学习中间件的使用方法。

#### 一. 路由中间件

1. 什么是中间件？中间件就是当程序接收 HTTP 请求时，拦截后进行过滤和处理；
2. 比如当用户登录时，可以通过中间件进行验证比对，错误后让其跳转到登录页面；
3. 框架系统自带了一些中间件，比如之前 CSRF 令牌保护，就是中间件实现的；
4. 如果创建一个自定义的中间件？可以通过一句命令创建一个 `check` 中间件；

```
php artisan make:middleware Check
```

5. 实现一个简单的登录身份验证的效果，首先创建一个 `Login` 控制器；

```
class LoginController  
{  
    public function index()  
    {  
        echo '管理员，您好！';  
    }  
  
    public function login()  
    {  
        echo '登录失败！';  
    }  
}  
  
Route::get('/admin','LoginController@index');  
Route::get('/login','LoginController@login');
```

6. 我们要求，当 `Http` 接受 `id=1` 的情况下，它就是管理员，跳转 `index` 否则 `login`；
7. 此时，我们打开一开始创建好的 `Check` 中间件文件，了解一下结构；

```
//固定方法，固定格式  
public function handle($request, Closure $next)  
{  
    //这里编写验证跳转方法  
    if ($request->get('id') != 1) {  
        return redirect(url('/login'));  
    }  
    //固定返回格式，让其继续往下执行  
    return $next($request);  
}
```

8. 下面，我们要对中间件进行注册，才可以使用，这个中间件适合在路由上使用；
9. 所以，我们打开 `Http/Kernel.php` 文件，在路由配置中间件的区域进行注册；

```
protected $routeMiddleware = [  
    'check' => \App\Http\Middleware\Check::class,  
]
```

10. 然后，我们在登录的路由上执行中间件即可完成登录验证；

```
Route::get('/admin', 'LoginController@index')->middleware('check');
```

11. 这种中间件，属于前置中间件，也就是先拦截 `Http` 请求，再执行主体代码；
12. 另一种，就是后置中间件，也就是先执行主体代码，再拦截处理；

```
//固定方法，固定格式  
public function handle($request, Closure $next)  
{  
    //先执行主体代码  
    $response = $next($request);  
  
    //再进行拦截 Http 请求处理  
    echo '我是后置中间件';  
  
    //固定格式返回  
    return $response;  
}
```

## 39. 中间件进阶

学习要点：

### 1. 中间件进阶

本节课我们来开始学习中间件的更多的其它方法。

#### 一. 中间件进阶

- 在路由中间件，我们可以设置多个中间件，进行调用；

```
->middleware('check', 'auth');
```

- 如果你没有在配置中注册中间件，可以采用完整的类名来进行调用；

```
->middleware(\App\Http\Middleware\Check::class);
```

- 全局中间件，直接配置在\$middleware 属性即可，每次执行都必然调用；

```
php artisan make:middleware Every;
```

```
protected $middleware = [  
    \App\Http\Middleware\Every::class,  
]
```

- 中间件的核心方法可以有第三个参数，可以在控制器调用时传递；

```
public function handle($request, Closure $next, $param)
```

```
->middleware('check:abc');
```

- 中间件组，如果有一些需要固定调用多个中间件，我们可以将它群组；

```
protected $middlewareGroups = [  
    'mymd' => ['check' =>\App\Http\Middleware\Check::class],  
];
```

- 中间件的terminate()方法，可以在中间件响应完之后(return \$next)再调用；

```
public function terminate($request, $response)  
{  
    echo '<br>Http 响应完毕之后再调用我';  
}
```

- 中间件也可以在控制器的构造方法里调用，这里注意错误跳转会死循环；

```
public function __construct()  
{  
    $this->middleware('check:abc');  
}
```

## 40. Blade 模板入门

学习要点：

- 1. Blade 简介
- 2. 模板基础功能

本节课我们来开始学习 Blade 模板的功能以及模板的一些基础功能。

### 一. Blade 简介

- 1. Blade 是 Laravel 内置的模板引擎，之前课程中已经知道如何创建；
- 2. 我们用 Task 控制器的 user() 方法来测试，使用 view() 方法来引入模板；

```
public function user()  
{  
    return view('user');  
}
```

- 3. 创建 user.blade.php 模板文件，模板支持原生 PHP 开发；

```
<?php echo 'Blade'.(1 + 1)?>
```

- 4. 和其它模板引擎一样，模板文件被执行后会缓存，而编辑修改后会自动重新缓存；
- 5. 我们将模板文件存放在 resources/views 目录里，后缀为：.blade.php；

### 二. 模板基础功能

- 1. 之前的课程中，简单使用过，我们再复习一下，在控制器可以给模板传递变量；

```
//参数 2 数组，声明模板变量  
return view('user', [  
    'name' => 'Mr.Lee' //{{ $name }} 模板变量  
]);  
  
//facade 方法  
return View::make('user', [  
    'name' => 'Mr.Lee'  
]);
```

- 2. 如果模板页面内容极其简单，也可以直接通过路由加载模板，绕过控制器；

```
Route::get('/task/user', function () {  
    return view('user', [  
        'name' => 'Mr.Lee'  
    ]);  
});
```

3. 如果模板根目录下建立子目录，调用方法用点符号作为路径格式；

```
//resources/views/admin/index.blade.php  
return view('admin.index');
```

4. 有时可能有判断模板文件是否存在需求，可以使用 `exists()` 方法；

```
//判断模板文件是否存在  
return view()->exists('admin.index');  
//facade 方法  
return View::exists('admin.index');
```

5. 也可以使用 `first()` 方法，加载存在于数组中的第一个模板；

```
//加载存在于数组中的第一个模板  
return view()->first(['abc', 'user', 'def'], [  
    'name' => 'Mr.Lee'  
]);  
//facade 方法  
return View::first(['abc', 'user', 'def']...;
```

## 41. 模版的流程控制

学习要点：

1. 条件判断
2. 循环遍历

本节课我们来开始学习 Blade 模板的一些控制流程：条件判断和循环遍历。

### 一. 条件判断

1. 在模版中我们可以使用@if @else @elseif @endif 来设置条件判断；

```
return view('user', [  
    'num' => 20  
]);
```

```
{{--单一判断--}}  
@if($num > 10)  
    num 大于 10  
@endif
```

```
{{--带 else 判断--}}  
@if($num > 10)  
    num 大于 10  
@else  
    num 小于 10  
@endif
```

```
{{--带 elseif 判断--}}  
@if($num > 10)  
    num 大于 10  
@elseif($num > 5)  
    num 大于 5  
@else  
    num 小于 5  
@endif
```

2. @unless @endunless 相当于@if 取反的操作，可通过编译文件参看；

```
@unless($num > 10)  
    num 小于 10  
@endunless
```

3. @isset 判断变量是否存在 @empty 判断变量是否为空；

```
@isset($name)  
    变量存在
```

```

@endisset

@empty($name)
    变量为空
@endempty

4. @switch 实现条件分支判断，包含@case @break @default;
@switch($num)
    @case(1)
        1
        @break
    @case(4)
        4
        @break
    @default
        不存在
@endswitch

```

## 二. 循环遍历

1. @for 循环，适合数值的循环；

```

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

```

2. @foreach 适合对象的变量循环；

```

@foreach($obj as $user)
    {{$user->username}}
@endforeach

```

3. @continue 可以跳出当且迭代，@break 跳出循环；

```

@foreach($obj as $user)
    @if ($user->username == '樱桃小丸子')
        @continue //@break
    @endif
    {{$user->username}}
@endforeach

```

PS: 变体写法: `@continue($user->username == '樱桃小丸子')`

4. @while 判断循环；

```

@while($num > 0)
    while 循环
    {{$num--}}
@endwhile

```

5. 在循环体内，会有一个@loop 变量，帮助我们处理各种问题；

```
@foreach($obj as $user)
```

```
@if($loop->first)
```

[起始数据之前]

```
@endif
```

```
@if($loop->last)
```

[末尾数据之前]

```
@endif
```

```
 {{$user->username}} --
```

```
@endforeach
```

属性	说明
\$loop->index	当前迭代的索引（从 0 开始）
\$loop->iteration	当前循环迭代（从 1 开始）
\$loop->remaining	循环中剩余迭代的个数
\$loop->count	被循环的数组元素个数
\$loop->first	是否为循环的第一次迭代
\$loop->last	是否为循环的最后一次迭代
\$loop->even	是否为循环中的偶数次迭代
\$loop->odd	是否为循环中的奇数次迭代
\$loop->depth	当前循环的嵌套深度
\$loop->parent	嵌套循环中的父循环的循环变量

6. PHP 注释和原生的另一种方案@php;

```
@php
```

```
 echo 123;
```

```
@endphp
```

## 42. 模板的继承布局

学习要点：

1. 继承布局
2. 其它技巧

本节课我们来开始学习 Blade 模板的定义和继承布局的方法。

### 一. 继承布局

1. 为了重复的页面代码更好的管理，首先要定义需要被继承的代码；
2. 比如，我们在 `views` 目录下建立 `public` 目录建立父模板 `base.blade.php`；
3. 这个 `base` 模板主要存放主体页面的，为了更好的理解，关闭 `debug` 调试；
4. 然后在正常的模板区域建立一个子模板 `index.blade.php`，并继承 `base`；  
`@extends('public.base')`

5. 父模板通过`@yield` 设置一个可替换的变量，子模板通过`@section` 改变变量；

```
{{--@yield 参数1 变量名, 参数2 默认值--}}
<title>Base -- @yield('title', 'no title')</title>
```

```
{{--改变父模板的变量值--}}
@section('title', '首页')
```

```
<div class="container">
    主区域
    @yield('main')
</div>
```

```
@section('main')
    <p>子区域</p>
@endsection
```

6. 当然，也可以设置子模板继承父模板的部分内容，具体如下：

```
{{--父模板--}}
@section('sidebar')
<nav>
    <ol>
        <li><a href="#">导航</a></li>
    </ol>
</nav>
@show
```

```
{{--子模板--}}
@section('sidebar')
```

```
@parent  
导航  
@endsection
```

## 二. 其它技巧

- 默认变量会被自动转义，比如特殊符号&，如果不想被转义可以用如下格式：

```
{!! $name !!}
```

- 可以使用@json 直接将数组转换成 json 格式，参数 2 格式化，和原生一样；

```
@json($list)  
@json($list, JSON_PRETTY_PRINT)
```

- 在 JavaScript 区域，可能也有{{name}}这种模式的操作，和模版变量冲突；

```
{{--加上@即可防止解析--}}  
@{{ name }}
```

```
{{--大量 JS 时，用范围方式--}}  
@verbatim  
{{ name }}  
@endverbatim
```

- 如果想让所有模版共享一个变量，在 Providers\AppServiceProvider.php；

```
public function boot()  
{  
    view()->share('title', 'Laravel');  
}
```

```
<title>{{$title}}</title>
```

PS：补遗一下：给模版传递变量除了参数 2 的数组，也可以使用 with()方法；

```
view()->with('name', '&');
```

## 43. 表单快速验证

学习要点：

### 1. 快速验证

本节课我们来开始学习表单提交之后的快速验证功能。

#### 一. 快速验证

1. 首先，创建两个路由：一个表单，一个处理表单；

```
//表单页
Route::get('/task/form', 'TaskController@form');
//接收页
Route::post('/task/receive', 'TaskController@receive');
```

2. 表单，我们先采用用户名和密码来演示验证；

```
<form action="/task/receive" method="post">
    @csrf
    用户名：<input type="text" name="username">
    密码：<input type="password" name="password">
    <button type="submit">提交</button>
</form>
```

3. 表单处理页，通过 `Request` 对象来实现字段验证，注释写了执行流程；

```
public function receive(Request $request)
{
    //验证请求方式
    //左边为验证字段，右边为验证规则，每个规则竖线隔开
    //比如 required 不得为空，min 不等小于，max 不得大于
    $request->validate([
        'username'      => 'required|min:2|max:10',
        'password'      => 'required|min:6',
    ]);

    //只有通过才能执行下面的语句，否则返回表单页
    return '恭喜验证通过！';
}
```

4. 如果验证没有被通过，会自动重定向到提交页，并将错误信息存储到 `session` 中；

```
@if ($errors->any())
    <div>
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
        @endforeach
    </ul>
</div>
```

```
    @endforeach
  </ul>
</div>
@endif
```

PS：内部操作是通过中间件：Illuminate\View\Middleware\ShareErrorsFromSession；

5. 也可以单独设置错误信息，使用@error 指令；

```
@error('username')
  <div>{{ $message }} : 用户名非法！</div>
@enderror

@error('password')
  <div>{{ $message }} : 密码非法！</div>
@enderror
```

6. 关于规则，还有其它写法，具体如下：

```
//数组方式验证
$request->validate([
  'username'      => ['required', 'min:2', 'max:10'],
  'password'      => ['required', 'min:6']
]);
```

## 44. 验证类的使用

学习要点：

### 1. 创建验证类

本节课我们来开始学习验证类的创建和使用方法。

#### 一. 创建验证类

1. 上节课，使用的快速验证，适合小规模单独的验证；
2. 也创建一个验证类，把验证规则写到验证类里，然后调用；

```
php artisan make:request Form
```

3. 验证类 `Http\Request\Form.php` 创建好之后，会提供两个固定方法；

```
public function authorize()  
{  
    //默认 false，为关闭授权，关闭状态会 403  
    //比如判断这个用户是否有操作权限  
    return true;  
}  
  
public function rules()  
{  
    return [  
        //规则  
        'username'      => 'required|min:2|max:10',  
        'password'       => 'required|min:6',  
    ];  
}
```

4. 在控制器端，直接使用验证类进行验证即可，其它和快速验证一样；

```
//通过验证类验证  
$request->validated();
```

5. 默认错误提示是英文的，如果想设置成中文提示，可在验证类创建 `message()`；

```
//验证器类方法  
public function messages()  
{  
    return [  
        'username.required'      => '用户名不得为空~',  
        'username.min'           => '用户名不得小于 2 位~',  
        'username.max'           => '用户名不得大于 10 位~',  
  
        'password.required'     => '密码不得为空~',
```

```
'password.min'      => '密码不得小于 6 位~',  
];  
}
```

6. 也可以自定义属性名，而其它采用默认提示；

```
public function attributes()  
{  
    return [  
        'username' => '用户名',  
    ];  
}
```

PS：提示语言包在：resources\lang\en\validation.php

7. 在请求验证之前，修改提交的请求数据；

```
protected function prepareForValidation()  
{  
    $this->merge([  
        'username' => 'Mr.Lee'  
    ]);  
}
```

## 45. 创建手动验证

学习要点：

### 1. 手动验证

本节课我们来开始学习手动验证的创建方法。

#### 一. 手动验证

1. 手动验证，和第一节快速验证是一样的，都是在控制器验证；
2. 只不过，快速验证是全自动化的，无法局部控制更多的细节；

```
//快速验证方式回顾
$validateData = $request->validate([
    'username'      => 'required|min:2|max:10',
    'password'      => 'required|min:6',
]);

//验证通过后得到提交的值
dd($validateData);
```

3. 创建手动验证，具体如下：

```
//手动创建验证器
$validator = Validator::make($request->post(), [
    'username'      => 'required|min:2|max:10',
    'password'      => 'required|min:6',
]);

//如果没有通过的话
if ($validator->fails()) {
    //跳转后，并将错误保存，还可以返回上一次填的信息
    return redirect('/task/form')->withErrors($validator)->withInput();
}
```

用户名：`<input type="text" name="username" value="{{old('username')}}">`

PS：快速验证，也支持 `old` 记忆上一次表单数据的功能；

4. 如果有多个表单，那可以设置`$errors` 的多个错误包，有两种方法：

```
$validator = Validator::make($request->post(), [
    'username'      => 'required|min:2|max:10',
    'password'      => 'required|min:6',
])->validateWithBag('post'); //设置快速验证和手动验证
```

```
if ($validator->fails()) {  
    //适合手动验证  
    return redirect('/task/form')->withErrors($validator,  
        'post')->withInput();  
}  
  
{  
{{ $errors->post }}  
}
```

5. 我们也可以在验证完毕之前，通过附加回调添加更多的自定义错误信息；

```
//验证钩子，  
$validator->after(function ($validator) {  
    $validator->errors()->add('info', '隐藏字段 info 的值不存在');  
});
```

6. 关于\$errors 对象，除了 any() 判断， all() 获取全部，还有一些别的方法；

```
{  
{{$errors->first('username')}}  
  
{  
{{$errors->has('username')}}
```

## 46. 验证规则大全

学习要点：

1. 验证规则
2. 其它用法

本节课我们来开始学习验证规则，挑出常用典型的理解一遍，其它自参。

### 一. 验证规则

1. 验证规则，提供了大量实用封装好的规则，我们只需要进行调用即可；
2. 基础课程中，我们使用了 `required`、`min` 和 `max` 这三种比较简单的；
3. 下面，我们挑出比较典型常用的，演示一遍，具体大全网址如下：

<https://learnku.com/docs/laravel/7.x/validation/7467#available-validation-rules>

#### 可用验证规则

以下是所有可用验证规则及其功能的列表：

Accepted	E-Mail	Not Regex
Active URL	Ends With	Nullable
After (Date)	Exclude If	Numeric
After Or Equal (Date)	Exclude Unless	Password
Alpha	Exists (Database)	Present
Alpha Dash	File	Regular Expression
Alpha Numeric	Filled	Required
Array	Greater Than	Required If
Bail	Greater Than Or Equal	Required Unless
Before (Date)	Image (File)	Required With
Before Or Equal (Date)	In	Required With All
Between	In Array	Required Without
Boolean	Integer	Required Without All
Confirmed	IP Address	Same
Date	JSON	Size
Date Equals	Less Than	Sometimes
Date Format	Less Than Or Equal	Starts With
Different	Max	String
Digits	MIME Types	Timezone
Digits Between	MIME Type By File Extension	Unique (Database)
Dimensions (Image Files)	Min	URL
Distinct	Not In	UUID

PS：约 66 个；

(1) `.alpha_dash`: 字段必须由字母、数字破折号(-)和下划线(\_)构成；  
`'username' => 'required|alpha_dash',`

(2) `.between`: 字段长度必须在指定的区间之间；  
`'username' => 'required|between:2,6',`

(3) `.size:6:` 字段长度固定在某个值;

```
'username' => 'required|size:6',
```

(4) `.email:` 字段必须符合 `email` 格式;

```
'username' => 'required|email',
```

(5) `.unique:users:` 字段值是唯一的，通过指定数据表字段检查;

```
'username' => 'required|unique:users',
```

(6) `.confirmed:` 这个基本用于密码和密码确认，固定了字段名称;

```
'password' => 'required|min:6|confirmed',
```

```
<input type="password" name="password_confirmation">
```

(7) `different:password:` 字段和另一个字段不可以相同的验证;

```
'username' => 'required|different:password',
```

(8) `.ip/json:` 字段是否是 `ip` 地址/字段是否是 `json` 数据;

```
'username' => 'required|ip',
```

```
'username' => 'required|json',
```

## 二. 其它用法

1. 验证方法还提供了 `Rule` 类提供更加流程的验证构造，比如 `in`、`not_in`;

```
'username' => 'required|in:tom,jack,lusy',
```

```
'username' => [
    'required',
    Rule::in(['tom', 'jack', 'lusy'])
],
```

//如果只验证一个规则，不需要数组，如果规则名有下划线，首字母大写

```
'username' => Rule::notIn(['tom', 'jack', 'lusy']),
```

2. `Rule` 类还可以结合 `where` 来限制验证范围等操作;

//验证是否重复

```
'username' => Rule::unique('users'),
```

```
'username' => Rule::unique('users')->where(function ($query) {
    $query->where('id', 20);
}),
```

## 47. 数据分页

学习要点：

1. 创建分页
2. 更多方法

本节课我们来开始学习分页的创建方法，包括数据库和模型方式。

### 一. 创建分页

1. 数据库 DB 方式的分页，使用 `paginate()` 方法，具体如下：

```
$users = DB::table('users')->paginate(5);

return view('data', [
    'list'      => $users
]);

@foreach($list as $user) ...
```

PS：传递 5，表示每页显示 5 条，地址栏默认听过`?page=2`，来切换显示页数；

2. 模型创建分页，和 DB 方法一致，具体如下：

```
$users = User::paginate(5);
```

3. 分页按钮直接使用 `links()` 方法即可，它继承了 Bootstrap 样式；

```
 {{$list->links()}}
```

4. 使用 `withPath()` 方法，更改路由地址；

```
$users->withPath('/users/list');
```

5. 如果你想带指定参数，可以使用 `appends()` 方法；

```
 {{$list->appends(['sort'=>'id'])->links()}}
```

6. 保存所有查询参数，可以使用 `withQueryString()` 方法；

```
 {{$list->withQueryString()->links()}}
```

7. 使用 `fragment()` 方法给 URL 地址#符号；

```
 {{$list->fragment('element')->links()}}
```

8. 大量分页中...省略分页数量，使用 `onEachSize()` 方法可设置外侧数量，默认 3；

```
 {{$list->onEachSide(1)->links()}}
```

## 二. 更多方法

1. 还有大量方法可供使用，具体如下：

方法	描述
<code>\$results-&gt;count()</code>	获取当前页数据的数量
<code>\$results-&gt;currentPage()</code>	获取当前页页码
<code>\$results-&gt;getUrlRange(\$start, \$end)</code>	创建分页 URL 的范围
<code>\$results-&gt;hasPages()</code>	是否有多页
<code>\$results-&gt;hasMorePages()</code>	是否有更多页
<code>\$results-&gt;firstItem()</code>	获取结果集中第一条数据的结果编号
<code>\$results-&gt;getOptions()</code>	获取分页器选项
<code>\$results-&gt;items()</code>	获取当前页的所有项
<code>\$results-&gt;lastItem()</code>	获取结果集中最后一条数据的结果编号
<code>\$results-&gt;lastPage()</code>	获取最后一页的页码。 (在 <code>simplePaginate</code> 无效)
<code>\$results-&gt;nextPageUrl()</code>	获取下一页的 URL
<code>\$results-&gt;onFirstPage()</code>	当前页是否为第一页
<code>\$results-&gt;perPage()</code>	每页的数据条数
<code>\$results-&gt;previousPageUrl()</code>	获取前一页的 URL
<code>\$results-&gt;total()</code>	数据总数 (在 <code>simplePaginate</code> 无效)
<code>\$results-&gt;url(\$page)</code>	获取指定页的 URL
<code>\$results-&gt;getPageName()</code>	获取分页的查询字符串变量
<code>\$results-&gt;setPageName(\$name)</code>	设置分页的查询字符串变量