翻译整理: leolovenet@gmail.com
Blog: http://leolovenet.com
官方文档: http://laravel.com/docs/quick

前言: 整理翻译这个文档,实属一个意外,一开始要做一个小项目,就看了一天的文档, 但是, 到后来发现那个小项目实在是太小了, 使用这么大的框架有点大材小用,有加上自己对框架也不太熟悉, 最终是没有使用 Laravel 的. 不过这激起了我学习这个框架的兴趣. 最终产生了本文. 本文所有的翻译整理全是为了自我学习, 肯定有翻译不准确, 甚至错白字存在, 实在抱歉. 以后慢慢修改吧.

本文没有讲解安装步骤,假设你已经安装成功 Laravel 了. 截至目前, Laravel 的**最新稳定版本为 4.1**

Artisan

Laravel 有一个命令行接口,叫做 Artisan . 它提供了一些在开放网站过程中非常有用的命令.是通过 Symfony Console component 来驱动的. 可以在项目 Root 目录下运行.

用法:

(1) 所有可用的命令: `php artisan list`
(2) 访问每个命令的帮助信息: `php artisan help migrate`
(3) 在特定配置环境下运行: `php artisan migrate --env=local`
(4) 现实当前 Laravel 版本: `php artisan --version`

Blade

Blade(刀锋), 是 Laravel 的模板系统(templating system). Blade 运行很快, 因为它使用简单的正则表达式把你的模板编译成为纯 PHP, 同时 Blade 又提供了像模板继承, 以及 php 的 if 和 for 类似的语法结构控制语句. 更多的信息请查看 Blade 的详细文档

Migration

当我们想创建一个数据库表格来存放我们的数据的时候, 可以是用Laravel的 migration (迁移)系统. Laravel 的 migration 系统可以直观的定义对数据库的修改, 并且很容易跟团队的其他人员共享.

使用 migration 的第一步,需要我们先配置对数据库的连接信息. 框架把所有对数据库的配置信息都存放在了 app/config/database.php 文件中. 默认行况下 Laravel 是使用 MySQL ,你需要在配置文件中指定好对数据库的连接信息. 如果你更愿意是用 sqlite 替代 MySQL 的话, 仅需要更改配置文件的 `driver` 选项为

`sqlite` ,随后框架将使用包含在 `app/database` 目录下的 SQLite 数据库作为存储媒介.

第二步,创建 migration .我们将要使用上面介绍的 Artisan 命令行. 进入终端, 在项目的根目录下运行下面的命令:

```
php artisan migrate:make create_users_table
```

这将会在 `app/database/migrations` 文件夹内生成 migration 文件, 文件中有一个由两个方法 `up` 和 `down` 组成的一个类(类名为 `CreateUsersTable`). 在 `up` 方法中,应该包含对数据库表格渴望的改变, 在 `down` 方法中应该做出相反的动作.

让我们定义一个像下面的这样的 migration:

```
public function up()
{
    Schema::create('users', function($table)     //创建一个 users 表格.
    {
        $table->increments('id');                //在 users 表中添加一个子增长的
 id 列
        $table->string('email')->unique();       //在 users 表中添加一个只能是唯
一的 email 字符列
        $table->string('name');                  //在 users 表中添加一个 name
字符列
        $table->timestamps();                    //在 users 表中添加一个时间戳的列

    });
}

public function down()
{
    Schema::drop('users');                       //删除这个表格
}
```

一切准备完毕,接下来,我们就可以在终端中使用 `migrate` 命令来运行我们刚刚设计好了的 miagration. 在你项目的根目录下面简单的执行下面的命令:

```
php artisan migrate
```

如果你希望做回滚 migration 的话, 你就需要执行 `migrate:rollback` 命令. 现在我们已经有了我们的数据库表格,接下来让我们向里面存放点数据.

Eloquent ORM

Laravel拥有一个超级 ORM 系统叫做 Eloquent
首先, 让我们来定义一个模型. 一个 Eloquent 模型一个用来查询与它相连的表格, 也可以用来代表表格里的一行. 一会儿你就会明白了. Models 一般是存储在 `app/models` 目录下面的. 让我们在这个目录下定义一个 `User.php` 模型文件,像这样

```
class User extends Eloquent {}
```

注意一下,这里我们并没有告诉 Eloquent 我们要使用那个表格. Eloquent 有各种自己的约定, 其中之一就是使用模型名字的复数形式作为模型数据库中表格的名字. 这些全是风俗约定而已.

你可以使用你最喜欢的数据库管理工具,在 users 表格中插入几行数据,我们接下来将使用 Eloquent 来检索他们,并把他们交给视图(View)来展示.
在 `app/routes.php` 中添加下面的配置:

```
Route::get('users', function()
{
    $users = User::all();

    return View::make('users')->with('users', $users);
});
```

User 模型的 `all` 方法检索出了所有在 users 表格中的行, 然后我们使用 `with` 方法传递这些记录到视图(view)中.
`with` 方法接受一个键和值,使用他们在 view 中可见. 我们可以在 view 中这样来展示他们,在 `app/view` 目录下,创建两个文件, layout.blade.php 和 users.blade.php

接下来我们看一下使用 刀锋(Blade) 的语法定义的视图(View):
layout.blade.php文件内容为:

```
<html>
    <body>
        <h1>Laravel Quickstart</h1>

        @yield('content')
    </body>
</html>
```

users.blade.php文件内容为:

```
@extends('layout')

@section('content')
    @foreach($users as $user)
        <p>{{ $user-> name }}</p>
```

```
    @endforeach
@stop
```

你可能会比较惊奇, 怎么没有看见 `echo` 语句呢? 当使用 Blade 的时候, 你可以使用 **{{ }}** 来显示数据. 这个很容易做到. 现在你可以打开你的浏览器观看 http://yoursite.com/user 来观看展示的结果了.

Laravel项目目录结构

```
├── app                 // 网站APP 代码目录, 自己的代码主要存放在这个目录下,主要是 MVC 结构
│   ├── commands
│   ├── config          // 存放所有laravel有关的配置文件,APP的主配置文件app.php就在存放在该目录下面.里面定义了 Service Providers
│   ├── controllers     // C 目录
│   ├── database
│   ├── filters.php
│   ├── lang
│   ├── models          // M 目录
│   ├── routes.php      // 路由配置文件,Request 请求到来后,由那些文件处理,在这里定义.
│   ├── start           // APP 对象启动前的加载文件
│   │   ├── artisan.php
│   │   ├── global.php
│   │   └── local.php
│   ├── storage         // 存储缓存,日志,session等文件的目录
│   ├── tests
│   └── views           // C 目录
├── artisan             // 命令行工具 artisan 的文件
├── bootstrap           // APP 启动前的加载文件
│   ├── autoload.php
│   ├── paths.php
│   └── 2.步 start.php  // 当请求到达后,第二个调用的文件, 创建 Laravel Application Object,并作为控制反转( IoC )的容器
├── composer.json
├── composer.lock
├── phpunit.xml
├── public              // 存放前台的静态文件,为 web 的 ROOT 目录,是 APP 的入口 index.php 的存放目录.
│   ├── favicon.ico
```

```
|   ├── 1.步 index.php      // APP入口文件
|   ├── packages
|   └── robots.txt
├── readme.md
├── server.php
└── vendor                  // Laravel Framework 源文件
```

Request Lifecycle 请求的处理流程， Laravel's bootstrap process

1. 所有到达 APP 的请求,首先全部进入 public/index.php 文件.

从这里开始 Laravel 开始着手处理到来的请求,并返回一个对应的回应.

2. 然后加载 bootstrap/start.php 文件,创建 Application (object) 并进行 detects environment

框架在这个文件中创建 **Laravel Application** 对象,作为 控制反转的**容器.**

关于Laravel运行环境

Laravel 可以配置多个运行环境，例如开发运行环境, 本机运行环境, 实际生产运行环境(默认值)等等, 在这些运行环境里可以重写默认运行环境配置文件中的定义. 新的环境配置文件是以新的环境名命名的目录,并存放在 `app/config` 目录下. 例如 `local`开发环境, 就需要创建 `app/config/local` 目录, 并在该目录下创建 app.php 文件, 就覆盖了默认运行环境配置文件 `app/config/app.php` 中的定义.
然后还需要告诉 Laravel 怎样监测它目前正在运行的环境是否为新添加的运行环境, 通过在 bootstrap/start.php 文件中 $app->detectEnvironment 调用中添加 `'local' => array('your-machine-name'),`让 Laravel 通过运行的机器名在判定应该使用那个运行环境. 还可以向 `detectEnvironment` 传递一个闭包, 让 Laravel 依靠其他机制判断当前运行环境. 例如:

```
$env = $app->detectEnvironment(function()
{
    return $_SERVER['MY_LARAVEL_ENV'];   // $_SERVER['MY_LARAVEL_ENV']; 属于敏感数据,从哪里定义见下面解释.
});
```

访问当前运行环境是通过，`$environment = App::environment();` 方法.

你还可以向 enviroment 方法中传递参数,来检查当前运行环境是否匹配给定值.

```
if (App::environment('local'))
{
    // The environment is local
}

if (App::environment('local', 'staging'))
{
    // The environment is either local OR staging...
}
```

有时,你可能需要"追加" 一些环境配置,而不是"覆盖" 配置, 此时需要,在相应运行环境目录下的配置文件中使用 append_config 方法.

```
'providers' => append_config(array(
    'LocalOnlyServiceProvider',
))
```

保护敏感配置

对于真正的"APP",最好保持敏感数据在配置文件外面, 例如, 数据库密码, API keys ,加密 KEY 等等.

Laravel 提供了一个非常简单的方法, 保存这些数据到 . 开头形式为 .env.环境名.php 的隐藏文件中.

例如,现在 Laravel 运行在 local 运行环境里, 那么创建在项目 root 目录下创建 .env.local.php 文件,让这个文件返回一个"键值对的数组" 就像 Larave 的其他配置文件一样:

```
<?php
return array(
    'TEST_STRIPE_KEY' => 'super-secret-sauce',
);
```

所有被这个文件返回的键值对,都将自动被添加到PHP 的 $_ENV 和 $_SERVER 超级变量中. 这样就可以在配置文件中引用这些变量了.

例如:

```
'key' => $_ENV['TEST_STRIPE_KEY']
```

还要确保添加 .env.local.php 文件到 .gitignore 文件中, 使其不会被添加到版本控制中.又可以使开发团队的其他成员创建他们自己的本地开放环境配置文件,由可以隐藏敏感数据.

在实际生产环境创建中需要创建 .env.php 隐藏文件.

3. Internal framework/start.php file configures settings and loads

Service Providers.

Framework 内部的 script 被调用, 根据监测的运行环境中配置文件对 App 进行设置, 如 timezone, error reporting 等等,

所有 Laravel 的配置文件都存储在 `app/config` 目录下.

访问/设置 "配置"

At run-time, 访问配置值, 是通过 `Config` 类实现的.

访问配置值, `Config::get('app.timezone');`

访问配置值,如果不存在就使用默认值, `$timezone = Config::get('app.timezone', 'UTC');`

注意: 点(.)访问符语法, 访问在不同的文件中的不同值.

设置属性值, `Config::set('database.default', 'sqlite');`

在运行环境设置属性值只会影响到当前的请求,不会对子请求有影响.

这时,还会进行一个非常重要的动作, 为 Application 对象注册所有配置过的 **Service Providers**. 主要指在`app/config/app.php`文件的`providers`数组中配置的 Service Providers. 这些 `Providers` 是 `Laraver` 的 primary bootstrapping mechanism.

Service Providers

`Service providers` 是把一组相关的控制反转(IoC) 注册在同一个位置的好办法.

把他们想象成为你程序的一组引导组件.

在一个 `Service providers` 中,

- 你可能会注册一个自定义认证的驱动,
- 或者在 `IoC container` 中注册你程序的存储类.
- 甚至设置一个自定的 Artisan 命令

事实上,大部分 `Laraver` 的核心组建都包含在 `Service Providers` 中.

定义一个 Service Providers

要创建一个 `Service providers` ,只需要简单的扩展 `Illuminate\Support\ServiceProvider` 类, 定义一个 `register` 方法:

```
use Illuminate\Support\ServiceProvider;

class FooServiceProvider extends ServiceProvider {

    public function register()
    {
```

```
        $this->app->bind('foo', function()
        {
            return new Foo;  // 返回 Foo 对象
        });
    }

}
```

注意: 在 register 方法中,程序的 Application IoC 容器是通过 $this->app 属性来访问的.

当创建好自己的 Provider, 并准备注册它到 Application 中时,只需要简单的添加它到 app/config/app.php 配置文件的 providers 数组中. 你可以在这个数据中找到一个 Server Provider 的列表. 或者要想在运行时注册一个 Provider ,需要使用 App::register 方法

```
App::register('FooServiceProvider');
```

本质上,每一个 Server Provider 是在容器中绑定了一个或多个的闭包. 允许在你的 Application 中访问这些绑定的服务.
当然, Server Provider 也可以运行任何的 bootstrapping task. 一个 Server Provider 可以注册 event listeners, view composers, **Artisan** commands, and more.

4. Application app/start files are loaded.

当所有 Server Provider 都注册好以后, app/start 里的 Start Files 文件将被加载.

Start Files 文件夹内默认这里有三个文件: global.php local.php 和 artisan.php

(1) artisan.php 是关于 Artisan command line 的文件.

(2) global.php 默认包含一些基础的项目,例如, 注册 Logger 和包含你的 app/filters.php 文件. 你可以在这里面自由的添加你想要的文件, 它将自动的添加到 application 的每个请求中, 而不考虑当前的运行环境.

(3) local.php 文件, 只有运行环境为 local 的时候,才会加载. 同理, 如果你在 bootstrap/start.php 中配置了 development 环境, 你就可以创建文件 app/start/development.php, 如果运行环境是 development 的话, 它将自动的被包含进 application 中.

什么应该放到 Start Files 中?

Start File 中可以存放简单的引导代码(bootstrapping code). 例

如,你可以注册一个 View composer, 配置你的 logging preferences, 进行一些 PHP设置,等等. 这完全取决于你.
当然, 把所有的引导代码都放到 start file 中,可能也会引起混乱, 对于大的 Application 来说, 如果你感到你的 Start File 开始混乱的时候, 可以考虑把他们转移到 `Service Providers` 中.

5. Application app/routes.php file is loaded.

最后是, `app/routes.php` 文件被加载.

6. Request object sent to Application, which returns Response object.

`Request object` 对象被发送到 `application`, 以便于它被分配到某个路由, 处理后, 然后返回 `Response object.`

7. Response object sent back to client.

Maintenance Mode 维护模式

当 Application 进入到维护模式后, 所有进入到程序的请求都被 routes 到一个自定义的 view 中.
在维护模式中, `App::down` 方法的返回值将会发送给用户.

进入 维护模式,需要在命令行执行 Artisan 命令:

```
php artisan down
```

这会导致调用默认定义的 `app/start/global.php` 文件中的 `App::down` 方法.

退出 维护模式,

```
php artisan up
```

当程序进入维护模式后,想要展示一个自定义的 view 的话,添加下面的代码到 `app/start/global.php` 文件中.

```
App::down(function()
{
    return Response::view('maintenance', array(), 503);
});
```

如果传入 `down` 方法的闭包返回 `NULL` 的话, 对新请求来说维护模式将被忽略.
当 App 进入维护模式后,没有 queue jobs 会被处理.

Application Events 应用程序事件

你或许需要在 Request Processing 之前或之后做一些处理, 就需要通过注册 `before, after, finish, shutdown` **应用程序事件**.

```
App::before(function($request)
```

```
{
    //
});

App::after(function($request, $response)
{
    //
});
```

这些注册的事件监听者,在每个到达 application 请求对象(Request)处理之前(before) 和之后(after)运行这些代码.
这些事件对于 全局过滤(global filtering) 和 全局修改(global modification) 回应(Responses)很有用.
这些事件可以被注册到 Start File 中, 或者一个 Server Provider 中.

你或许也需要注册一个 matched 事件的监听者. 它将在到来的请求(Request)被测试符合路由(route)规则,但是路由(route)还没有被调用时执行.

```
Route::matched(function($route, $request)
{
    //
});
```

当 Application 已经把回应(Response)发送会客户端后, finish 事件将会被调用.
这里是最后能够对到达你 Application 的请求处理的地方. 当 finish 事件完成处理后, shutdown 事件立即被调用, 这里是在脚本被终止前,做一些处理的最后机会.
大部分情况下,你是不需要用到这些事件.

Routing

关于程序的大部分路由规则都被定义在了 app/routes.php 文件中. 最简单的路由规则只有 URI 和 闭包回调.

基础的 GET 路由配置: 注意看紫色的不同 Route 函数.

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

基础的 POST 路由配置:

```
Route::post('foo/bar', function()
{
```

```
    return 'Hello World';
});
```

注册一个路由应对多个 HTTP 动作:

```
Route::match(array('GET', 'POST'), '/', function()
{
    return 'Hello World';
});
```

注册一个路由应对任何 HTTP 请求: 使用 any 方法.

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

强制一个只能 HTTPS 的路由:

```
Route::get('foo', array('https', function()    //第二个参数为一个数组
{
    return 'Must be over HTTPS';
}));
```

使用 `URL::to` 方法, 生成 URLs:

```
$url = URL::to('foo');
```

路由参数(Routing Parameters)

获取请求参数:

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

含有默认值的路由参数:

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

基于正规表达式的路由参数: (where 方法的使用)

```
Route::get('user/{name}', function($name)
{
    //
})
->where('name', '[A-Za-z]+');      //这里声明正则表达式
```

```
Route::get('user/{id}', function($id)
{
    //
})
->where('id', '[0-9]+');          //这里声明正则表达式
```

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

定义一个全局格式(Defining Global Patterns)

如果你想让一个路由参数(route parameter) 总是被限制在给定的正则表达式中, 你可以考虑使用 `pattern` 方法:

```
Route::pattern('id', '[0-9]+');

Route::get('user/{id}', function($id)
{
    // Only called if {id} is numeric.
});
```

访问一个路由参数值(Accessing A Route Parameter Value)

如果你需要在路由外, 访问一个路由参数值,你可能需要 `Route::input` 方法

```
Route::filter('foo', function() //这里是路由过滤器,看下面解释
{
    if (Route::input('id') == 1)
    {
        //
    }
});
```

路由过滤器(Routing Filter)

对于一个路由, 路由过滤器(Routing Filter) 提供了一个约定好了的方式达到限制访问目的. 对于你的站点来说,如果需要创建一个认证的区域这非常有用.
在 Laravel 框架内部有几个默认 过滤器, 包括一个 `auth` 过滤器, 一个 `auth.basic` 过滤器, 和一个 `csrf` 过滤器. 他们都位于 `app/filters.php` 文件中.

对于一个路由,可以有两个路由过滤器, `before` 和 `after` ,在定义路由时, 通过向路由的第二个参数的数组中指定过滤器的名字.

(1) 对于在定义前过滤器(before filter)的闭包函数时,可以传入3个参数,

分别为$route(路由对象), $request(请求对象), $value(路由传入的值,见下面实例).前过滤器在进入路由动作之前调用.

(2) 对于在定义后过滤器(after filter)的闭包函数时,可以出入3个参数,分别为$route(路由对象), $request(请求对象), $response(回应对象),后过滤器在路由动作的 return 前调用.

定义一个路由过滤器

```
Route::filter('old', function()
{
    if (Input::get('age') < 200)
    {
        return Redirect::to('home');
    }
});
```

如果过滤器返回一个 回应对象(Response), 这个对象会被认为是对请求(Request)的回应, 之后的路由(Route)将不会被调用运行. 在这个路由(Route)之上的其他 后过滤器(after) 都将被取消.

附加过滤器到路由上(Attaching A Filter To A Route)

```
Route::get('user', array('before' => 'old', function()  // Route::get方法的第二个参数为一个数组,before 表示为前过滤器, old 值为过滤器的名字.还可以添加 after 后过滤器.
{
    return 'You are over 200 years old!';
}));
```

附加多个过滤器到一个路由上, 过滤器名以 | 分开

```
Route::get('user', array('before' => 'auth|old', function()
{
    return 'You are authenticated and over 200 years old!';
}));
```

或以数组方式

```
Route::get('user', array('before' => array('auth', 'old'), function()
{
    return 'You are authenticated and over 200 years old!';
}));
```

向过滤器传入参数, 以 : 分离

```
Route::filter('age', function($route, $request, $value) //对于前过滤器来说,第三
```

```
个参数为 传入的参数值, 对于后过滤器来说,第三个参数为请求的回应对象
{
    //
});

Route::get('user', array('before' => 'age:200', function() // 以 : 分割向多虑起传入参数
{
    return 'Hello World';
}));
```

基础过滤器模式

对于基于特定 URL 的一组路由, 你可能需要指定基础 "过滤器" 以全部支持这些路由.

```
Route::filter('admin', function()
{
    //
});

Route::when('admin/*', 'admin'); //使用 * 通配符来指定一组基础路由, 注意,这里使用了路由的 when 方法
```

在上面的例子中, `admin` 过滤器,将会支持所有的 以 `admin/` 开头的路由. * 为通配符.

基于 HTTP 动作创建过滤器

```
Route::when('admin/*', 'admin', array('post'));  //注意,这里使用了路由的 when 方法
```

过滤器类(Filter Classes)

对于高级的过滤器,你可能需要用类替代闭包.

注册一个基于类的过滤器

```
Route::filter('foo', 'FooFilter');
```

默认,过滤器类的 `filter` 方法将要被调用.

```
class FooFilter {

    public function filter()
    {
        // Filter logic...
    }
```

```
}
```

如果,你不希望用 `filter` 方法,你需要在定义时指定方法名.

```
Route::filter('foo', 'FooFilter@foo');
```

命名路由(Named Routes)

当你需要生成一个 重定向(redirects) 或 URLs 时,使用一个路由的引用 会更方便. 这时就需要一个命名的路由了.

```
Route::get('user/profile', array('as' => 'profile', function() // as 为这个路由
指定名字
{
    //
}));
```

还可以为路由指定到一个控制器的动作(controller actions)

```
Route::get('user/profile', array('as' => 'profile', 'uses' => 'UserControll
er@showProfile'));    as 为这个路由指定名字
```

现在你可以这样使用这个路由的名字去生成重定向(redirects) 或 URLs.

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

$redirect = Redirect::route('profile');   //生成重定向, Redirect::route 方法
```

访问当前的"路由名"通过 currentRouteName 方法:

```
$name = Route::currentRouteName();     //获得定义 route 时,以 as 命名的路由名
```

路由组(Route Groups)

有时候,你可能需要将一个过滤器应用于一组路由. 相对于为每一个路由指定过滤器, 你可以将一组路由封装到一个路由组,然后在路由组上添加过滤器

```
Route::group(array('before' => 'auth'), function()    // group 方法构建路由组,
before 指定前过滤器
{
    Route::get('/', function()
    {
        // Has Auth Filter
    });

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

```
    {
        // Has Auth Filter
    });
});
```

在你的 group 数组中添加 namespace 参数,用于指定组内的控制器(controller)在一个给定的 namespace 中:

```
Route::group(array('namespace' => 'Admin'), function()   //使用 namespace 指定命名空间
{
    //
});
```

子域路由(Sub-Domain Routing)

Laravel Routes 同样有能力应对子域通配符, 并把匹配的值作为参数传递到路由:

Registering Sub-Domain Routes 注册一个子域路由

```
Route::group(array('domain' => '{account}.myapp.com'), function()  // domain 指定一个子域通配符
{

    Route::get('user/{id}', function($account, $id) //  第一个参数$account就为上面 account 的值, id 为user的id
    {
        // 如果访问 http://leo.myapp.com/user/100 ,那么 $account 值为 leo, $id 值为 100
    });

});
```

路由前缀(Route Prefixing)

一组路由可以在属性数组中通过使用 prefix 指定前缀.

```
Route::group(array('prefix' => 'admin'), function()
{

    Route::get('user', function()  // prefix 指定了前缀,要想访问到 user,需要访问 admin/user
    {
        //
    });
```

```
});
```

绑定路由与模型(Route Model Binding)

模型绑定提供了一个方便的方式,将模型实例插入到你的路由中. 例如: 相对于插入用户 ID, 你可以插入匹配这个用户 ID 的整个用户模型实例.
首先,使用 `Route::model` 方法指定对于给定的参数应该使用的模型.

首先,绑定一个参数到一个模型

```
Route::model('user', 'User');    // Route::model 方法绑定一个数据库的模型
```

然后, 定义一个包含 `{user}` 参数的路由

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

因为,我们绑定了 `{user}` 参数到 `User` 模型,因此一个 `User` 实例将要被插入到 `route`.
所以,对于请求 `profile/1` 将会插入用户 id 为 1 的 `User` 实例.
注意: 如果绑定的用户模型实例没有在数据库中找到,一个 404 的错误将会返回.

如果你希望指定自己的 "not found" 的行为, 你可能需要传递一个闭包作为一个 `model` 方法的第三个参数

```
Route::model('user', 'User', function()
{
    throw new NotFoundHttpException;
});
```

默认是以用户 ID 解析传入路由的user参数, 有时候对于传入的路由参数, 你需要使用你自己的 解析器(Resolver)解析. 此时使用 `Route::bind` 方法:

```
Route::bind('user', function($value, $route)
{
    return User::where('name', $value)->first();  //路由将收到以用户名查询的结果,而不是用户 id
});
```

上面绑定了新的解析器,当请求 `profile/1` ,时 1 将会被作为用户的名字,而不是 id 号.

触发 404 错误(Throwing 404 Errors)

在一个路由中,有两种方法手工触发一个404错误.

第一种, 你需要使用 `App::abort` 方法.

```
App::abort(404);
```

第二种, 你可以抛出一个 `Symfony\Component\HttpKernel\Exception\NotFoundHttpException` 实例.

处理404错误(Handling 404 Errors)

你可能需要在你的程序中注册一个"错误处理器",处理所有的"404 Not Found" 错误. 允许你返回自定义的 404 错误页面.

```
App::missing(function($exception)
{
    return Response::view('errors.missing', array(), 404); // 这会导致返回 app/view/errors/missing.php 文件
});
```

路由到控制器(Routing To Controllers)

Laravel 不仅仅允许你路由到一个闭包, 而且还允许路由到一个控制类(controller classes). 具体看下面的控制器的介绍.
甚至允许创建一个 资源控制器(resource controllers).

Requests & Input

基础输入(Basic Input)

你可以通过简单的方法,访问所有的用户输入,而不必担心对于到来请求的 HTTP 动作是 GET 还是 POST,或者其他.
对于多有的 HTTP 动作的用户输入访问都是一样的.

检索用户输入

```
$name = Input::get('name');
```

如果用户输入缺少的话,检索回一个默认值

```
$name = Input::get('name', 'Sally');
```

检索是否存在用户输入值

```
if (Input::has('name'))
{
```

```
    //
}
```

对于一个请求获取所有的用户输入(Getting All Input For The Request)

```
$input = Input::all();
```

只获取特定的一些用户输入值(Getting Only Some Of The Request Input)

```
$input = Input::only('username', 'password');

$input = Input::except('credit_card');
```

当跟 表单的 "数据" 输入交互时, 你可以使用**点(.)**操作符,访问数组.

```
$input = Input::get('products.0.name');
```

注意: 一些 javascript 库, 像 Backbone 也许发送 用户输入(input) 到 application 通过 JSON 格式. 你可以像其他普通的获取 input 值一样,通过 `Input::get` 方法访问.

Cookies

所有被 Laravel framework 创建的 cookies 都被使用一个认证代码(authentication code)进行了加密和签名, 这意味着,如果 cookies 被客户端篡改的话,那么它将会失效.

检索一个 cookie 值

```
$value = Cookie::get('name');
```

对于一个回应对象粘附上一个新的 Cookie

```
$response = Response::make('Hello World');

$response->withCookie(Cookie::make('name', 'value', $minutes));
```

对下一个的回应排队创建一个 cookie

如果你想在一个回应对象(Response)被创建之前设置一个 cookie 的话,使用 `Cookies::queue()` 方法. 这个 cookie 将会自动被依附到从你程序发出的最终回应对象上.

```
Cookie::queue($name, $value, $minutes);
```

创建一个永远不过期的 cookie

```
$cookie = Cookie::forever('name', 'value');
```

旧输入值(Old Input)

你有时可能需要保持输入值, 从一个请求到另一个请求. 例如: 当你检查到提交上来

的表格存在效验错误的话,可能需要退回并重现它

将输入值存储到 session 中(Flashing Input To The Session)

```
Input::flash();
```

只保存部分值到 session 中,避免敏感信息

```
Input::flashOnly('username', 'email');

Input::flashExcept('password');
```

对于需要重定向到之前的页面, 而你又经常需要保存对应的输入(input)这种情况, 你可以简单的连接 输入保存(input flashing) 到一个 重定向(redirect)

```
return Redirect::to('form')->withInput();

return Redirect::to('form')->withInput(Input::except('password'));
```

注意: 你可以使用 **`Session Class`** 跨请求存储其他数据

检索回旧的输入值(Retrieving Old Data)

```
Input::old('username');
```

文件(Files)

检索一个上传的文件

```
$file = Input::file('photo');
```

监测文件是否已经上传了

```
if (Input::hasFile('photo'))
{
    //
}
```

被 `file` 方法返回的对象是一个 `Symfony\Component\HttpFoundation\File\UploadedFile` 类的实例, 它扩展了PHP 的 `SplFileInfo` 类,并提供了多种与文件交互的方法.

监测一个上传的文件是否是有效的

```
if (Input::file('photo')->isValid())
{
    //
}
```

移动一个上传的文件

```
Input::file('photo')->move($destinationPath);
```

```
Input::file('photo')->move($destinationPath, $fileName);
```

检索一个上传文件的路径

```
$path = Input::file('photo')->getRealPath();
```

检索一个上传文件的原始名称

```
$name = Input::file('photo')->getClientOriginalName();
```

检索一个上传文件的扩展名

```
$extension = Input::file('photo')->getClientOriginalExtension();
```

检索上传文件的大小

```
$size = Input::file('photo')->getSize();
```

检索上传文件的 MIME 类型

```
$mime = Input::file('photo')->getMimeType();
```

请求信息(Request Information)

Request 类提供很多方法来检索到达 Application 的请求(Request), 同时他还扩展了 Symfony\Component\HttpFoundation\Request 类.
这个类定义在文件
vendor/laravel/framework/src/Illuminate/Http/Request.php 中.

检索获取请求的 URI

```
$uri = Request::path();
```

获取请求 URL

```
$url = Request::url();
```

检索获取请求方法

```
$method = Request::method();

if (Request::isMethod('post'))
{
    //
}
```

检测请求的路径是否符合给定的模式

```
if (Request::is('admin/*'))
{
    //
}
```

检索一个请求 URI 的分片

```
$segment = Request::segment(1);
```

检索一个请求头

```
$value = Request::header('Content-Type');
```

从 $_SERVER 变量中检索一个值

```
$value = Request::server('PATH_INFO');
```

检测请求是否是通过 HTTPS

```
if (Request::secure())
{
    //
}
```

检测请求是否使用的是 Ajax

```
if (Request::ajax())
{
    //
}
```

检测请求类型是否含有 JSON Content Type

```
if (Request::isJson())
{
    //
}
```

检测请求是否是想要 JSON 的返回(AcceptableContentTypes == 'application/json')

```
if (Request::wantsJson())
{
    //
}
```

检查发起请求时,要求服务器的回应格式, 基于 HTTP Accept header 信息

```
if (Request::format() == 'json') //这个跟上面的方法其实是一样的
{
    //
}
```

Views & Responses

Basic Responses

从路由返回一个字符串

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

创建一个自定义的回应, 回应对象(Response)继承自 Symfony\Component\HttpFoundation\Response 类,提供了丰富的方法构建 HTTP 回应对象.

```
$response = Response::make($contents, $statusCode);

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

return $response;
```

如果你需要使用 Response 的类方法,返回一个 view 作为回应内容的话, 那么使用 Response::view 方法就很方便

```
return Response::view('hello')->header('Content-Type', $type);
```

粘附 cookies 到回应对象

```
$cookie = Cookie::make('name', 'value');

return Response::make($content)->withCookie($cookie);
```

重定向(Redirects)

返回一个重定向

```
return Redirect::to('user/login');
```

返回一个重定向, 并存储数据

```
return Redirect::to('user/login')->with('message', 'Login Failed');
```

注意: 因为 `with` 方法存储数据到 Session, 你可以使用 Session::get 方法检索回存储的数据.

返回一个重定向到一个命名了的路由

```
return Redirect::route('login'); //路由的命名看上面,主要是在定义路由时,在数据属性中添加 as
```

返回一个重定向到一个命名了的路由, 并传递参数

```
return Redirect::route('profile', array( 1 ));
```

返回一个重定向到一个命名了的路由, 并传递命名了的参数

```
return Redirect::route('profile', array( 'user' => 1 ));
```

返回一个重定向到一个动作控制器, 并传递参数, 或者带名字的参数

```
return Redirect::action('HomeController@index'); // 导致会调用 HomeController
 的 index 方法
```

```
return Redirect::action('UserController@profile', array(1));
```

```
return Redirect::action('UserController@profile', array('user' => 1));
```

视图(view)

视图一般包含着你 Application 的 HTML 代码. 并提供了一个方便方式, 使你的控制器和站内逻辑从展示中分离出来.
视图(View)存储在 `app/view` 目录下.

一个简单的视图是这样的

```
<!-- View stored in app/views/greeting.php -->

<html>
    <body>
        <h1>Hello, <?php echo $name; ?></h1>
    </body>
</html>
```

这个视图可以像这样被返回给浏览器

```
Route::get('/', function()
{
    return View::make('greeting', array('name' => 'Taylor'));  //使用 make
方法生产 view 视图
});
```

传递给 View::make 的第二个数组参数的数据,可以在 view 中被查看.

向视图传递数据

```
// Using conventional approach
$view = View::make('greeting')->with('name', 'Steve');

// Using Magic Methods
$view = View::make('greeting')->withName('steve');
```

如果你愿意,你可以给向 `make` 方法的第二个参数传递一个包含数据的数组.

```
$view = View::make('greetings', $data);
```

你还可以在所有的 view 之间共享数据片段

```
View::share('name', 'Steve');
```

向一个视图中传递一个子视图(Sub-view)

有时候,你可能需要将一个 view 传递到另一个 view 中去, 例如,将一个存储在 app/views/child/view.php 中的视图, 传递给另一个 view

```
$view = View::make('greeting')->nest('child', 'child.view'); // 第一个
 child 为 key,就是这个 view 的一个名字,下面用到

$view = View::make('greeting')->nest('child', 'child.view', $data); /
/ 第一个 child 为 key,就是这个 view 的一个名字,下面用到
```

然后这个子视图(Sub-view) 可以在父视图中像这样被渲染

```
<html>
    <body>
        <h1>Hello!</h1>
        <?php echo $child; ?>
    </body>
</html>
```

视图设计者(View Composers)

View Composers 是 view 在渲染时的回调函数或类方法.

如果有些数据, 你想你的视图(view)每次通过 application 渲染的时候,这些数据都跟渲染的 view 绑定起来, 那么 View Composers 能够把这些代码组织到一个单独的地方. 从这方面看的话, View Composer 更像是 "视图模型(View Models)" 或者 "表现者(presenters)".

定义一个 View Composers

```
View::composer('profile', function($view)     // porfile 为 一个 view 的名字
{
    $view->with('count', User::count());     //count 为绑定到 profile view 的
数据
});
```

现在每次 profile 视图(view)被渲染的时候, count 数据都会被绑定到这个视图(view).

基于 view composer 的类

如果你宁愿使用一个基于 composer 的类的话, 你可以通过 Application 控制反转(IoC) 容器 解决这个需求, 你可以这样做:

```
View::composer('profile', 'ProfileComposer');    // porfile 为 一个 view 的名字,
```

一个基于 view composer 的类应该这样被定义:

```
class ProfileComposer {

    public function compose($view)   //定义一个 compose 方法,将会被调用
    {
        $view->with('count', User::count());
    }

}
```

定义多个 Composers

你可以使用 `composers` 方法在同一时间注册一个关于 composer 的组.

```
View::composers(array(
    'AdminComposer' => array('admin.index', 'admin.profile'),
    'UserComposer' => 'user',
));
```

注意: 这里没有规定 composers 应该被存储在哪里. 你可以自由的存储他们在任何的地方,只要他们可以被你 composer.json 文件中的指令自由加载就好.

视图创建者(View Creators)

View Creators 基本上和 View Composers 做一样工作. 但是, 他们是在 view 被实例化(instantiated)后立即被调用.

要注册一个 View Creators 可以使用 View 类的 `creator` 方法.

```
View::creator('profile', function($view)  // porfile 为 一个 view 的名字
{
    $view->with('count', User::count());
});
```

特殊的回应(Special Responses)

创建一个 JSON 回应

```
return Response::json(array('name' => 'Steve', 'state' => 'CA'));
```

创建一个 JSONP 回应

```
return Response::json(array('name' => 'Steve', 'state' => 'CA'))->setCallback(Input::get('callback'));
```

创建一个文件下载回应

```
return Response::download($pathToFile);

return Response::download($pathToFile, $name, $headers);
```

注意: Symfony HttpFoundation 类管理着文件下载功能, 被请求下载的文件必须有一个 ASCII 文件名.

回应宏 (Response Macros)

如果你想定义一个能够被多个 路由(Routes) 和 控制器(controller) 重用的 回应(Response), 你可以使用 `Response::macro` 方法

```
Response::macro('caps', function($value)   // caps  为宏的名字
{
    return Response::make(strtoupper($value));
});
```

`macro` 方法使用它的第一个参数作为它的名字, 第二个参数为一个闭包. 当在 回应类(Response) 中调用宏名字时, 这个闭包将会被执行.

```
return Response::caps('foo');   //掉用一个定义好了的宏, 使用上面定义的宏名字作为方法名
```

你可以在把你的宏定义在 app/start 中的一个文件里. 或者你可以组织你的宏到一个单独的文件,然后在 start 文件中包含它.

控制器(Controllers)

基础控制器 (Basic Controllers)

相对于把所有 路由级别(Route-Level) 的逻辑定义在单一的 routes.php 文件中, 你可能更愿意使用 控制类(Controller Class) 来组织这些行为(behavior). 控制器(Controllers) 可以组织相关的 路由逻辑 到一个类中, 而且还能够采用更多高级框架的特征,像自动的 dependency injection.

控制器一般存放在 app/controllers 目录下,这个目录是在 composer.json 文件的 classmap 选项中被默认注册的. 然而, 控制器可以存储在任何的目录或者子目录中.路由声明(Route declarations)并不依赖于 控制器类 在硬盘上的存储位置, 所以只要 Composer 知道怎样自动加载 控制器类(controller class), 它就可以存放在任何你想存放的位置.

这里是一个基础控制器类的例子:

```
class UserController extends BaseController {  //所有的控制器都应该扩展于 BaseController
```

```
    /**
     * Show the profile for the given user.
     */
    public function showProfile($id)
    {
        $user = User::find($id);

        return View::make('user.profile', array('user' => $user));
    }

}
```

所有的控制器都应该扩展于 `BaseController`, `BaseController`类被存储在了 app/controllers 目录下, 这里可以被用作存放共享控制逻辑的地方. `BaseController`类扩展于框架的 `Controller` 类, 现在我们可以像这样路由到这个控制的动作.

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

如果你选择使用 PHP 的命名空间(namespaces) 嵌套或者组织你的 控制器(controller), 只要在定义 路由 的时候使用 完全限定类名(fully qualified class name)

```
Route::get('foo', 'Namespace\FooController@method');
```

注意: 因为我们使用了 Composer 来自动加载我们的 PHP 类, 控制器 可以存放在文件系统的任何位置,只要 Composer 知道怎样加载他们. 控制器目录并不强制要求你的 application 的拥有某种目录结构. 路由到你的控制器是和整个于文件系统耦合的.

你还可以在控制器路由上指定名字:

```
Route::get('foo', array('uses' => 'FooController@method',
                                        'as' => 'name'));
```

要生成到一个控制器下动作的 URL, 你可以使用 `URL::action` 方法,或者使用 action helper method:

```
$url = URL::action('FooController@method');

$url = action('FooController@method');
```

你可以通过运行 `currentRouteAction` 方法获取运行的 控制器下的名字 (controller action).

```
$action = Route::currentRouteAction();
```

控制器过滤器(Controller Filters)

过滤器(Filters)可以被指定在控制器的路由(Controller Routes) 上就像正规的路由一样

```
Route::get('profile', array('before' => 'auth', // 设置前过滤器
            'uses' => 'UserController@showProfile'));
```

然后,你还可以在你的控制器内部设置过滤器

```
class UserController extends BaseController {

    /**
     * Instantiate a new UserController instance.
     */
    public function __construct()     //构造函数
    {
        $this->beforeFilter('auth', array('except' => 'getLogin'));  //设置前过滤器

        $this->beforeFilter('csrf', array('on' => 'post'));

        $this->afterFilter('log', array('only' =>
                            array('fooAction', 'barAction')));   //设置 后过滤器
    }

}
```

你还可以使用一个闭包来指定控制器的过滤器

```
class UserController extends BaseController {

    /**
     * Instantiate a new UserController instance.
     */
    public function __construct()
    {
        $this->beforeFilter(function()
        {
            //
        });
    }

}
```

如果你想使用控制器内的另外一个方法作为控制器的过滤器, 你可以使用 @ 语法定义这个过滤器.

```
class UserController extends BaseController {

    /**
```

```
     * Instantiate a new UserController instance.
     */
    public function __construct()
    {
        $this->beforeFilter('@filterRequests');     //指定一个内部的方法作为过滤器

    }

    /**
     * Filter the incoming requests.
     */
    public function filterRequests($route, $request)
    {
        //
    }

}
```

RESTful Controllers

Laravel allows you to easily define a single route to handle every action in a controller using simple, REST naming conventions. First, define the route using the Route::controller method:

Laravel 允许你很容易的定义一条单独路由, 到一个可以使用 REST 命名规范来处理所有动作的控制器(Controller)中.
首先, 使用 `Route::controller` 方法定义一个路由.

```
Route::controller('users', 'UserController');
```

Contoller 方法接受两个参数,第一个为控制器处理的 基础 URI , 第二个是一个控制器的类名.
接下来,在控制器中添加一些 以 HTTP 动作(GET,POST,DELETE)开头的回应方法.

```
class UserController extends BaseController {

    public function getIndex()
    {
        //
    }

    public function postProfile()
    {
        //
    }

}
```

index 方法可以使控制器回应到根 基础 URI 的请求,在上面的案例中是 users

如果在控制器的动作方法中包含多个单词,你可以在 URI 中使用下划线(_)语法来访问这个动作. 例如下面在我们的 UserController 类中的控制器动作方法,可以回应的URI 为 users/admin-profile

```
public function getAdminProfile() {}
```

资源控制器(Resource Controllers)

资源控制器(Resource Controllers)可以很容易的围绕资源构建 RESTful 控制器. 例如, 你可能希望创建一个管理 Application 存储的照片的控制器. 通过 Artisan 命令行的 `controller::make` 命令和 `Route::resource` 方法我们可以很快的创建这个控制器:

通过命令行创建控制器,执行下面的命令:

```
php artisan controller:make PhotoController
```

现在我们注册一个 资源路由(resourceful route) 到这个控制器:

```
Route::resource('photo', 'PhotoController');
```

这个单一的路由申明,创建了处理路由到照片资源上的各种 RESTful 动作. 同样的,生成的控制器也已经有了各种的预设方法来处理这些动作,并伴随这注释信息表明他们处理那个 URI 和动作

被资源处理器处理的动作

Verb	Path	Action	Route Name
GET	/resource	index	resource.index
GET	/resource/create	create	resource.create
POST	/resource	store	resource.store
GET	/resource/{resource}	show	resource.show
GET	/resource/{resource}/edit	edit	resource.edit
PUT/PATCH	/resource/{resource}	update	resource.update
DELETE	/resource/{resource}	destroy	resource.destroy

有时候,你可能进需要去处理一个资源动作的子集:

```
php artisan controller:make PhotoController --only=index,show

php artisan controller:make PhotoController --except=index
```

你还可以在路由上指定处理动作的子集:

```
Route::resource('photo', 'PhotoController',
                array('only' => array('index', 'show')));

Route::resource('photo', 'PhotoController',
                array('except' => array('create', 'store', 'update', 'destr
oy')));
```

默认情况下,所有的资源控制器动作都有一个路由名字, 然后,你可以通过传递一个 `name` 键值的数组来重写这个名字.

```
Route::resource('photo', 'PhotoController',
                array('names' => array('create' => 'photo.build')));
```

处理不存在的方法(Handling Missing Methods)

在一个控制器中如果找不到一个匹配的方法来处理到来的请求,一个能够捕获所有请求的方法将要被调用. 这个方法应该被命名为 `missingMethod` ,并且接收请求参数.

定义一个捕获所有的方法

```
public function missingMethod($parameters = array()) // $paramters 参数为本该
到其他方法的请求的参数
{
    //例如,如果请求的 URL 为 user/miss/misss2/miss3
    //那么,paramters 的值为
    //array (size=3)
    //   0 => string 'miss' (length=4)
    //   1 => string 'misss2' (length=6)
    //   2 => string 'miss3' (length=5)


}
```

错误与日志(Errors & Logging)

配置(Configuration)

对于你 `Application` 日志相关的处理在开始文件(start files)中的

app/start/global.php 中被注册. 默认情况下, 日志处理器(logger) 被配置使用单一日志文件.然后,你可以根据你的需求自定义这些行为. 因为 Laravel 采用了很流行的 Monolog 日志库,你可以利用 Monolog 库提供的各种处理器.

例如,如果你希望使用每天的日志文件,替代单一的巨大日志文件,你可以在你的 start file 中做出下面的更改:

```
$logFile = 'laravel.log';

Log::useDailyFiles(storage_path().'/logs/'.$logFile);
```

错误日志(Error Detail)

默认情况下,对于你的 Application 的错误细节是被记录的. 这意味这,当一个错误出现的时候,你将会看到一个包含了详细的栈跟踪和错误信息的错误页面.
你可以通过设置 app/config/app.php 文件里的 debug 选项为 false 来关闭错误信息.

注意: 强力推荐你在生产环境关闭详细错误信息的展示.

处理错误(Handling Errors)

默认情况下,在 app/start/global.php 文件中包含了一个对所有错误 exceptions 的处理器.

```
App::error(function(Exception $exception)
{
    Log::error($exception);
});
```

这时最基本错误处理器. 如果你需要的话,你可以指定更多的处理器. 处理器按照他们处理的错误Exception的类型来调用. 例如,你可以创建一个处理器, 只处理 RuntimeException 实例:

```
App::error(function(RuntimeException $exception)
{
    // Handle the exception...
});
```

如果一个错误处理器返回一个回应(Response), 这个回应将会被放松到客户端浏览器,没有其他的错误处理器会被继续调用.

```
App::error(function(InvalidUserException $exception)
{
    Log::error($exception);
```

```
    return 'Sorry! Something is wrong with this account!';
});
```

监听 PHP fatal errors, 你可能需要 `App::fatal` 方法

```
App::fatal(function($exception)
{
    //
});
```

如果你有一些 exception处理器, 那么你应该从大而全到小而具体的方向去定义他. 例如: 一个处理所有 `Exception` 类型 exceptions 的处理器应该在一个用户自定义的 `Illuminate\Encryption\DecryptException` 类型之前定义

在哪里放置错误处理器(Where To Place Error Handlers)

框架没有定义默认的的位置来注册错误处理器(error handler). 在这方面 Laravel 给了你自由的空间. 不过一个不错的选择是在 `start/global.php` 文件中定义. 一般来说,这是一个约定的位置用来存放 "启动代码". 如果这个文件变的太大而臃肿的话,你可以创建自己的 app/errors.php 文件. 然后从你的 start/global.php 文件中 `require` 这个文件. 第三个选项是创建一个 Server Provider 来注册这些处理器. 再声明一下,这里没有标准的答案, 你可以选择你舒服的方式.

HTTP Exceptions

一些 exceptions 描述了服务器的 HTTP 错误代码. 例如, 像 "page not found" 错误(404), "unauthorized error"(401) 或者 一个服务器内部错误(500).为了能够返回这些回应(Response),使用下面的代码:

```
App::abort(404);
```

你还可以提供一个回应消息的选项

```
App::abort(403, 'Unauthorized action.');
```

这个方法可以在 请求生命周期(request's lifecycle) 的任何地方使用.

Handling 404 Errors

你可以在 application 中注册一个错误处理器,来处理 "404 Not Found" 错误. 这可以使你很容易的返回一个 404 错误页面:

```
App::missing(function($exception)
```

```
{
    return Response::view('errors.missing', array(), 404);
});
```

Logging

Larvel 日志功能提供了一个在 Monolog 库之上的简单层. 默认情况下, Laravel 被配置为你的 Application 创建了单一的日志文件, 这个文件存储在 `app/storage/logs/laravel.log` ,你可以向这个文件中写入信息.

```
Log::info('This is some useful information.');

Log::warning('Something could be going wrong.');

Log::error('Something is really going wrong.');
```

日志处理器(logger) 定义了一下在 RFC5424 中定义的日志级别: **debug, info, notice, warning, error, critical,** and **alert.**
作为上下文信息的数组也可能被传递进日志方法中.

```
Log::info('Log message', array('context' => 'Other helpful information'));
```

Monolog 还提供了多种额外的处理器(handlers)可以在 logging 中使用. 如果有需要, 你可以访问被 Laravel 在底层使用的 Monolog 实例.

```
$monolog = Log::getMonolog();
```

你还可以注册一个事件来捕获所有的传递给 log 的消息.

注册一个日志事件监听者(Registering A Log Listener)

```
Log::listen(function($level, $message, $context)
{
    //
});
```