第二章 控制器详解

2.1 控制器定义

2.1.1 定义控制器

控制器文件通常放在`controller`下面，类名和文件名保持大小写一致，并采用驼峰命名（首字母大写）。
一个典型的（多应用）控制器类定义如下：

```
<?php
namespace app\index\controller;

class Index
{
    public function index()
    {
        return 'index';
    }
}
```

访问URL地址是（假设没有定义路由的情况下）

```
http://localhost/index.php/index
```

如果你的控制器是`HelloWorld`，并且定义如下：

```
<?php
namespace app\index\controller;

class HelloWorld
{
    public function index()
    {
        return 'hello, world! ';
    }
}
```

访问URL地址是（假设没有定义路由的情况下）

```
http://localhost/index.php/hello_world
```

2.1.2 渲染输出

默认情况下，控制器的输出全部采用return的方式，无需进行任何的手动输出，系统会自动完成渲染内容的输出。
下面都是有效的输出方式：

```
<?php
namespace app\index\controller;
```

```php
class Index
{
    public function hello()
    {
        // 输出hello,world!
        return 'hello,world!';
    }

    public function json()
    {
        // 输出JSON
        return json($data);
    }

    public function read()
    {
        // 渲染默认模板输出
        return view();
    }

}
```

注意：不要在控制器中使用包括`die`、`exit`在内的中断代码。如果你需要调试并中止执行，可以使用系统提供的`halt`助手函数。

2.2 空控制器

空控制器的概念是指当系统找不到指定的控制器名称的时候，系统会尝试定位当前应用下的空控制器(`Error`)类，利用这个机制我们可以用来定制错误页面和进行URL的优化。

例如，下面是单应用模式下，我们可以给项目定义一个`Error`控制器类。

```php
<?php
namespace app\controller;

class Error
{
    public function __call($method, $args)
    {
        return 'error request!';
    }
}
```

2.3 资源控制器和中间件

2.3.1 资源控制器

资源控制器可以让你轻松的创建`RESTFul`资源控制器，可以通过命令行生成需要的资源控制器，例如生成index应用的Blog资源控制器使用：

```
php think make:controller index@Blog
```

或者使用完整的命名空间生成

```
php think make:controller app\index\controller\Blog
```

如果只是用于接口开发，可以使用

```
php think make:controller index@Blog --api
```

然后你只需要为资源控制器注册一个资源路由：

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

设置后会自动注册7个路由规则，对应资源控制器的7个方法。

2.3.2 控制器中间件

支持为控制器定义中间件，你只需要在你的控制器中定义`middleware`属性，例如：

```
<?php
namespace app\controller;

class Index
{
    protected $middleware = ['Auth'];

    public function index()
    {
        return 'index';
    }

    public function hello()
    {
        return 'hello';
    }
}
```

当执行`index`控制器的时候就会调用`Auth`中间件，一样支持使用完整的命名空间定义。

如果需要设置控制器中间的生效操作，可以如下定义：

```
<?php
namespace app\index\controller;

class Index
{
    protected $middleware = [
        'Auth'  => ['except'    => ['hello'] ],
        'Hello' => ['only'      => ['hello'] ],
    ];

    public function index()
    {
        return 'index';
    }

    public function hello()
    {
        return 'hello';
    }
}
```

可以通过给请求对象赋值的方式传参给控制器（或者其它地方），例如

```
<?php

namespace app\http\middleware;

class Hello
{
    public function handle($request, \Closure $next)
    {
        $request->hello = 'ThinkPHP';

        return $next($request);
    }
}
```

然后在控制器的方法里面可以直接使用

```
public function index(Request $request)
{
    return $request->hello; // ThinkPHP
}
```

2.4 请求对象和请求信息

2.4.1 请求对象

当前的请求对象由`think\Request`类负责，该类不需要单独实例化调用，通常使用依赖注入即可。在其它场合则可以使用`think\facade\Request`静态类操作。

构造方法注入，一般适用于没有继承系统的控制器类的情况。

```
?php

namespace app\index\controller;

use think\Request;

class Index
{
    /**
     * @var \think\Request Request实例
     */
    protected $request;

    /**
     * 构造方法
     * @param Request $request Request对象
     * @access public
     */
    public function __construct(Request $request)
    {
        $this->request = $request;
    }
```

```
    public function index()
    {
        return $this->request->param('name');
    }
}
```

操作方法注入，每个方法都可以使用依赖注入。

```
<?php

namespace app\index\controller;

use think\Request;

class Index
{

    public function index(Request $request)
    {
        return $request->param('name');
    }
}
```

静态调用，在没有使用依赖注入的场合，可以通过`Facade`机制来静态调用请求对象的方法（注意`use`引入的类库区别）

```
<?php

namespace app\index\controller;

use think\facade\Request;

class Index
{

    public function index()
    {
        return Request::param('name');
    }
}
```

2.4.2 请求信息

`Request`对象支持获取当前的请求信息，包括：

方法	含义
host	当前访问域名或者IP
scheme	当前访问协议
port	当前访问的端口
remotePort	当前请求的REMOTE_PORT
protocol	当前请求的SERVER_PROTOCOL
contentType	当前请求的CONTENT_TYPE
domain	当前包含协议的域名
subDomain	当前访问的子域名
panDomain	当前访问的泛域名
rootDomain	当前访问的根域名

```
url                 当前完整URL
baseUrl             当前URL（不含QUERY_STRING）
query               当前请求的QUERY_STRING参数
baseFile            当前执行的文件
root                URL访问根地址
rootUrl             URL访问根目录
pathinfo            当前请求URL的pathinfo信息（含URL后缀）
ext                 当前URL的访问后缀
time                获取当前请求的时间
type                当前请求的资源类型
method               当前请求类型
rule                当前请求的路由对象实例
```

对于上面的这些请求方法，一般调用无需任何参数，但某些方法可以传入`true`参数，表示获取带域名的完整地址，例如：

```
use think\facade\Request;
// 获取完整URL地址 不带域名
Request::url();
// 获取完整URL地址 包含域名
Request::url(true);
// 获取当前URL（不含QUERY_STRING） 不带域名
Request::baseFile();
// 获取当前URL（不含QUERY_STRING） 包含域名
Request::baseFile(true);
// 获取URL访问根地址 不带域名
Request::root();
// 获取URL访问根地址 包含域名
Request::root(true);
```

获取当前控制器和操作，可以通过请求对象获取当前请求的控制器和操作。

```
方法                含义
controller          当前请求的控制器名
action              当前请求的操作名
```

获取当前控制器

```
Request::controller();
```

返回的是控制器的驼峰形式（首字母大写），和控制器类名保持一致（不含后缀）。
如果需要返回小写可以使用

```
Request::controller(true);
```

如果要返回小写+下划线的方式，可以使用

```
parse_name(Request::controller());
```

如果使用了多应用模式，可以通过下面的方法来获取当前应用

```
app('http')->getName();
```

2.5 输入变量和请求类型

2.5.1 输入变量

检测变量是否设置

```
// 可以使用has方法来检测一个变量参数是否设置，如下：
Request::has('id','get');
Request::has('name','post');
```

变量获取

```
// 变量获取使用\think\Request类的如下方法及参数：
// 函数规则 变量类型方法('变量名/变量修饰符','默认值','过滤方法')


方法              描述
param             获取当前请求的变量
get               获取 $_GET 变量
post              获取 $_POST 变量
put               获取 PUT 变量
delete            获取 DELETE 变量
session           获取 SESSION 变量
cookie            获取 $_COOKIE 变量
request           获取 $_REQUEST 变量
server            获取 $_SERVER 变量
env               获取 $_ENV 变量
route             获取 路由（包括PATHINFO） 变量
middleware        获取 中间件赋值/传递的变量
file              获取 $_FILES 变量
```

获取 param 变量

```
// PARAM类型变量是框架提供的用于自动识别当前请求的一种变量获取方式，是系统推荐的获取请求参数的方法，用法如下：
// 获取当前请求的name变量
Request::param('name');
// 获取当前请求的所有变量（经过过滤）
Request::param();
// 获取当前请求未经过滤的所有变量
Request::param(false);
// 获取部分变量
Request::param(['name', 'email']);
```

默认值

```
// 获取输入变量的时候，可以支持默认值，例如当URL中不包含$_GET['name']的时候，使用下面的方式输出的结果比较。
Request::get('name'); // 返回值为null
Request::get('name',''); // 返回值为空字符串
Request::get('name','default'); // 返回值为default
```

变量过滤

```
// 框架默认没有设置任何全局过滤规则，你可以在app\Request对象中设置filter全局过滤属性：
namespace app;

class Request extends \think\Request
{
    protected $filter = ['htmlspecialchars'];
}

// 也可以在获取变量的时候添加过滤方法，例如：
Request::get('name','','htmlspecialchars'); // 获取get变量 并用htmlspecialchars函数过滤
Request::param('username','','strip_tags'); // 获取param变量 并用strip_tags函数过滤
Request::post('name','','org\Filter::safeHtml'); // 获取post变量 并用org\Filter类的safeHtml方法过滤

// 可以支持传入多个过滤规则
Request::param('username','','strip_tags,strtolower'); // 获取param变量 并依次调用strip_tags、strtolower函数过滤

// 如果当前不需要进行任何过滤的话，可以使用
// 获取get变量 并且不进行任何过滤 即使设置了全局过滤
Request::get('name', '', null);
```

变量修饰符

```
// 用法如下
Request::变量类型('变量名/修饰符');
变量的修饰符，包括：
修饰符                作用
 s                    强制转换为字符串类型
 d                    强制转换为整型类型
 b                    强制转换为布尔类型
 a                    强制转换为数组类型
 f                    强制转换为浮点类型

 // 使用例子
Request::get('id/d');
Request::post('name/s');
Request::post('ids/a');
```

中间件变量

```
// 可以在中间件里面设置和获取请求变量的值，这个值的改变不会影响PARAM变量的获取。
<?php

namespace app\http\middleware;

class Check
{
    public function handle($request, \Closure $next)
    {
        if ('think' == $request->name) {
            $request->name = 'ThinkPHP';
        }
```

```
        return $next($request);
    }
}
```

2.5.2 请求类型

在很多情况下面，我们需要判断当前操作的请求类型是GET、POST、PUT、DELETE或者HEAD，一方面可以针对请求类型作出不同的逻辑处理，另外一方面有些情况下面需要验证安全性，过滤不安全的请求。
请求对象Request类提供了下列方法来获取或判断当前请求类型：

用途	方法
获取当前请求类型	method
判断是否GET请求	isGet
判断是否POST请求	isPost
判断是否PUT请求	isPut
判断是否DELETE请求	isDelete
判断是否AJAX请求	isAjax
判断是否PJAX请求	isPjax
判断是否JSON请求	isJson
判断是否手机访问	isMobile
判断是否HEAD请求	isHead
判断是否PATCH请求	isPatch
判断是否OPTIONS请求	isOptions
判断是否为CLI执行	isCli
判断是否为CGI模式	isCgi

请求类型伪装

支持请求类型伪装，可以在`POST`表单里面提交`_method`变量，传入需要伪装的请求类型，例如：

```
<form method="post" action="">
    <input type="text" name="name" value="Hello">
    <input type="hidden" name="_method" value="PUT" >
    <input type="submit" value="提交">
</form>
```

提交后的请求类型会被系统识别为`PUT`请求。如果要获取原始的请求类型，可以使用。

```
Request::method(true);
```

2.6 响应输出和下载

2.6.1 响应输出

大多数情况，我们不需要关注Response对象本身，只需要在控制器的操作方法中返回数据即可。
最简单的响应输出是直接在路由闭包或者控制器操作方法中返回一个字符串，例如：

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

<?php
```

```
namespace app\controller;

class Index
{
    public function hello($name='thinkphp')
    {
        return 'Hello,' . $name . '!';
    }
}
```

由于默认是输出`Html`输出，所以直接以html页面方式输出响应内容。如果你发起一个JSON请求的话，输出就会自动使用JSON格式响应输出。

为了规范和清晰起见，最佳的方式是在控制器最后明确输出类型（毕竟一个确定的请求是有明确的响应输出类型），默认支持的输出类型包括：

输出类型	快捷方法	对应Response类
HTML输出	response	\think\Response
渲染模板输出	view	\think\response\View
JSON输出	json	\think\response\Json
JSONP输出	jsonp	\think\response\Jsonp
XML输出	xml	\think\response\Xml
页面重定向	redirect	\think\response\Redirect
附件下载	download	\think\response\Download

例如我们需要输出一个JSON数据给客户端（或者AJAX请求），可以使用：

```
<?php
namespace app\controller;

class Index
{
    public function hello()
    {
        $data = ['name' => 'thinkphp', 'status' => '1'];
        return json($data);
    }
}
```

2.6.2 文件下载

支持文件下载功能，可以更简单的读取文件进行下载操作，支持直接下载输出内容。
你可以在控制器的操作方法中添加如下代码：

```
public function download()
{
        $file=  new \think\response\File('image.jpg');
        return $file->name('my.jpg');
        // 或者使用助手函数完成相同的功能
        // download是系统封装的一个助手函数
        return download('image.jpg', 'my.jpg');
  }
// 访问download操作就会下载命名为my.jpg的图像文件。
```

如果需要设置文件下载的有效期，可以使用

```
public function download()
{
        // 设置300秒有效期
        return download('image.jpg', 'my')->expire(300);
}
// 除了expire方法外，还支持下面的方法：
方法                        描述
name                        命名下载文件
expire                      下载有效期
isContent                   是否为内容下载
mimeType                    设置文件的mimeType类型
```

助手函数提供了内容下载的参数，如果需要直接下载内容，可以在第三个参数传入`true`：

```
public function download()
{
    $data = '这是一个测试文件';
    return download($data, 'test.txt', true);
}
```

2.7 响应参数和重定向

2.7.1 响应参数

`Response`对象提供了一系列方法用于设置响应参数，包括设置输出内容、状态码及`header`信息等，并且支持链式调用以及多次调用。

设置数据

```
// Response基类提供了data方法用于设置响应数据。
response()->data($data);
json()->data($data);

// 不过需要注意的是data方法设置的只是原始数据，并不一定是最终的输出数据，
// 最终的响应输出数据是会根据当前的Response响应类型做自动转换的，例如：
json()->data($data);  // 最终的数据就是 json数据
```

设置状态码

`Response`基类提供了`code`方法用于设置响应数据，但大部分情况一般我们是直接在调用助手函数的时候直接传入状态码，例如：

```
json($data,201);
view($data,401);
// 或者在后面链式调用code方法是等效的：
json($data)->code(201);
```

设置头信息

可以使用`Response`类的`header`设置响应的头信息

```
json($data)->code(201)->header([
    'Cache-control' => 'no-cache,must-revalidate'
]);
// 除了header方法之外，Response基类还提供了常用头信息的快捷设置方法：
方法名                  作用
lastModified            设置Last-Modified头信息
expires                 设置Expires头信息
eTag                    设置ETag头信息
cacheControl            设置Cache-control头信息
contentType             设置Content-Type头信息
```

2.7.2 重定向

可以使用`redirect`助手函数进行重定向

```php
<?php
namespace app\controller;

class Index
{
    public function hello()
    {
        return redirect('http://www.thinkphp.cn');
    }
}
```

重定向传参

```php
// 如果是站内重定向的话，可以支持URL组装，有两种方式组装URL，第一种是直接使用完整地址（/打头）
redirect('/index/hello/name/thinkphp');

// 如果你需要自动生成URL地址，应该在调用之前调用url函数先生成最终的URL地址。
redirect(url('hello',['name' => 'think']));
```

记住请求地址

在很多时候，我们重定向的时候需要记住当前请求地址（为了便于跳转回来），我们可以使用remember方法记住重定向之前的请求地址。
下面是一个示例，我们第一次访问index操作的时候会重定向到hello操作并记住当前请求地址，然后操作完成后到restore方法，restore方法则会自动重定向到之前记住的请求地址，完成一次重定向的回归，回到原点！（再次刷新页面又可以继续执行）

```php
<?php
namespace app\controller;

class Index
{
    public function index()
    {
        // 判断session完成标记是否存在
        if (session('?complete')) {
            // 删除session
            session('complete', null);
            return '重定向完成，回到原点!';
        } else {
```

```
            // 记住当前地址并重定向
            return redirect('hello')
                ->with('name', 'thinkphp')
                ->remember();
        }
    }

    public function hello()
    {
        $name = session('name');
        return 'hello,' . $name . '! <br/><a href="/index/index/restore">点击回到来源地址</a>';
    }

    public function restore()
    {
        // 设置session标记完成
        session('complete', true);
        // 跳回之前的来源地址
        return redirect()->restore();
    }
}
```

各位同学，以上就是控制器的全部内容。