第六章 调试与验证

6.1 调试模式

ThinkPHP有专门为开发过程而设置的调试模式，开启调试模式后，会牺牲一定的执行效率，但带来的方便和除错功能非常值得。

应用默认是部署模式，在开发阶段，可以修改环境变量APP_DEBUG开启调试模式，上线部署后切换到部署模式。
本地开发的时候可以在应用根目录下面定义.env文件。

通过`create-project`默认安装的话， 会在根目录自带一个`.example..env`文件，你可以直接更名为`.env`文件。

`.env`文件的定义格式如下：

```
// 设置开启调试模式
APP_DEBUG =  true
// 其它的环境变量设置
// ...
```

调试模式的优势在于：

- 开启日志记录，任何错误信息和调试信息都会详细记录，便于调试；
- 会详细记录整个执行过程；
- 模板修改可以即时生效；
- 通过Trace功能更好的调试和发现错误；
- 发生异常的时候会显示详细的异常信息；

一旦关闭调试模式，发生错误后不会提示具体的错误信息，如果你仍然希望看到具体的错误信息，那么可以在`app.php`文件中如下设置：

```
// 显示错误信息
'show_error_msg'        =>  true,
```

6.2 Trace调试 SQL调试 变量调试

6.2.1 Trace调试

调试模式并不能完全满足我们调试的需要，有时候我们需要手动的输出一些调试信息。除了本身可以借助一些开发工具进行调试外，ThinkPHP还提供了一些内置的调试工具和函数。

`Trace`调试功能就是ThinkPHP提供给开发人员的一个用于开发调试的辅助工具。可以实时显示当前页面或者请求的请求信息、运行情况、SQL执行、错误信息和调试信息等，并支持自定义显示，并且支持没有页面输出的操作调试。最新版本页面Trace功能已经不再内置在核心，但默认安装的时候会自动安装`topthink/think-trace`扩展，所以你可以在项目里面直接使用。

如果部署到服务器的话，你可以通过下面方式安装

```
composer install --no-dev  // 就不会安装页面 Trace
```

安装页面Trace扩展后，如果开启调试模式并且运行后有页面有输出的话，页面右下角会显示ThinkPHP的LOGO：

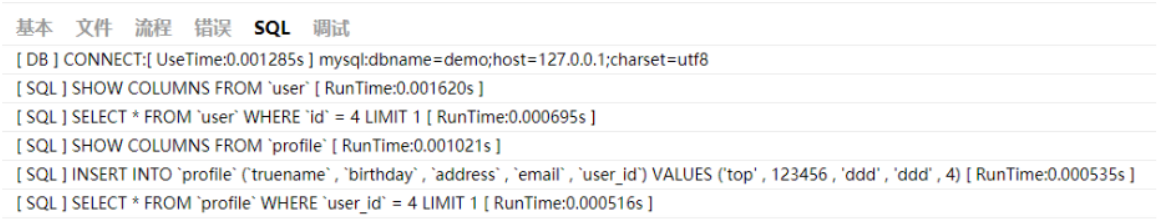
LOGO后面的数字就是当前页面的执行时间（单位是秒） 点击该图标后，会展开详细的Trace信息，如图：

Trace框架有6个选项卡，分别是基本、文件、流程、错误、SQL和调试，点击不同的选项卡会切换到不同的Trace信息窗口。

选项卡	描述
基本	当前页面的基本摘要信息，例如执行时间、内存开销、文件加载数、查询次数等等
文件	详细列出当前页面执行过程中加载的文件及其大小
流程	会列出当前页面执行到的行为和相关流程
错误	当前页面执行过程中的一些错误信息，包括警告错误
SQL	当前页面执行到的SQL语句信息
调试	开发人员在程序中进行的调试输出

6.2.2 SQL调试

通过查看页面Trace信息可以看到当前请求所有执行的SQL语句，例如：



查看sql日志

如果开启了数据库的日志监听（`trigger_sql`）的话，可以在日志文件（或者设置的日志输出类型）中看到详细的SQL执行记录（甚至包含性能分析）。通常我们建议设置把SQL日志级别写入到单独的日志文件中，具体可以参考日志处理部分。

下面是一个典型的SQL日志：

```
[ SQL ] SHOW COLUMNS FROM `think_user` [ RunTime:0.001339s ]
[ SQL ] SELECT * FROM `think_user` LIMIT 1 [ RunTime:0.000539s ]
```

调试执行的sql语句

在模型操作中，为了更好的查明错误，经常需要查看下最近使用的SQL语句，我们可以用`getLastsql`方法来输出上次执行的sql语句。例如：

```
User::find(1);
echo User::getLastSql();    // getLastSql方法只能获取最后执行的SQL记录。
```

输出结果是

```
SELECT * FROM 'think_user' WHERE `id` = 1
```

6.2.3 变量调试

输出某个变量是开发过程中经常会用到的调试方法，除了使用php内置的`var_dump`和`print_r`之外，ThinkPHP框架内置了一个对浏览器友好的`dump`方法，用于输出变量的信息到浏览器查看。

用法和PHP内置的`var_dump`一致：

```
dump($var1, ...$varN)
```

使用示例：

```
$blog = Db::name('blog')->where('id', 3)->find();
$user = User::find();
dump($blog, $user);
```

如果需要在调试变量输出后中止程序的执行，可以使用`halt`函数，例如：

```
$blog = Db::name('blog')->where('id', 3)->find();
$user = User::find();
halt($blog, $user);
echo '这里的信息是看不到的';
```

6.3 验证器

6.3.1 验证器定义

为具体的验证场景或者数据表定义好验证器类，直接调用验证类的`check`方法即可完成验证，下面是一个例子：
我们定义一个`\app\validate\User`验证器类用于`User`的验证。可以直接在验证器类中使用`message`属性定义错误提示信息。

```
namespace app\validate;

use think\Validate;

class User extends Validate
{
    protected $rule =   [
        'name'  => 'require|max:25',
        'age'   => 'number|between:1,120',
        'email' => 'email',
    ];

    protected $message  =   [
        'name.require' => '名称必须',
        'name.max'     => '名称最多不能超过25个字符',
        'age.number'   => '年龄必须是数字',
        'age.between'  => '年龄只能在1-120之间',
        'email'        => '邮箱格式错误',
    ];

}
```

6.3.2 数据验证

在需要进行`User`验证的控制器方法中，添加如下代码即可：

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

use app\validate\User;
use think\exception\ValidateException;

class Index
{
    public function index()
    {
        try {
            validate(User::class)->check([
                'name'  => 'thinkphp',
                'email' => 'thinkphp@qq.com',
            ]);
        } catch (ValidateException $e) {
            // 验证失败 输出错误信息
            dump($e->getError());
        }
    }
}
```

6.3.3 自定义验证规则

系统内置了一些常用的规则（参考后面的内置规则），如果不能满足需求，可以在验证器重添加额外的验证方法，例如：

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

use think\Validate;

class User extends Validate
{
    protected $rule = [
        'name'  =>  'checkName:thinkphp',
        'email' =>  'email',
    ];

    protected $message = [
        'name'  =>  '用户名必须',
        'email' =>  '邮箱格式错误',
    ];

    // 自定义验证规则
    protected function checkName($value, $rule, $data=[])
    {
        return $rule == $value ? true : '名称错误';
    }
}
```

6.4 验证规则和错误信息

6.4.1 验证规则

验证规则的定义通常有两种方式，如果你使用了验证器的话，通常通过`rule`属性定义验证规则，而如果使用的是独立验证的话，则是通过`rule`方法进行定义。

属性定义

属性定义方式仅限于验证器，通常类似于下面的方式：

```
<?php
namespace app\validate;

use think\Validate;

class User extends Validate
{
    protected $rule = [
      'name'  => 'require|max:25',
      'age'   => 'number|between:1,120',
      'email' => 'email',
    ];

}
```

因为一个字段可以使用多个验证规则（如上面的`age`字段定义了`number`和`between`两个验证规则），在一些特殊的情况下，为了避免混淆可以在`rule`属性中使用数组定义规则。

```
<?php
namespace app\validate;

use think\Validate;

class User extends Validate
{
    protected $rule = [
      'name'  => ['require', 'max' => 25, 'regex' => '/^[\w|\d]\w+/'],
      'age'   => ['number', 'between' => '1,120'],
      'email' => 'email',
    ];

}
```

方法定义

如果使用的是独立验证（即手动调用验证类进行验证）方式的话，通常使用`rule`方法进行验证规则的设置，举例说明如下。独立验证通常使用`Facade`或者自己实例化验证类。

```
$validate = \think\facade\Validate::rule('age', 'number|between:1,120')
->rule([
    'name'  => 'require|max:25',
    'email' => 'email'
]);

$data = [
    'name'  => 'thinkphp',
    'email' => 'thinkphp@qq.com'
```

```
];

if (!$validate->check($data)) {
    dump($validate->getError());
}
```

6.4.2 错误信息

如果没有定义任何的验证提示信息，系统会显示默认的错误信息，如果要输出自定义的错误信息，可以定义messgae属性。

```
namespace app\validate;

use think\Validate;

class User extends Validate
{
    protected $rule = [
      'name'  => 'require|max:25',
      'age'   => 'number|between:1,120',
      'email' => 'email',
    ];

    protected $message = [
      'name.require' => '名称必须',
      'name.max'     => '名称最多不能超过25个字符',
      'age.number'   => '年龄必须是数字',
      'age.between'  => '年龄必须在1~120之间',
      'email'        => '邮箱格式错误',
    ];
}
```

错误信息可以支持数组定义，并且通过JSON方式传给前端。

```
namespace app\validate;

use think\Validate;

class User extends Validate
{
    protected $rule = [
      'name'  => 'require|max:25',
      'age'   => 'number|between:1,120',
      'email' => 'email',
    ];

    protected $message = [
      'name.require' => ['code' => 1001, 'msg' => '名称必须'],
      'name.max'     => ['code' => 1002, 'msg' => '名称最多不能超过25个字符'],
      'age.number'   => ['code' => 1003, 'msg' => '年龄必须是数字'],
      'age.between'  => ['code' => 1004, 'msg' => '年龄必须在1~120之间'],
      'email'        => ['code' => 1005, 'msg' =>'邮箱格式错误'],
    ];
}
```

6.5 验证场景和路由验证

6.5.1 验证场景

验证器支持定义场景，并且验证不同场景的数据，例如：

```
namespace app\validate;

use think\Validate;

class User extends Validate
{
    protected $rule =   [
        'name'  => 'require|max:25',
        'age'   => 'number|between:1,120',
        'email' => 'email',
    ];

    protected $message  =   [
        'name.require' => '名称必须',
        'name.max'     => '名称最多不能超过25个字符',
        'age.number'   => '年龄必须是数字',
        'age.between'  => '年龄只能在1-120之间',
        'email'        => '邮箱格式错误',
    ];

    protected $scene = [
        'edit'  =>  ['name','age'],
    ];
}
```

然后可以在验证方法中制定验证的场景

```
$data = [
    'name'  => 'thinkphp',
    'age'   => 10,
    'email' => 'thinkphp@qq.com',
];

try {
    validate(app\validate\User::class)
        ->scene('edit')
        ->check($data);
catch (ValidateException $e) {
    // 验证失败 输出错误信息
    dump($e->getError());
}
```

可以单独为某个场景定义方法（方法的命名规范是`scene`+场景名），并且对某些字段的规则重新设置，例如：

- 注意：场景名不区分大小写，且在调用的时候不能将驼峰写法转为下划线

```
namespace app\validate;

use think\Validate;

class User extends Validate
```

```
{
    protected $rule =   [
        'name'  => 'require|max:25',
        'age'   => 'number|between:1,120',
        'email' => 'email',
    ];

    protected $message  =   [
        'name.require' => '名称必须',
        'name.max'     => '名称最多不能超过25个字符',
        'age.number'   => '年龄必须是数字',
        'age.between'  => '年龄只能在1-120之间',
        'email'        => '邮箱格式错误',
    ];

    // edit 验证场景定义
    public function sceneEdit()
    {
        return $this->only(['name','age'])
            ->append('name', 'min:5')
            ->remove('age', 'between')
            ->append('age', 'require|max:100');
    }
}
```

主要方法说明如下：

方法名	描述
only	场景需要验证的字段
remove	移除场景中的字段的部分验证规则
append	给场景中的字段需要追加验证规则

如果对同一个字段进行多次规则补充（包括移除和追加），必须使用下面的方式：

```
remove('field', ['rule1','rule2'])
// 或者
remove('field', 'rule1|rule2')
```

下面的方式会导致rule1规则remove不成功

```
remove('field', 'rule1')
->remove('field', 'rule2')
```

6.5.2 路由验证

可以在路由规则定义的时候调用`validate`方法指定验证器类对请求的数据进行验证。
例如下面的例子表示对请求数据使用验证器类`app\validate\User`进行自动验证，并且使用`edit`验证场景：

```
Route::post('hello/:id', 'index/hello')
    ->validate(\app\validate\User::class,'edit');
```

或者不使用验证器而直接传入验证规则

```
Route::post('hello/:id', 'index/hello')
    ->validate([
        'name'  =>  'min:5|max:50',
        'email' =>  'email',
    ]);
```

也支持使用对象化规则定义

```
Route::post('hello/:id', 'index/hello')
    ->validate([
        'name'  =>  ValidateRule::min(5)->max(50),
        'email' =>  ValidateRule::isEmail(),
    ]);
```

6.6 内置规则

系统内置了一些常用的验证规则，可以完成大部分场景的验证需求，包括(这里只列出常用的):

格式验证类

require 验证某个字段必须

```
'name'=>'require'
```

number 验证某个字段的值是否为纯数字（采用`ctype_digit`验证，不包含负数和小数点）

```
'num'=>'number'
```

integer 验证某个字段的值是否为整数（采用`filter_var`验证）

```
'num'=>'integer'
```

email 验证某个字段的值是否为email地址（采用`filter_var`验证），例如:

```
'email'=>'email'
```

date 验证值是否为有效的日期

```
'date'=>'date'
```

url 验证某个字段的值是否为有效的URL地址（采用`filter_var`验证），例如:

```
'url'=>'url'
```

长度和区间验证类

in 验证某个字段的值是否在某个范围，例如:

```
'num'=>'in:1,2,3'
```

notIn 验证某个字段的值不在某个范围，

```
'num'=>'notIn:1,2,3'
```

length:num1,num2 验证某个字段的值的长度是否在某个范围

```
'name'=>'length:4,25'
```

字段比较类

confirm 验证某个字段是否和另外一个字段的值一致

```
'repassword'=>'require|confirm:password'
```

支持字段自动匹配验证规则，如`password`和`password_confirm`是自动相互验证的，只需要使用

```
'password'=>'require|confirm'
```

different 验证某个字段是否和另外一个字段的值不一致

```
'name'=>'require|different:account'
```

上传验证

```
file
验证是否是一个上传文件

image:width,height,type
验证是否是一个图像文件，width height和type都是可选，width和height必须同时定义。

fileExt:允许的文件后缀
验证上传文件后缀

fileMime:允许的文件类型
验证上传文件类型

fileSize:允许的文件字节大小
验证上传文件大小
```

6.7 表单令牌

添加令牌 token

验证规则支持对表单的令牌验证，首先需要在你的表单里面增加下面隐藏域：

```
<input type="hidden" name="__token__" value="{:token()}" />
```

也可以直接使用

```
{:token_field()}
```

默认的令牌Token名称是`__token__`，如果需要自定义名称及令牌生成规则可以使用

```
{:token_field('__hash__', 'md5')}
```

第二个参数表示token的生成规则，也可以使用闭包。
如果你没有使用默认的模板引擎，则需要自己生成表单隐藏域

```
namespace app\controller;

use think\Request;
use think\facade\View;

class Index
{
    public function index(Request $request)
    {
        $token = $request->buildToken('__token__', 'sha1');
        View::assign('token', $token);
        return View::fetch();
    }
}
```

然后在模板表单中使用：

```
<input type="hidden" name="__token__" value="{$token}" />
```

AJAX提交

如果是AJAX提交的表单，可以将`token`设置在`meta`中

```
<meta name="csrf-token" content="{:token()}">
```

或者直接使用

```
{:token_meta()}
```

然后在全局Ajax中使用这种方式设置`X-CSRF-Token`请求头并提交：

```
$.ajaxSetup({
    headers: {
        'X-CSRF-TOKEN': $('meta[name="csrf-token"]').attr('content')
    }
});
```

路由验证

然后在路由规则定义中，使用

```
Route::post('blog/save','blog/save')->token();
```

如果自定义了`token`名称，需要改成

```
Route::post('blog/save','blog/save')->token('__hash__');
```

令牌检测如果不通过，会抛出`think\exception\ValidateException`异常。

控制器验证

如果没有使用路由定义，可以在控制器里面手动进行令牌验证

```
namespace app\controller;

use think\exception\ValidateException;
use think\Request;

class Index
{
    public function index(Request $request)
    {
        $check = $request->checkToken('__token__');

        if(false === $check) {
            throw new ValidateException('invalid token');
        }

        // ...
    }
}
```

提交数据默认获取`post`数据，支持指定数据进行`Token`验证。

```
namespace app\controller;

use think\exception\ValidateException;
use think\Request;

class Index
{
    public function index(Request $request)
    {
        $check = $request->checkToken('__token__', $request->param());

        if(false === $check) {
            throw new ValidateException('invalid token');
        }

        // ...
    }
}
```

使用验证器验证

在你的验证规则中，添加`token`验证规则即可，例如，如果使用的是验证器的话，可以改为：

```
protected $rule = [
        'name'  =>  'require|max:25|token',
        'email' =>  'email',
    ];
```

如果你的令牌名称不是`__token__`（假设是`__hash__`），验证器中需要改为：

```
protected $rule = [
        'name'  =>  'require|max:25|token:__hash__',
        'email' =>  'email',
    ];
```

各位同学，以上就是全部内容。