零、序言

< ThinkPHP官方出品，入门TP5必读系列（一） >

概述

本系列入门文档版权归ThinkPHP官方所有，未经授权，禁止任何方式转载和下载，侵权必究！

ThinkPHP5.0版本的优势在于：

* 更灵活的路由；
* 依赖注入；
* 请求缓存；
* 更强大的查询语法；
* 引入了请求/响应对象；
* 路由地址反解生成；
* 增强的模型功能；
* API开发友好；
* 改进的异常机制；
* 远程调试支持；
* 单元测试支持；
* 命令行工具；
* Composer支持；

**本快速入门系列**是官方出品的学习和掌握ThinkPHP5.0不可多得的入门指引教程和**标准参照**，针对新手用户给出了较易理解的使用。

本系列围绕WEB开发和API开发常用的一系列基础功能进行循序渐进的讲解。推荐在看完和理解快速入门之后，再去通读官方的完全开发手册，会更容易理解。

希望支持ThinkPHP5的用户能够到[Github](https://github.com/top-think/think)给我们一个star^\_^

目前章节完成情况（√表示已经完成）

* 第零章、序言 √
* 第一章、基础 √
* 第二章、URL和路由 √
* 第三章、请求和响应 √
* 第四章、数据库 √
* 第五章、查询语言 √
* 第六章、模型和关联 √
* 第七章、视图和模板 √
* 第八章、调试大法 √
* 第九章、API开发 √
* 第十章、命令行工具 √
* 第十一章、扩展√
* 第十二章、杂项√
* 番外篇：学习ThinkPHP5的正确姿势√
* 概念篇：ThinkPHP5名词解释√
* 附录A、常见问题集（更新中）
* 附录B、3.2和5.0区别√
* 附录C、助手函数√
* 附录D、5.1你必须努力避免的一些问题

阅读须知

建议阅读本书之前首先掌握一些基础知识和概念，包括但不限于：

* 掌握HTML/PHP基础知识；
* 熟悉PHP环境部署；
* 了解命名空间概念；
* 了解面向对象编程；
* 了解GIT和版本库；
* 了解PHP依赖管理composer；

要确保学习示例的效果，请确保你使用的是最新的5.0正式版本。

由于编写过程中5.0版本也在不断完善，本快速入门系列会保持更新，涉及到的内容和示例，以及用户反馈的勘误也会随时进行修订。

陆续加入了5.1版本的更新，以下标识的段落表示为5.1版本适用。

【 5.1 】使用须知

这里表示只有5.1才适用的内容

# 一、基础

# 快速入门（一）：基础

本章介绍了ThinkPHP5.0的安装及基本使用，并给出了一个最简单的示例带你了解如何开始开发，主要包含：

#### 在学习ThinkPHP5.0之前，如果你还不理解[面向对象](http://php.net/manual/zh/language.oop5.php)和[命名空间](http://php.net/manual/zh/language.namespaces.php)的概念，建议首先去PHP手册恶补下PHP的相关基础知识，否则将成为你学习5.0的最大障碍。

（1）简介

简介

ThinkPHP是一个快速、简单的基于MVC和面向对象的轻量级PHP开发框架，遵循Apache2开源协议发布，自2006年诞生以来一直秉承简洁实用的设计原则，在保持出色的性能和至简代码的同时，尤其注重开发体验和易用性，并且拥有众多的原创功能和特性，为WEB应用和API开发提供了强有力的支持。

相当于以往版本，ThinkPHP5.0版本是一个颠覆和重构版本，也是ThinkPHP十周年献礼版本，基于PHP5.4设计（完美支持PHP7），采用全新的架构思想，引入了很多的PHP新特性，优化了核心，减少了依赖，支持Composer，实现了真正的惰性加载，并且为API开发做了深入的支持，在功能、性能以及灵活性方面都较为突出。

# （2）安装

## 安装

ThinkPHP提供多个方式的安装，包括官网下载、Composer安装以及GIT下载，对于新手来说，有必要理解这几种安装方式的区别。

* **官网下载**：一般都是稳定版本（并不会实时更新），有些大的版本还会提供核心版（不含扩展）和完整版（包含常用扩展）两个版本；
* **Composer安装**：是一种主流的安装方式，Composer也分为稳定版和开发版安装，如果是安装的稳定版则可以update到最新的稳定版，如果是安装的开发版那么update到的也是实时的开发版，安装慢的可以使用国内镜像，但注意存在一定的缓存时间；
* **GIT安装**：直接通过git地址安装的方式，优势是可以实时更新，跟着官方开发版本走的用户可以选择GIT更新，也方便及时反馈和提交PR，除了Github之外，国内的码云和Coding代码托管平台都有ThinkPHP5的镜像，请自行选择。

### 官网下载

ThinkPHP最新的稳定版本可以在（[官方网站下载页](http://thinkphp.cn/down/framework.html)）下载，不过官网下载版本并不是实时更新的，我们会在每个版本更新发布的时候重新打包，如果你需要实时更新版本，请使用Git版本库或者Composer安装。

### 【 5.1 】

5.1版本开始官方不再提供打包下载版本，请选择Composer或者GIt安装

### Composer安装和更新

ThinkPHP5.0支持使用Composer安装和更新，如果还没有安装 Composer，你可以按 [Composer安装](http://www.kancloud.cn/thinkphp/composer/35669) 中的方法安装。在 Linux和 Mac OS X 中可以运行如下命令：

curl -sS https://getcomposer.org/installer | php

mv composer.phar /usr/local/bin/composer

在 Windows 中，你需要下载并运行 [Composer-Setup.exe](https://getcomposer.org/Composer-Setup.exe)。

### 提示：

如果遇到任何问题或者想更深入地学习 Composer，请参考 [Composer 官方文档（英文）](https://getcomposer.org/doc/)，看云上有 [Composer 的中文版本](http://www.kancloud.cn/thinkphp/composer)。

如果你已经安装有 Composer 请确保使用的是最新版本，或者可以用 composer self-update 命令更新为最新版本。

由于众所周知的原因，国外的网站连接速度很慢，并且随时可能被“墙”甚至“不存在”。因此安装的时间可能会比较长，我们建议通过下面的方式使用国内镜像。

打开命令行窗口（windows用户）或控制台（Linux、Mac 用户）并执行如下命令：

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

注意：最近phpcomposer镜像存在问题，可以改成

composer config -g repo.packagist composer https://packagist.laravel-china.org

然后在命令行下面，切换到你的web根目录下面并执行下面的命令：

composer create-project topthink/think=5.0.\* tp5 --prefer-dist

### 【 5.1 】使用须知

5.1版本安装使用下面的指令即可

composer create-project topthink/think tp5 --prefer-dist

不过注意，如果你的PHP版本低于5.6的话 将仍然安装5.0版本

如果出现错误提示，请根据提示操作或者参考[Composer中文文档](http://www.kancloud.cn/thinkphp/composer)。

如果之前使用Composer安装的话，首先切换到你的tp5目录，然后使用下面的命令更新框架到最新版本（注意因为缓存关系，composer不一定是及时更新的）：

composer update

### 注意：

使用composer update更新核心框架会清空原来的核心框架目录，如果你定制了核心框架或者添加了扩展在核心目录的话，建议使用后面的Git方式更新。

### Git下载和更新

ThinkPHP使用Git版本库进行更新迭代，如果你不太了解Composer或者觉得Composer太慢，也可以使用git版本库安装和更新，ThinkPHP5.0拆分为多个仓库，下面是github（主要维护仓库）及国内的仓库地址：

#### [ Github ]

应用项目：https://github.com/top-think/think

核心框架：https://github.com/top-think/framework

#### [ 码云 ]

应用项目：https://gitee.com/liu21st/thinkphp5.git

核心框架：https://gitee.com/liu21st/framework.git

#### [ Coding ]

应用项目：https://git.coding.net/liu21st/thinkphp5.git

核心框架：https://git.coding.net/liu21st/framework.git

### 【 5.1 】使用须知

5.0版本和5.1版本分别属于不同的分支，5.1需要switch到5.1分支

### 提示：

之所以设计为应用和核心仓库分离，是为了支持Composer单独更新核心框架。

如果你还没安装Git，可以参考阅读 [Pro Git第二版（中文）](http://www.kancloud.cn/kancloud/progit)先。

首先克隆下载应用项目仓库

git clone https://github.com/top-think/think tp5

然后切换到tp5目录下面，再克隆核心框架仓库：

git clone https://github.com/top-think/framework thinkphp

两个仓库克隆完成后，就完成了ThinkPHP5.0的Git方式下载，如果需要更新核心框架的时候，只需要切换到thinkphp核心目录下面，然后执行：

git pull https://github.com/top-think/framework

如果不熟悉git命令行，可以使用任何一个GIT客户端进行操作，在此不再详细说明。

Github上还给新手提供了包含了常用扩展的完整版，可以免去composer安装的烦恼，下载地址：<https://github.com/top-think/think/tree/archive>

GIT方式只能安装核心框架，官方扩展只能通过Composer安装。

（3）目录结构

目录结构

通过Composer安装后（或者下载后的压缩文件解压后）可以看到下面的目录结构：

tp5

├─application 应用目录

├─extend 扩展类库目录（可定义）

├─public 网站对外访问目录

├─runtime 运行时目录（可定义）

├─vendor 第三方类库目录（Composer）

├─thinkphp 框架核心目录

├─build.php 自动生成定义文件（参考）

├─composer.json Composer定义文件

├─LICENSE.txt 授权说明文件

├─README.md README 文件

├─think 命令行工具入口

【 5.1 】使用须知

5.1由于配置和路由目录都独立出来了（不再纳入应用目录），所以目录结构增加了config和route，安装后可以比对下

注意：

如果在linux环境下面的话，需要给runtime目录755权限。

有几个关键的路径先了解下：

| **目录** | **说明** | **常量** |
| --- | --- | --- |
| tp5 | 项目根目录 | ROOT\_PATH |
| tp5/application | 应用目录 | APP\_PATH |
| tp5/thinkphp | 框架核心目录 | THINK\_PATH |
| tp5/extend | 应用扩展目录 | EXTEND\_PATH |
| tp5/vendor | Composer扩展目录 | VENDOR\_PATH |

【 5.1 】使用须知

5.1版本取消了所有的系统常量，改为环境变量获取  
获取方式为： think\facade\Env::get('环境变量名')

| **目录** | **说明** | **环境变量** |
| --- | --- | --- |
| tp5 | 项目根目录 | root\_path |
| tp5/application | 应用目录 | app\_path |
| tp5/thinkphp | 框架核心目录 | think\_path |
| tp5/extend | 应用扩展目录 | extend\_path |
| tp5/vendor | Composer扩展目录 | vendor\_path |

核心框架目录的结构如下：

├─thinkphp 框架系统目录

│ ├─lang 语言包目录

│ ├─library 框架核心类库目录

│ │ ├─think think 类库包目录

│ │ └─traits 系统 traits 目录

│ ├─tpl 系统模板目录

│ │

│ ├─.htaccess 用于 apache 的重写

│ ├─.travis.yml CI 定义文件

│ ├─base.php 框架基础文件

│ ├─composer.json composer 定义文件

│ ├─console.php 控制台入口文件

│ ├─convention.php 惯例配置文件

│ ├─helper.php 助手函数文件（可选）

│ ├─LICENSE.txt 授权说明文件

│ ├─phpunit.xml 单元测试配置文件

│ ├─README.md README 文件

│ └─start.php 框架引导文件

【 5.1 】使用须知

5.1版本的start.php文件和console.php已经移出核心框架放入应用包。

# （4）运行环境

## 运行环境

ThinkPHP5.0的环境要求如下：

* PHP >= 5.4.0 （完美支持PHP7）
* PDO PHP Extension
* MBstring PHP Extension
* CURL PHP Extension

### 【 5.1 】使用须知

5.1版本最低要求PHP5.6+

在开始之前，你需要一个Web服务器和\*\*PHP5.4+\*\*运行环境，如果你暂时还没有，我们推荐使用集成开发环境[WAMPServer](http://www.wampserver.com/en/)（Windows系统下集成Apache、PHP和MySQL的服务套件）来使用ThinkPHP进行本地开发和测试，最新版本的WAMP在[这里下载](https://sourceforge.net/projects/wampserver/files/WampServer%203/)。

如果你不想安装任何WEB服务器，也可以直接使用PHP自带的WebServer，并且运行router.php来运行测试。

我们进入命令行，进入tp5/public目录后，输入如下命令：

php -S localhost:8888 router.php

如果提示错误，请检查你的php.exe所在路径是否已经加入系统环境变量Path。

接下来可以直接访问

http://localhost:8888

注意： S 一定要大写，端口号可以随意设置，只要和已有的不冲突，如果要停止服务，直接在命令行下面按CTRL+C 即可退出。

（5）入口文件

入口文件

ThinkPHP5.0版本的默认自带的入口文件位于public/index.php（实际部署的时候public目录为你的应用对外访问目录），入口文件内容如下：

// 定义应用目录

define('APP\_PATH', \_\_DIR\_\_ . '/../application/');

// 加载框架引导文件

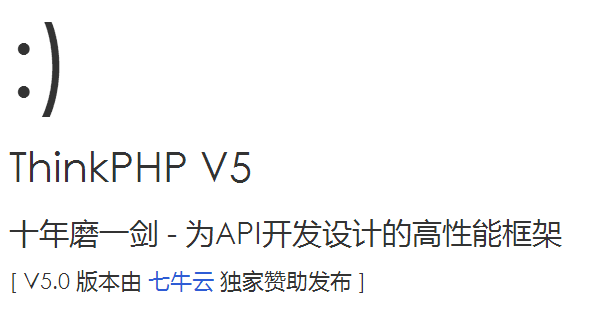
require \_\_DIR\_\_ . '/../thinkphp/start.php';

这段代码的作用就是定义应用目录APP\_PATH和加载ThinkPHP框架的入口文件，这是所有基于ThinkPHP开发应用的第一步。

我们可以在浏览器中访问入口文件

http://localhost/tp5/public/

运行后我们会看到欢迎页面：



官方提供的默认应用的实际目录结构和说明如下：

├─application 应用目录（可设置）

│ ├─index 模块目录(可更改)

│ │ ├─config.php 模块配置文件

│ │ ├─common.php 模块公共文件

│ │ ├─controller 控制器目录

│ │ ├─model 模型目录

│ │ └─view 视图目录

│ │

│ ├─command.php 命令行工具配置文件

│ ├─common.php 应用公共文件

│ ├─config.php 应用配置文件

│ ├─tags.php 应用行为扩展定义文件

│ ├─database.php 数据库配置文件

│ └─route.php 路由配置文件

5.0版本采用模块化的设计架构，默认的应用目录下面只有一个index模块目录，如果我要添加新的模块可以使用控制台命令来生成。

切换到命令行模式下，进入到应用根目录并执行如下指令：

php think build --module demo

就会生成一个默认的demo模块，包括如下目录结构：

├─demo

│ ├─controller 控制器目录

│ ├─model 模型目录

│ ├─view 视图目录

│ ├─config.php 模块配置文件

│ └─common.php 模块公共文件

同时也会生成一个默认的Index控制器文件。

注意：这只是一个初始默认的目录结构，在实际的开发过程中可能需要创建更多的目录和文件。

在后面的示例中，为了方便访问，我们设置vhost访问，以apache为例的话定义如下：

<VirtualHost \*:80>

DocumentRoot "/home/www/tp5/public"

ServerName tp5.com

</VirtualHost>

把DocumentRoot修改为你本机tp5/public所在目录，并注意修改本机的hosts文件把tp5.com指向本地127.0.0.1。

如果你暂时不想设置vhost或者还不是特别了解如何设置，可以先把入口文件移动到框架的ROOT\_PATH目录，并更改入口文件中的APP\_PATH和框架入口文件的位置（这里顺便展示下如何更改相关目录名称），index.php文件内容如下：

// 定义应用目录为apps

define('APP\_PATH', \_\_DIR\_\_ . '/apps/');

// 加载框架引导文件

require \_\_DIR\_\_ . '/think/start.php';

这样最终的应用目录结构如下：

tp5

├─index.php 应用入口文件

├─apps 应用目录

├─public 资源文件目录

├─runtime 运行时目录

└─think 框架目录

实际的访问URL变成了

http://localhost/tp5/

提示：

如非特别说明，我们后面的示例均以tp5.com进行访问，如果你使用了其它的方式请自行修改。

【 5.1 】使用须知

5.1版本的入口文件改变了5.0引入框架的start.php文件的方式，而是直接在入口文件中引入base.php文件。

如果要改变入口文件的位置，只需要把入口文件改为：

<?php

namespace think;

// 定义应用目录

define('APP\_PATH', \_\_DIR\_\_ . '/app/');

// 加载框架基础引导文件

require \_\_DIR\_\_ . '/thinkphp/base.php';

// 执行应用并响应

Container::get('app')

->path(APP\_PATH)

->run()

->send();

（6）资源访问

资源访问

网站的资源文件访问不会影响正常的操作访问，只有当访问的资源文件不存在的时候才会解析到入口文件，一般就会提示模块不存在的错误。

网站的资源文件一般放入public目录的子目录下面，例如下面是一个建议规范：

public

├─index.php 应用入口文件

├─static 静态资源目录

│ ├─css 样式目录

│ ├─js 脚本目录

│ └─img 图像目录

记住，千万不要在public目录之外的任何位置放置资源文件，包括application目录。

访问资源文件的URL路径是：

http://tp5.com/static/css/style.css

http://tp5.com/static/js/common.js

http://tp5.com/static/img/picture.jpg

如果你没有设置域名绑定，而是使用子目录方式访问的话，那么可能的资源访问地址是：

http://localhost/public/static/css/style.css

http://localhost/public/static/js/common.js

http://localhost/public/static/img/picture.jpg

网站的入口文件就是资源文件的起始位置，如果你的入口文件不是在public目录下面的话，还需要自行调整。  
如果不清楚当前的入口文件位置，可以使用phpinfo()在页面输出中查看DOCUMENT\_ROOT的值。

（7）调试模式

调试模式

ThinkPHP支持调试模式，默认情况下是开启状态（5.0.10+版本开始，默认关闭调试模式，需要自己开启）。调试模式以除错方便优先，而且在异常的时候可以显示尽可能多的信息，所以对性能有一定的影响。

调试模式不支持模块单独开启，只能应用全局开启

我们强烈建议开发者在使用ThinkPHP开发的过程中使用调试模式，5.0默认情况下可以捕获到任何细微的错误并抛出异常，这样可以更好的获取错误提示和避免一些问题和隐患，**不要畏惧错误，要勇敢面对，并消除隐患**。

开发完成后，我们实际进行项目部署的时候，修改应用配置文件（application/config.php）中的app\_debug配置参数：

// 关闭调试模式

'app\_debug' => false,

为了安全考虑，避免泄露你的服务器WEB目录信息等资料，一定记得正式部署的时候关闭调试模式。

【 5.1 】使用须知

5.1版本的调试模式设置建议使用环境变量（开发环境则在.env文件中设置）  
如果没有设置任何环境变量，则读取app\_debug配置。  
环境变量设置如下：

APP\_DEBUG = true

（8）控制器

控制器

我们找到index模块的Index控制器（文件位于application/index/controller/Index.php 注意大小写），我们把Index控制器类的index方法修改为Hello,World！。

<?php

namespace app\index\controller;

class Index

{

public function index()

{

return 'Hello,World！';

}

}

提示：

根据类的命名空间可以快速定位文件位置，在ThinkPHP5.0的规范里面，命名空间其实对应了文件的所在目录，app命名空间通常代表了文件的起始目录为application，而think命名空间则代表了文件的起始目录为thinkphp/library/think，后面的命名空间则表示从起始目录开始的子目录。

我们访问URL地址

http://tp5.com

就会看到Hello,World！的输出结果。

如果要访问一个驼峰命名的控制器，例如我们把上面的例子改成一个HelloWorld控制器。

<?php

namespace app\index\controller;

class HelloWorld

{

public function index($name = 'World')

{

return 'Hello,' . $name . '！';

}

}

默认情况下正确的方法是使用下面的URL进行访问

http://tp5.com/index.php/index/hello\_world

下面的访问地址是错误的

http://tp5.com/index.php/index/HelloWorld



因为默认的URL访问是不区分大小写的，全部都会转换为小写的控制器名，除非你在应用配置文件中，设置了关闭url自动转换如下：

'url\_convert' => false,

那么就可以正常访问

http://tp5.com/index.php/index/HelloWorld

如果要继承一个公共的控制器类，可以使用：

<?php

namespace app\index\controller;

use app\index\controller\Base;

class Index extends Base

{

public function index()

{

return 'Hello,World！';

}

}

可以为操作方法定义参数，例如：

<?php

namespace app\index\controller;

class Index

{

public function index($name = 'World')

{

return 'Hello,' . $name . '！';

}

}

当我们带name参数访问入口文件地址（例如 http://tp5.com?name=ThinkPHP）的时候，在浏览器中可以看到如下输出：

Hello,ThinkPHP！

控制器类可以包括多个操作方法，但如果你的操作方法是protected或者private类型的话，是无法直接通过URL访问到该操作的，也就是说只有public类型的操作方法才是可以通过URL访问的。

我们来验证下，把Index控制器类的方法修改为：

<?php

namespace app\index\controller;

class Index

{

public function hello()

{

return 'hello,thinkphp!';

}

public function test()

{

return '这是一个测试方法!';

}

protected function hello2()

{

return '只是protected方法!';

}

private function hello3()

{

return '这是private方法!';

}

}

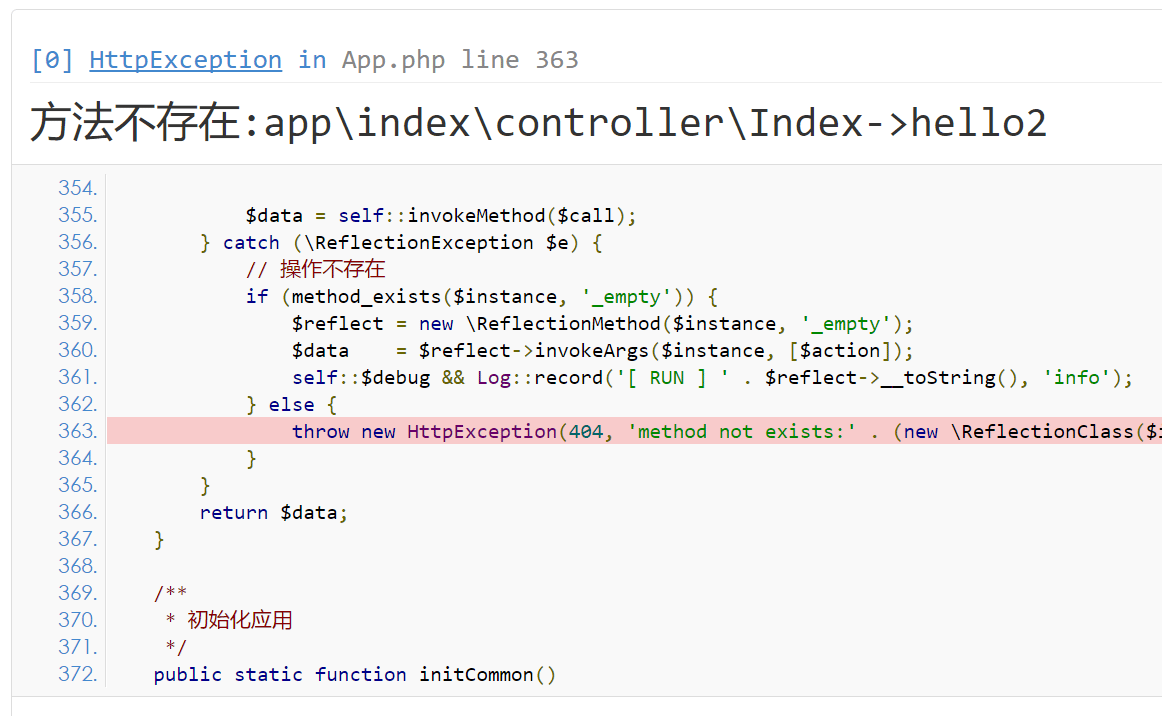
当我们访问如下URL地址的时候，前面两个是正常访问，后面两个则会显示异常。

http://tp5.com/index.php/index/index/hello

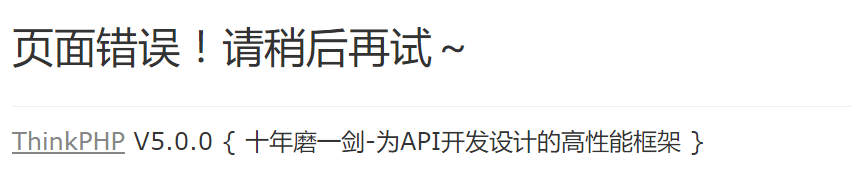
http://tp5.com/index.php/index/index/test

http://tp5.com/index.php/index/index/hello2

http://tp5.com/index.php/index/index/hello3

当我们访问hello2和hello3操作方法后的结果都会显示类似的异常信息：  


异常页面包含了详细的错误信息，是因为开启了调试模式，如果关闭调试模式的话，看到的默认信息如下：



（9）视图

视图

现在我们在给控制器添加视图文件功能，我们在application/index目录下面创建一个view目录，然后添加模板文件view/index/hello.html（注意大小写），我们添加模板内容如下：

<html>

<head>

<title>hello {$name}</title>

</head>

<body>

hello, {$name}!

</body>

</html>

要输出视图，必须在控制器方法中进行模板渲染输出操作，现在修改控制器类如下：

<?php

namespace app\index\controller;

use think\Controller;

class Index extends Controller

{

public function hello($name = 'thinkphp')

{

$this->assign('name', $name);

return $this->fetch();

}

}

[ 新手须知 ]

这里使用了use来导入一个命名空间的类库，然后可以在当前文件中直接使用该别名而不需要使用完整的命名空间路径访问类库。也就说，如果没有使用

use think\Controller;

就必须使用

class Index extends \think\Controller

这种完整命名空间方式。

在后面的内容中，如果我们直接调用系统的某个类的话，都会假设已经在类的开头使用use进行了别名导入。

注意，Index控制器类继承了 think\Controller类之后，我们可以直接使用封装好的assign和fetch方法进行模板变量赋值和渲染输出。

fetch方法中我们没有指定任何模板，所以按照系统默认的规则（**视图目录/控制器/操作方法**）输出了view/index/hello.html模板文件。

接下来，我们在浏览器访问

http://tp5.com/index.php/index/index/hello

输出：

hello,thinkphp!

（10）读取数据

读取数据

在开始之前，我们首先在数据库demo中创建一个think\_data数据表（这里以mysql数据库为例）：

CREATE TABLE IF NOT EXISTS `think\_data`(

`id` int(8) unsigned NOT NULL AUTO\_INCREMENT,

`data` varchar(255) NOT NULL,

PRIMARY KEY (`id`)

) ENGINE=MyISAM DEFAULT CHARSET=utf8 ;

INSERT INTO `think\_data`(`id`,`data`) VALUES

(1,'thinkphp'),

(2,'php'),

(3,'framework');

首先我们需要在应用的数据库配置文件application/database.php中添加数据库的连接信息如下：

return [

// 数据库类型

'type' => 'mysql',

// 服务器地址

'hostname' => '127.0.0.1',

// 数据库名

'database' => 'demo',

// 数据库用户名

'username' => 'root',

// 数据库密码

'password' => '',

// 数据库连接端口

'hostport' => '',

// 数据库连接参数

'params' => [],

// 数据库编码默认采用utf8

'charset' => 'utf8',

// 数据库表前缀

'prefix' => 'think\_',

// 数据库调试模式

'debug' => true,

];

接下来，我们修改下控制器方法，添加读取数据的代码：

<?php

namespace app\index\controller;

use think\Controller;

use think\Db;

class Index extends Controller

{

public function index()

{

$data = Db::name('data')->find();

$this->assign('result', $data);

return $this->fetch();

}

}

定义好控制器后，我们修改模板文件，添加数据输出标签如下：

<html>

<head>

<title></title>

</head>

<body>

{$result.id}--{$result.data}

</body>

</html>

模板标签的用法和Smarty类似，就是用于输出数据的字段，这里就表示输出think\_data表的id和data字段的值。

我们访问会输出：

1--thinkphp

（11）总结

总结

本章我们学习了如何安装ThinkPHP和框架的目录结构，如何创建项目的入口文件和开启调试模式，并通过一个Hello,Name例子说明了如何定义控制器和模板，以及如何读取数据库的数据并在模板渲染输出。

二、URL和路由

快速入门（二）：URL和路由

本章讲解URL访问和路由的使用。

（1）URL访问

URL访问

ThinkPHP采用单一入口模式访问应用，对应用的所有请求都定向到应用的入口文件，系统会从URL参数中解析当前请求的模块、控制器和操作，下面是一个标准的URL访问格式：

http://domainName/index.php/模块/控制器/操作

其中index.php就称之为应用的入口文件（注意入口文件可以被隐藏，后面会提到）。

提示：

模块在ThinkPHP中的概念其实就是应用目录下面的子目录，而官方的规范是目录名小写，因此模块全部采用小写命名，无论URL是否开启大小写转换，模块名都会强制小写。

应用的index模块的Index控制器定义如下：

<?php

namespace app\index\controller;

class Index

{

public function index()

{

return 'index';

}

public function hello($name = 'World')

{

return 'Hello,' . $name . '!';

}

}

如果我们直接访问入口文件的话，由于URL中没有模块、控制器和操作，因此系统会访问默认模块（index）下面的默认控制器（Index）的默认操作（index），因此下面的访问是等效的：

http://tp5.com/index.php

http://tp5.com/index.php/index/index/index

如果要访问控制器的hello方法，则需要使用完整的URL地址

http://tp5.com/index.php/index/index/hello/name/thinkphp

访问URL地址后页面输出结果为：

Hello,thinkphp!

由于name参数为可选参数，因此也可以使用

http://tp5.com/index.php/index/index/hello

访问URL地址后页面输出结果为：

Hello,World!

默认情况下，URL地址中的控制器和操作名是不区分大小写的，因此下面的访问其实是等效的：

http://tp5.com/index.php/index/Index/Index

http://tp5.com/index.php/index/INDEX/INDEX

如果你的控制器是驼峰的，例如定义一个HelloWorld控制器（application/index/controller/HelloWorld.php）：

<?php

namespace app\index\controller;

class HelloWorld

{

public function index($name = 'World')

{

return 'Hello,' . $name . '!';

}

}

正确的URL访问地址（该地址可以使用url方法生成）应该是：

http://tp5.com/index.php/index/hello\_world/index

系统会自动定位到HelloWorld控制器类去操作。

如果使用

http://tp5.com/index.php/index/HelloWorld/index

将会报错，并提示Helloworld控制器类不存在。

如果希望严格区分大小写访问（这样就可以支持驼峰法进行控制器访问），可以在应用配置文件中设置：

// 关闭URL自动转换（支持驼峰访问控制器）

'url\_convert' => false,

关闭URL自动转换之后，必须使用下面的URL地址访问（控制器名称必须严格使用控制器类的名称，不包含控制器后缀）：

http://tp5.com/index.php/index/Index/index

http://tp5.com/index.php/index/HelloWorld/index

提示：

操作方法的访问本身不会受URL自动转换的影响，但会影响默认的模板渲染输出。

如果你的服务器环境不支持pathinfo方式的URL访问，可以使用兼容方式，例如：

http://tp5.com/index.php?s=/index/Index/index

其中变量s的名称的可以配置的。

5.0不再支持普通的URL访问方式，所以下面的访问是无效的，你会发现无论输入什么，访问的都是默认的控制器和操作^\_^

http://tp5.com/index.php?m=index&c=Index&a=hello

（2）参数传入

参数传入

通过操作方法的参数绑定功能，可以实现自动获取URL的参数，仍然以上面的控制器为例，控制器代码如下：

<?php

namespace app\index\controller;

class Index

{

public function index()

{

return 'index';

}

public function hello($name = 'World')

{

return 'Hello,' . $name . '!';

}

}

当我们访问

http://tp5.com/index.php/index/index/hello

就是访问app\index\controller\Index控制器类的hello方法，因为没有传入任何参数，name参数就使用默认值World。如果传入name参数，则使用：

http://tp5.com/index.php/index/index/hello/name/thinkphp

页面输出结果为：

Hello,thinkphp!

现在给hello方法增加第二个参数：

public function hello($name = 'World', $city = '')

{

return 'Hello,' . $name . '! You come from ' . $city . '.';

}

访问地址为

http://tp5.com/index.php/index/index/hello/name/thinkphp/city/shanghai

页面输出结果为：

Hello,thinkphp! You come from shanghai.

可以看到，hello方法会自动获取URL地址中的同名参数值作为方法的参数值，而且这个参数的传入顺序不受URL参数顺序的影响，例如下面的URL地址输出的结果和上面是一样的：

http://tp5.com/index.php/index/index/hello/city/shanghai/name/thinkphp

或者使用

http://tp5.com/index.php/index/index/hello?city=shanghai&name=thinkphp

还可以进一步对URL地址做简化，前提就是我们必须明确参数的顺序代表的变量，我们更改下URL参数的获取方式，把应用配置文件中的url\_param\_type参数的值修改如下：

// 按照参数顺序获取

'url\_param\_type' => 1,

现在，URL的参数传值方式就变成了严格按照操作方法的变量定义顺序来传值了，也就是说我们必须使用下面的URL地址访问才能正确传入name和city参数到hello方法：

http://tp5.com/index.php/index/index/hello/thinkphp/shanghai

页面输出结果为：

Hello,thinkphp! You come from shanghai.

如果改变参数顺序为

http://tp5.com/index.php/index/index/hello/shanghai/thinkphp

页面输出结果为：

Hello,shanghai! You come from thinkphp.

显然不是我们预期的结果。

同样，我们试图通过

http://tp5.com/index.php/index/index/hello/name/thinkphp/city/shanghai

访问也不会得到正确的结果。

注意

按顺序绑定参数的话，操作方法的参数只能使用URL pathinfo变量，而不能使用get或者post变量。

# （3）隐藏入口

## 隐藏index.php

可以去掉URL地址里面的入口文件index.php，但是需要额外配置WEB服务器的重写规则。

以Apache为例，需要在入口文件的同级添加.htaccess文件（官方默认自带了该文件），内容如下：

<IfModule mod\_rewrite.c>

Options +FollowSymlinks -Multiviews

RewriteEngine on

RewriteCond %{REQUEST\_FILENAME} !-d

RewriteCond %{REQUEST\_FILENAME} !-f

RewriteRule ^(.\*)$ index.php/$1 [QSA,PT,L]

</IfModule>

如果用的phpstudy，规则如下：

<IfModule mod\_rewrite.c>

Options +FollowSymlinks -Multiviews

RewriteEngine on

RewriteCond %{REQUEST\_FILENAME} !-d

RewriteCond %{REQUEST\_FILENAME} !-f

RewriteRule ^(.\*)$ index.php [L,E=PATH\_INFO:$1]

</IfModule>

接下来就可以使用下面的URL地址访问了

http://tp5.com/index/index/index

http://tp5.com/index/index/hello

如果你使用的apache版本使用上面的方式无法正常隐藏index.php，可以尝试使用下面的方式配置.htaccess文件：

<IfModule mod\_rewrite.c>

Options +FollowSymlinks -Multiviews

RewriteEngine on

RewriteCond %{REQUEST\_FILENAME} !-d

RewriteCond %{REQUEST\_FILENAME} !-f

RewriteRule ^(.\*)$ index.php?/$1 [QSA,PT,L]

</IfModule>

如果是Nginx环境的话，可以在Nginx.conf中添加：

location / { // …..省略部分代码

if (!-e $request\_filename) {

rewrite ^(.\*)$ /index.php?s=/$1 last;

break;

}

}

更多的Nginx的rewrite用法可以参考[这篇文章](https://segmentfault.com/a/1190000008102599)。

#### 提示：

后面的示例访问地址，我们都假设配置了隐藏index.php。

（4）定义路由

定义路由

URL地址里面的index模块怎么才能省略呢，默认的URL地址显得有点长，下面就来说说如何通过路由简化URL访问。

我们在路由定义文件（application/route.php）里面添加一些路由规则，如下：

return [

// 添加路由规则 路由到 index控制器的hello操作方法

'hello/:name' => 'index/index/hello',

];

该路由规则表示所有hello开头的并且带参数的访问都会路由到index控制器的hello操作方法。

路由之前的URL访问地址为：

http://tp5.com/index/index/hello/name/thinkphp

定义路由后就只能访问下面的URL地址

http://tp5.com/hello/thinkphp

注意

定义路由规则后，原来的URL地址将会失效，变成非法请求。



但这里有一个小问题，如果我们只是访问

http://tp5.com/hello

将发生错误，



事实上这是由于路由没有正确匹配到，我们修改路由规则如下：

return [

// 路由参数name为可选

'hello/[:name]' => 'index/hello',

];

使用[]把路由规则中的变量包起来，就表示该变量为可选，接下来就可以正常访问了。

http://tp5.com/hello

当name参数没有传入值的时候，hello方法的name参数有默认值World，所以输出的内容为 Hello,World!

除了路由配置文件中定义之外，还可以采用动态定义路由规则的方式定义，例如在路由配置文件（application/route.php）的开头直接添加下面的方法：

use think\Route;

Route::rule('hello/:name', 'index/hello');

完成的效果和使用配置方式定义是一样的。

无论是配置方式还是通过Route类的方法定义路由，都统一放到路由配置文件application/route.php文件中，具体原因后面会揭晓。

提示：

注意路由配置不支持在模块配置文件中设置。

完整匹配

前面定义的路由是只要以hello开头就能进行匹配，如果需要完整匹配，可以使用下面的定义：

return [

// 路由参数name为可选

'hello/[:name]$' => 'index/hello',

];

当路由规则以$结尾的时候就表示当前路由规则需要完整匹配。

当我们访问下面的URL地址的时候：

http://tp5.com/hello // 正确匹配

http://tp5.com/hello/thinkphp // 正确匹配

http://tp5.com/hello/thinkphp/val/value // 不会匹配

闭包定义

还支持通过定义闭包为某些特殊的场景定义路由规则，例如：

return [

// 定义闭包

'hello/[:name]' => function ($name) {

return 'Hello,' . $name . '!';

},

];

或者

use think\Route;

Route::rule('hello/:name', function ($name) {

return 'Hello,' . $name . '!';

});

提示：

闭包函数的参数就是路由规则中定义的变量。

因此，当访问下面的URL地址：

http://tp5.com/hello/thinkphp

会输出

Hello,thinkphp!

设置URL分隔符

如果需要改变URL地址中的pathinfo参数分隔符，只需要在应用配置文件（application/config.php）中设置：

// 设置pathinfo分隔符

'pathinfo\_depr' => '-',

路由规则定义无需做任何改变，我们就可以访问下面的地址：

http://tp5.com/hello-thinkphp

路由参数

我们还可以约束路由规则的请求类型或者URL后缀之类的条件，例如：

return [

// 定义路由的请求类型和后缀

'hello/[:name]' => ['index/hello', ['method' => 'get', 'ext' => 'html']],

];

上面定义的路由规则限制了必须是get请求，而且后缀必须是html的，所以下面的访问地址：

http://tp5.com/hello // 无效

http://tp5.com/hello.html // 有效

http://tp5.com/hello/thinkphp // 无效

http://tp5.com/hello/thinkphp.html // 有效

更多的路由参数请参考完全开发手册的路由参数一节。

变量规则

接下来，我们来尝试一些复杂的路由规则定义满足不同的路由变量。在此之前，首先增加一个控制器类如下：

<?php

namespace app\index\controller;

class Blog

{

public function get($id)

{

return '查看id=' . $id . '的内容';

}

public function read($name)

{

return '查看name=' . $name . '的内容';

}

public function archive($year, $month)

{

return '查看' . $year . '/' . $month . '的归档内容';

}

}

添加如下路由规则：

return [

'blog/:year/:month' => ['blog/archive', ['method' => 'get'], ['year' => '\d{4}', 'month' => '\d{2}']],

'blog/:id' => ['blog/get', ['method' => 'get'], ['id' => '\d+']],

'blog/:name' => ['blog/read', ['method' => 'get'], ['name' => '\w+']],

];

在上面的路由规则中，我们对变量进行的规则约束，变量规则使用正则表达式进行定义。

我们看下几种URL访问的情况

// 访问id为5的内容

http://tp5.com/blog/5

// 访问name为thinkphp的内容

http://tp5.com/blog/thinkphp

// 访问2015年5月的归档内容

http://tp5.com/blog/2015/05

路由分组

上面的三个路由规则由于都是blog打头，所以我们可以做如下的简化：

return [

'[blog]' => [

':year/:month' => ['blog/archive', ['method' => 'get'], ['year' => '\d{4}', 'month' => '\d{2}']],

':id' => ['blog/get', ['method' => 'get'], ['id' => '\d+']],

':name' => ['blog/read', ['method' => 'get'], ['name' => '\w+']],

],

];

对于这种定义方式，我们称之为路由分组，路由分组一定程度上可以提高路由检测的效率。

复杂路由

有时候，我们还需要对URL做一些特殊的定制，例如如果要同时支持下面的访问地址

http://tp5.com/blog/thinkphp

http://tp5.com/blog-2015-05

我们只要稍微改变路由定义规则即可：

return [

'blog/:id' => ['blog/get', ['method' => 'get'], ['id' => '\d+']],

'blog/:name' => ['blog/read', ['method' => 'get'], ['name' => '\w+']],

'blog-<year>-<month>' => ['blog/archive', ['method' => 'get'], ['year' => '\d{4}', 'month' => '\d{2}']],

];

对 blog-<year>-<month> 这样的非正常规范，我们需要使用<变量名>这样的变量定义方式，而不是 :变量名方式。

简单起见，我们还可以把变量规则统一定义，例如：

return [

// 全局变量规则定义

'\_\_pattern\_\_' => [

'name' => '\w+',

'id' => '\d+',

'year' => '\d{4}',

'month' => '\d{2}',

],

// 路由规则定义

'blog/:id' => 'blog/get',

'blog/:name' => 'blog/read',

'blog-<year>-<month>' => 'blog/archive',

];

在\_\_pattern\_\_中定义的变量规则我们称之为全局变量规则，在路由规则里面定义的变量规则我们称之为局部变量规则，如果一个变量同时定义了全局规则和局部规则的话，当前的局部规则会覆盖全局规则的，例如：

return [

// 全局变量规则

'\_\_pattern\_\_' => [

'name' => '\w+',

'id' => '\d+',

'year' => '\d{4}',

'month' => '\d{2}',

],

'blog/:id' => 'blog/get',

// 定义了局部变量规则

'blog/:name' => ['blog/read', ['method' => 'get'], ['name' => '\w{5,}']],

'blog-<year>-<month>' => 'blog/archive',

];

【 5.1 】使用须知

5.1的路由配置文件改为route/route.php，并且支持随意命名，都会自动加载。并尽量使用方法注册路由的方式替代数组配置的方式，例如。

use think\facade\Route;

Route::get('blog/:id','blog/get');

Route::get('blog/:name','blog/read');

（5）URL生成

URL生成

定义路由规则之后，我们可以通过Url类来方便的生成实际的URL地址（路由地址），针对上面的路由规则，我们可以用下面的方式生成URL地址。

// 输出 blog/thinkphp

Url::build('blog/read', 'name=thinkphp');

Url::build('blog/read', ['name' => 'thinkphp']);

// 输出 blog/5

Url::build('blog/get', 'id=5');

Url::build('blog/get', ['id' => 5]);

// 输出 blog/2015/05

Url::build('blog/archive', 'year=2015&month=05');

Url::build('blog/archive', ['year' => '2015', 'month' => '05']);

提示：

build方法的第一个参数使用路由定义中的完整路由地址。

我们还可以使用系统提供的助手函数url来简化

url('blog/read', 'name=thinkphp');

// 等效于

Url::build('blog/read', 'name=thinkphp');

通常在模板文件中输出的话，可以使用助手函数，例如：

{:url('blog/read', 'name=thinkphp')}

如果我们的路由规则发生调整，生成的URL地址会自动变化。

如果你配置了url\_html\_suffix参数的话，生成的URL地址会带上后缀，例如：

'url\_html\_suffix' => 'html',

那么生成的URL地址 类似

blog/thinkphp.html

blog/2015/05.html

如果你的URL地址全部采用路由方式定义，也可以直接使用路由规则来定义URL生成，例如：

url('/blog/thinkphp');

Url::build('/blog/8');

Url::build('/blog/archive/2015/05');

生成方法的第一个参数一定要和路由定义的路由地址保持一致，如果你的路由地址比较特殊，例如使用闭包定义的话，则需要手动给路由指定标识，例如：

// 添加hello路由标识

Route::rule(['hello','hello/:name'], function($name){

return 'Hello,'.$name;

});

// 根据路由标识快速生成URL

Url::build('hello', 'name=thinkphp');

// 或者使用

Url::build('hello', ['name' => 'thinkphp']);

目前为止，我们掌握的路由功能还只是ThinkPHP5.0路由功能的冰山一角，以后我们还会通过更多的专题来讲解路由。

【 5.1 】使用须知

5.1版本，你需要引入think\facade\Url才能使用静态方法调用，其它用法不变。

三、请求和响应

快速入门（三）：请求和响应

本章主要了解如何获取当前的请求信息，以及进行不同的输出响应、跳转和页面重定向。

ThinkPHP5的架构设计和之前版本的主要区别之一就在于增加了Request请求对象和Response响应对象的概念，了解了这两个对象的作用和用法对你的应用开发非常关键。

（1）请求对象

请求对象

Request对象的作用是与客户端交互，收集客户端的Form、Cookies、超链接，或者收集服务器端的环境变量。

Request对象是从客户端向服务器发出请求，包括用户提交的信息以及客户端的一些信息。客户端可通过HTML表单或在网页地址后面提供参数的方法提交数据，然后通过Request对象的相关方法来获取这些数据。Request的各种方法主要用来处理客户端浏览器提交的请求中的各项参数和选项。

ThinkPHP5的Request对象由think\Request类完成。

Request对象的一个主要职责是统一和更安全地获取当前的请求信息，你需要避免直接操作$\_GET、$\_POST、$\_REQUEST、$\_SESSION、$\_COOKIE，甚至$\_FILES等全局变量，而是统一使用Request对象提供的方法来获取请求变量。

下面来举一个最简单的例子（获取当前访问的url地址）来说明在不同的情况下如何调用Request对象。

传统方式调用

该用法主要是用来告诉大家Request对象是如何实例化的，但实际开发中很少选择这种方式调用。

<?php

namespace app\index\controller;

use think\Request;

class Index

{

public function hello($name = 'World')

{

$request = Request::instance();

// 获取当前URL地址 不含域名

echo 'url: ' . $request->url() . '<br/>';

return 'Hello,' . $name . '！';

}

}

访问下面的URL地址：

http://tp5.com/index/index/hello.html?name=thinkphp

页面输出结果为：

url: /index/index/hello.html?name=thinkphp

Hello,thinkphp！

继承think\Controller

如果控制器类继承了think\Controller的话，可以做如下简化调用：

<?php

namespace app\index\controller;

use think\Controller;

class Index extends Controller

{

public function hello($name = 'World')

{

// 获取当前URL地址 不含域名

echo 'url: ' . $this->request->url() . '<br/>';

return 'Hello,' . $name . '！';

}

}

自动注入请求对象

如果没有继承think\Controller，则可以使用Request对象注入的方式来简化调用，任何情况下都适用，也是系统建议的方式：

<?php

namespace app\index\controller;

use think\Request;

class Index

{

public function hello(Request $request, $name = 'World')

{

// 获取当前URL地址 不含域名

echo 'url: ' . $request->url() . '<br/>';

return 'Hello,' . $name . '！';

}

}

提示：

hello方法的request参数是系统自动注入的，而不需要通过URL请求传入。

动态绑定属性

可以给Request请求对象绑定属性，方便全局调用，例如我们可以在公共控制器中绑定当前登录的用户模型到请求对象：

<?php

namespace app\index\controller;

use app\index\model\User;

use think\Controller;

use think\Request;

use think\Session;

class Base extends Controller

{

public function \_initialize()

{

$user = User::get(Session::get('user\_id'));

Request::instance()->bind('user',$user);

}

}

注意

User对象的用法我们会在模型和关联一章提及，如果不明白这里的可以暂时忽略。

然后，直接继承这个控制器类后就可以直接使用

<?php

namespace app\index\controller;

use app\index\controller\Base;

use think\Request;

class Index extends Base

{

public function index(Request $request)

{

echo $request->user->id;

echo $request->user->name;

}

}

使用助手函数

如果既没有继承think\Controller也不想给操作方法添加额外的Request对象参数，那么也可以使用系统提供的助手，例如：

<?php

namespace app\index\controller;

class Index

{

public function hello($name = 'World')

{

// 获取当前URL地址 不含域名

echo 'url: ' . request()->url() . '<br/>';

return 'Hello,' . $name . '！';

}

}

上面任意一种方式都可以调用当前请求的Request对象实例，然后通过请求对象实例的方法来完成不同的信息获取或者设置。

在后面的示例中，如果没有特别说明，我们均以第三种方式，自动注入请求对象的方式来讲解，请注意根据自己的调用方式灵活运用。

【 5.1 】使用须知

5.1的请求对象使用注意事项如下：

不需要使用Request::instance()然后再调用方法，直接使用Facade特性即可，例如：

<?php

namespace app\index\controller;

use think\facade\Request;

class Index

{

public function hello($name = 'World')

{

// 获取当前URL地址 不含域名

echo 'url: ' . Request::url() . '<br/>';

return 'Hello,' . $name . '！';

}

}

（2）请求信息

请求信息

我们来陆续了解下请求对象的常用方法。

获取请求变量

系统推荐使用param方法统一获取当前请求变量，该方法最大的优势是让你不需要区分当前请求类型而使用不同的全局变量或者方法，并且可以满足大部分的参数需求，下面举一个例子。

<?php

namespace app\index\controller;

use think\Request;

class Index

{

public function hello(Request $request)

{

echo '请求参数：';

dump($request->param());

echo 'name:'.$request->param('name');

}

}

访问下面的URL地址：

http://tp5.com/index/index/hello.html?test=ddd&name=thinkphp

页面输出结果为：

请求参数：

array (size=2)

'test' => string 'ddd' (length=3)

'name' => string 'thinkphp' (length=8)

name:thinkphp

系统提供了一个input助手函数来简化Request对象的param方法，用法如下：

<?php

namespace app\index\controller;

class Index

{

public function hello()

{

echo '请求参数：';

dump(input());

echo 'name:'.input('name');

}

}

param方法获取的参数会自动判断当前的请求，以POST请求为例的话，参数的优先级别为：

路由变量 > 当前请求变量（$\_POST变量） > $\_GET变量

注意：

这里的路由变量指的是路由规则里面定义的变量或者PATH\_INFO地址中的变量。路由变量无法使用get方法或者$\_GET变量获取。

param方法支持变量的过滤和默认值，例如：

<?php

namespace app\index\controller;

use think\Request;

class Index

{

public function hello(Request $request)

{

echo 'name:'.$request->param('name','World','strtolower');

echo '<br/>test:'.$request->param('test','thinkphp','strtoupper');

}

}

访问下面的URL地址：

http://tp5.com/index/index/hello.html?test=ddd&name=thinkphp

页面输出结果为：

name:thinkphp

test:DDD

可以设置全局的过滤方法，如下：

// 设置默认的全局过滤规则 多个用数组或者逗号分隔

'default\_filter' => 'htmlspecialchars ',

除了Param方法之外，Request对象也可以用于获取其它的输入参数，例如：

<?php

namespace app\index\controller;

use think\Request;

class Index

{

public function hello(Request $request)

{

echo 'GET参数：';

dump($request->get());

echo 'GET参数：name';

dump($request->get('name'));

echo 'POST参数：name';

dump($request->post('name'));

echo 'cookie参数：name';

dump($request->cookie('name'));

echo '上传文件信息：image';

dump($request->file('image'));

}

}

页面输出的结果为：

GET参数：

array (size=1)

'name' => string 'thinkphp' (length=8)

GET参数：name

string 'thinkphp' (length=8)

POST参数：name

null

当前请求（自动识别请求类型）的参数：name

string 'thinkphp' (length=8)

cookie参数：name

null

上传文件信息：image

null

使用助手函数的代码为：

<?php

namespace app\index\controller;

class Index

{

public function hello()

{

echo 'GET参数：';

dump(input('get.'));

echo 'GET参数：name';

dump(input('get.name'));

echo 'POST参数：name';

dump(input('post.name'));

echo 'cookie参数：name';

dump(input('cookie.name'));

echo '上传文件信息：image';

dump(input('file.image'));

}

}

获取变量的方法包括：

| **方法** | **作用** |
| --- | --- |
| param | 获取请求变量 |
| get | 获取$\_GET变量 |
| post | 获取$\_POST变量 |
| put | 获取PUT请求变量 |
| delete | 获取DELETE请求变量 |
| patch | 获取PATCH请求变量 |
| request | 获取$\_REQUEST变量 |
| route | 获取路由（URL）变量 |
| session | 获取$\_SESSION变量 |
| cookie | 获取$\_COOKIE变量 |
| server | 获取$\_SERVER变量 |
| env | 获取$\_ENV变量 |
| file | 获取上传文件信息 |

除了file方法之外，其它方法都支持默认值和过滤方法。

获取请求参数

把Hello方法改为如下：

<?php

namespace app\index\controller;

use think\Request;

class Index

{

public function hello(Request $request)

{

echo '请求方法：' . $request->method() . '<br/>';

echo '资源类型：' . $request->type() . '<br/>';

echo '访问IP：' . $request->ip() . '<br/>';

echo '是否AJax请求：' . var\_export($request->isAjax(), true) . '<br/>';

echo '请求参数：';

dump($request->param());

echo '请求参数：仅包含name';

dump($request->only(['name']));

echo '请求参数：排除name';

dump($request->except(['name']));

}

}

访问下面的URL地址：

http://tp5.com/index/index/hello/test/ddd.html?name=thinkphp

页面输出结果为：

请求方法：GET

资源类型：html

访问地址：127.0.0.1

是否AJax请求：false

请求参数：

array (size=2)

'test' => string 'ddd' (length=3)

'name' => string 'thinkphp' (length=8)

请求参数：仅包含name

array (size=1)

'name' => string 'thinkphp' (length=8)

请求参数：排除name

array (size=1)

'test' => string 'ddd' (length=3)

获取URL信息

<?php

namespace app\index\controller;

use think\Request;

class Index

{

public function hello(Request $request,$name = 'World')

{

// 获取当前域名

echo 'domain: ' . $request->domain() . '<br/>';

// 获取当前入口文件

echo 'file: ' . $request->baseFile() . '<br/>';

// 获取当前URL地址 不含域名

echo 'url: ' . $request->url() . '<br/>';

// 获取包含域名的完整URL地址

echo 'url with domain: ' . $request->url(true) . '<br/>';

// 获取当前URL地址 不含QUERY\_STRING

echo 'url without query: ' . $request->baseUrl() . '<br/>';

// 获取URL访问的ROOT地址

echo 'root:' . $request->root() . '<br/>';

// 获取URL访问的ROOT地址

echo 'root with domain: ' . $request->root(true) . '<br/>';

// 获取URL地址中的PATH\_INFO信息

echo 'pathinfo: ' . $request->pathinfo() . '<br/>';

// 获取URL地址中的PATH\_INFO信息 不含后缀

echo 'pathinfo: ' . $request->path() . '<br/>';

// 获取URL地址中的后缀信息

echo 'ext: ' . $request->ext() . '<br/>';

return 'Hello,' . $name . '！';

}

}

访问下面的URL地址：

http://tp5.com/index/index/hello.html?name=thinkphp

页面输出结果为：

domain: http://tp5.com

file: /index.php

url: /index/index/hello.html?name=thinkphp

url with domain: http://tp5.com/index/index/hello.html?name=thinkphp

url without query: /index/index/hello.html

root:

root with domain: http://tp5.com

pathinfo: index/index/hello.html

pathinfo: index/index/hello

ext: html

Hello,thinkphp！

URL请求和信息方法可以总结如下：

| **方法** | **作用** |
| --- | --- |
| domain | 获取当前的域名 |
| url | 获取当前的完整URL地址 |
| baseUrl | 获取当前的URL地址，不含QUERY\_STRING |
| baseFile | 获取当前的SCRIPT\_NAME |
| root | 获取当前URL的root地址 |
| pathinfo | 获取当前URL的pathinfo地址 |
| path | 获取当前URL的pathinfo地址，不含后缀 |
| ext | 获取当前URL的后缀 |
| type | 获取当前请求的资源类型 |
| scheme | 获取当前请求的scheme |
| query | 获取当前URL地址的QUERY\_STRING |
| host | 获取当前URL的host地址 |
| port | 获取当前URL的port号 |
| protocol | 获取当前请求的SERVER\_PROTOCOL |
| remotePort | 获取当前请求的REMOTE\_PORT |

url、baseUrl、baseFile、root方法如果传入true，表示获取包含域名的地址。

获取当前模块/控制器/操作信息

hello方法修改如下：

public function hello(Request $request, $name = 'World')

{

echo '模块：'.$request->module();

echo '<br/>控制器：'.$request->controller();

echo '<br/>操作：'.$request->action();

}

访问URL地址：

http://tp5.com/index/index/hello.html?test=ddd&name=thinkphp

页面会显示：

模块：index

控制器：Index

操作：hello

controller方法获取的是驼峰命名的实际的控制器名，其它都是小写返回。

获取路由和调度信息

hello方法修改如下：

<?php

namespace app\index\controller;

use think\Request;

class Index

{

public function hello(Request $request, $name = 'World')

{

echo '路由信息：';

dump($request->routeInfo());

echo '调度信息：';

dump($request->dispatch());

return 'Hello,' . $name . '！';

}

}

然后在配置文件中添加路由定义为：

return [

'hello/:name' =>['index/hello',[],['name'=>'\w+']],

];

访问下面的URL地址：

http://tp5.com/hello/thinkphp

页面输出信息为：

路由信息：

array (size=4)

'rule' => string 'hello/:name' (length=11)

'route' => string 'index/hello' (length=11)

'pattern' =>

array (size=1)

'name' => string '\w+' (length=3)

'option' =>

array (size=0)

empty

调度信息：

array (size=2)

'type' => string 'module' (length=6)

'module' =>

array (size=3)

0 => null

1 => string 'index' (length=5)

2 => string 'hello' (length=5)

Hello,thinkphp！

（3）响应对象

响应对象

Response对象用于动态响应客户端请示，控制发送给用户的信息，并将动态生成响应。通常用于输出数据给客户端或者浏览器。

ThinkPHP5的Response响应对象由think\Response类或者子类完成，下面介绍下基本的用法。

自动输出

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，系统会根据default\_return\_type和default\_ajax\_return配置决定响应输出的类型。

默认的自动响应输出会自动判断是否AJAX请求，如果是的话会自动输出default\_ajax\_return配置的输出类型。

<?php

namespace app\index\controller;

class Index

{

public function hello()

{

$data = ['name' => 'thinkphp', 'status' => '1'];

return $data;

}

}

由于默认是输出Html输出，所以访问页面输出结果为：

https://box.kancloud.cn/2016-05-22_574167bc1dd12.png

修改配置文件，添加：

// 默认输出类型

'default\_return\_type' => 'json',

再次访问的输出结果为：

{"name":"thinkphp","status":"1"}

修改输出类型为xml：

// 默认输出类型

'default\_return\_type' => 'xml',

则输出结果变成：

<think>

<name>thinkphp</name>

<status>1</status>

</think>

手动输出

在必要的时候，可以手动控制输出类型和参数，这种方式较为灵活。如果需要指定输出类型，可以通过下面的方式：

<?php

namespace app\index\controller;

class Index

{

public function hello()

{

$data = ['name' => 'thinkphp', 'status' => '1'];

return json($data);

}

}

无论配置参数如何设置，页面输出的结果为：

{"name":"thinkphp","status":"1"}

默认的情况下发送的http状态码是200，如果需要返回其它的状态码，可以使用：

<?php

namespace app\index\controller;

class Index

{

public function hello()

{

$data = ['name' => 'thinkphp', 'status' => '1'];

return json($data, 201);

}

}

或者发送更多的响应头信息：

<?php

namespace app\index\controller;

class Index

{

public function hello()

{

$data = ['name' => 'thinkphp', 'status' => '1'];

return json($data, 201, ['Cache-control' => 'no-cache,must-revalidate']);

}

}

也支持使用下面的方式：

<?php

namespace app\index\controller;

class Index

{

public function hello()

{

$data = ['name' => 'thinkphp', 'status' => '1'];

return json($data)->code(201)->header(['Cache-control' => 'no-cache,must-revalidate']);

}

}

默认支持的输出类型包括：

| **输出类型** | **快捷方法** |
| --- | --- |
| 渲染模板输出 | view |
| JSON输出 | json |
| JSONP输出 | jsonp |
| XML输出 | xml |
| 页面重定向 | redirect |

所以，同样的可以使用xml方法输出XML数据类型：

<?php

namespace app\index\controller;

class Index

{

public function hello()

{

$data = ['name' => 'thinkphp', 'status' => '1'];

return xml($data, 201);

}

}

页面跳转

如果需要进行一些简单的页面操作提示或者重定向，可以引入traits\controller\Jump，就可以使用相关页面跳转和重定向方法，下面举一个简单的例子，当页面传入name参数为thinkphp的时候，跳转到欢迎页面，其它情况则跳转到一个guest页面。

namespace app\index\controller;

class Index

{

use \traits\controller\Jump;

public function index($name='')

{

if ('thinkphp' == $name) {

$this->success('欢迎使用ThinkPHP

5.0','hello');

} else {

$this->error('错误的name','guest');

}

}

public function hello()

{

return 'Hello,ThinkPHP!';

}

public function guest()

{

return 'Hello,Guest!';

}

}

这里我们使用

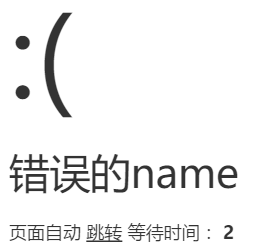
use \traits\controller\Jump;

引入了一个Jump trait，这是PHP5.4版本的新特性，如果你的控制器类是继承的\think\Controller的话，系统已经自动为你引入了 \traits\controller\Jump，无需再次引入。

当我们访问

http://tp5.com

页面显示如图所示：



然后显示 Hello,Guest!。

如果访问URL地址为：

http://tp5.com?name=thinkphp

则页面显示如图所示：



然后显示 Hello,ThinkPHP!。

页面重定向

如果要进行页面重定向跳转，可以使用：

namespace app\index\controller;

class Index

{

use \traits\controller\Jump;

public function index($name='')

{

if ('thinkphp' == $name) {

$this->redirect('http://thinkphp.cn');

} else {

$this->success('欢迎使用ThinkPHP','hello');

}

}

public function hello()

{

return 'Hello,ThinkPHP!';

}

}

当我们再次访问

http://tp5.com?name=thinkphp

就会重定向到ThinkPHP官网 http://thinkphp.cn

redirect方法默认使用302跳转，如不需要可以使用第二个参数进行301跳转。

$this->redirect('http://thinkphp.cn',301);

在任何时候（即使没有引入Jump trait的话），我们可以使用系统提供的助手函数redirect函数进行重定向。

namespace app\index\controller;

class Index

{

public function index($name='')

{

if ('thinkphp' == $name) {

return redirect('http://thinkphp.cn');

} else {

return 'Hello,ThinkPHP!';

}

}

}

注意，使用redirect助手函数重定向的时候必须加上return返回才会生效。

（4）总结

总结

本章我们了解了Request请求对象和Response响应对象的基本用法，以及如何更轻松的获取请求变量。

四、数据库

快速入门（四）：数据库

本章我们来学习下如何使用Db类操作数据库，主要包含：

（1）准备

准备

5.0的数据查询由低到高分三个层次：

1. 数据库原生查询（SQL查询）；
2. 数据库链式查询（查询构造器）；
3. 模型的对象化查询；

本章会涉及到前面两个，模型的查询会在第六章进行讲解。

在第一章已经提到，在使用Db类进行数据库查询之前，首先必须先创建一个控制器类，以及一个操作方法用于测试，类似于：

<?php

namespace app\index\controller;

use think\Db;

class Index

{

public function index()

{

// 后面的数据库查询代码都放在这个位置

}

}

然后，要查看数据库执行结果的话，访问下面的URL地址：

http://tp5.com/

在开始之前，我们首先在数据库demo中创建一个think\_data数据表（这里以mysql数据库为例），SQL代码如下：

CREATE TABLE IF NOT EXISTS `think\_data`(

`id` int(8) unsigned NOT NULL AUTO\_INCREMENT,

`name` varchar(255) NOT NULL COMMENT '名称',

`status` tinyint(2) NOT NULL DEFAULT '0' COMMENT '状态',

PRIMARY KEY (`id`)

) ENGINE=MyISAM DEFAULT CHARSET=utf8 ;

INSERT INTO `think\_data`(`id`,`name`,`status`) VALUES

(1,'thinkphp',1),

(2,'onethink',1),

(3,'topthink',1);

（2）数据库配置

数据库配置

我们给应用定义数据库配置文件（appliation/database.php），里面设置了应用的全局数据库配置信息。

该数据库配置文件的基本定义如下：

return [

// 数据库类型

'type' => 'mysql',

// 服务器地址

'hostname' => '127.0.0.1',

// 数据库名

'database' => 'test',

// 数据库用户名

'username' => 'root',

// 数据库密码

'password' => '',

// 数据库连接端口

'hostport' => '',

// 数据库连接参数

'params' => [],

// 数据库编码默认采用utf8

'charset' => 'utf8',

// 数据库表前缀

'prefix' => '',

// 数据库调试模式

'debug' => true,

];

如果你使用了多个模块，并且不同的模块采用不同的数据库连接，那么可以在每个模块的目录下面单独定义数据库配置。

后面的例子，我们都采用index模块的数据库配置文件（application/index/database.php），配置如下（模块数据库配置中我们使用了长连接）：

return [

// 数据库名

'database' => 'demo',

// 数据库表前缀

'prefix' => 'think\_',

// 数据库连接参数

'params' => [

// 使用长连接

\PDO::ATTR\_PERSISTENT => true,

],

];

提示：

模块的数据库配置文件中只需要配置和全局数据库配置文件差异的部分，相同的不需要重复配置。

也可以在调用Db类的时候，使用connect方法动态连接或者切换不同的数据库，这个我们会在后面提到。

【 5.1 】使用须知

模块的数据库配置文件应该放到application/index/config/database.php

（3）原生查询

原生查询

设置好数据库连接信息后，我们就可以直接进行原生的SQL查询操作了，包括query和execute两个方法，分别用于查询操作和写操作，下面我们来实现数据表think\_data的CURD操作。

创建（create）

// 插入记录

$result = Db::execute('insert into think\_data (id, name ,status) values (5, "thinkphp",1)');

dump($result);

更新（update）

// 更新记录

$result = Db::execute('update think\_data set name = "framework" where id = 5 ');

dump($result);

读取（read）

// 查询数据

$result = Db::query('select \* from think\_data where id = 5');

dump($result);

query方法返回的结果是一个数据集（数组），如果没有查询到数据则返回空数组。

删除（delete）

// 删除数据

$result = Db::execute('delete from think\_data where id = 5 ');

dump($result);

其它操作

可以执行一些其他的数据库操作，原则上，读操作都使用query方法，写操作使用execute方法即可，例如：

// 显示数据库列表

$result = Db::query('show tables from demo');

dump($result);

// 清空数据表

$result = Db::execute('TRUNCATE table think\_data');

dump($result);

query方法用于查询，默认情况下返回的是数据集（二维数组），execute方法的返回值是影响的记录数。

切换数据库

在进行数据库查询的时候，支持切换数据库进行查询，例如：

$result = Db::connect([

// 数据库类型

'type' => 'mysql',

// 服务器地址

'hostname' => '127.0.0.1',

// 数据库名

'database' => 'thinkphp',

// 数据库用户名

'username' => 'root',

// 数据库密码

'password' => '123456',

// 数据库连接端口

'hostport' => '',

// 数据库连接参数

'params' => [],

// 数据库编码默认采用utf8

'charset' => 'utf8',

// 数据库表前缀

'prefix' => 'think\_',

])->query('select \* from think\_data');

dump($result);

或者采用字符串方式定义（字符串方式无法定义数据表前缀和连接参数），如下：

$result = Db::connect('mysql://root:123456@127.0.0.1:3306/thinkphp#utf8')->query('select \* from think\_data where id = 1');

dump($result);

为了简化代码，通常的做法是事先在配置文件中定义好多个数据库的连接配置，例如，我们在应用配置文件（application/config.php）中添加配置如下：

// 数据库配置1

'db1' => [

// 数据库类型

'type' => 'mysql',

// 服务器地址

'hostname' => '127.0.0.1',

// 数据库名

'database' => 'thinkphp',

// 数据库用户名

'username' => 'root',

// 数据库密码

'password' => '123456',

// 数据库连接端口

'hostport' => '',

// 数据库连接参数

'params' => [],

// 数据库编码默认采用utf8

'charset' => 'utf8',

// 数据库表前缀

'prefix' => 'think\_',

],

// 数据库配置2

'db2' => [

// 数据库类型

'type' => 'mysql',

// 服务器地址

'hostname' => '127.0.0.1',

// 数据库名

'database' => 'test',

// 数据库用户名

'username' => 'root',

// 数据库密码

'password' => '',

// 数据库连接端口

'hostport' => '',

// 数据库连接参数

'params' => [],

// 数据库编码默认采用utf8

'charset' => 'utf8',

// 数据库表前缀

'prefix' => 'test\_',

],

然后就可以直接在connect方法中传入配置参数进行切换数据库连接，例如：

$result = Db::connect('db1')->query('select \* from think\_data where id = 1');

$result = Db::connect('db2')->query('select \* from think\_data where id = 1');

connect方法中的配置参数需要完整定义，并且仅仅对当此查询有效，下次调用Db类的时候还是使用默认的数据库连接。如果需要多次切换数据库查询，可以使用：

$db1 = Db::connect('db1');

$db2 = Db::connect('db2');

$db1->query('select \* from think\_data where id = 1');

$db2->query('select \* from think\_data where id = 1');

connect方法必须直接进行查询，下面的写法是无效的

Db::connect('db1');

Db::query('select \* from think\_data where id = 1');

参数绑定

实际开发中，可能某些数据使用的是外部传入的变量，为了让查询操作更加安全，我们建议使用参数绑定机制，例如上面的操作可以改为：

Db::execute('insert into think\_data (id, name ,status) values (?, ?, ?)', [8, 'thinkphp', 1]);

$result = Db::query('select \* from think\_data where id = ?', [8]);

dump($result);

也支持命名占位符绑定，例如：

Db::execute('insert into think\_data (id, name , status) values (:id, :name, :status)', ['id' => 10, 'name' => 'thinkphp', 'status' => 1]);

$result = Db::query('select \* from think\_data where id=:id', ['id' => 10]);

dump($result);

（4）查询构造器

查询构造器

除了原生查询外，5.0还提供了数据库查询构造器，可以更方便执行数据库操作，查询构造器基于PDO实现，对不同的数据库驱动都是统一的语法。

注意：

ThinkPHP 5.0查询构造器使用 PDO参数绑定，以保护应用程序免于 SQL注入，因此传入的参数不需额外转义特殊字符。

同样是实现上面的功能，我们可以改成：

// 插入记录

Db::table('think\_data')

->insert(['id' => 18, 'name' => 'thinkphp', 'status' => 1]);

// 更新记录

Db::table('think\_data')

->where('id', 18)

->update(['name' => "hello"]);

// 查询数据

$list = Db::table('think\_data')

->field('name,email')

->where('id', 18)

->select();

// 删除数据

Db::table('think\_data')

->where('id', 18)

->delete();

由于我们在数据库配置文件中设置了数据表的前缀为think\_，因此，table方法可以改成name方法，这样就不会因为数据表前缀的修改而改动CURD代码，例如：

// 插入记录

Db::name('data')

->insert(['id' => 18, 'name' => 'thinkphp']);

// 更新记录

Db::name('data')

->where('id', 18)

->update(['name' => "framework"]);

// 查询数据

$list = Db::name('data')

->where('id', 18)

->select();

dump($list);

// 删除数据

Db::name('data')

->where('id', 18)

->delete();

如果使用系统提供的助手函数db则可以进一步简化查询代码如下：

$db = db('data');

// 插入记录

$db->insert(['id' => 20, 'name' => 'thinkphp']);

// 更新记录

$db->where('id', 20)->update(['name' => "framework"]);

// 查询数据

$list = $db->where('id', 20)->select();

dump($list);

// 删除数据

$db->where('id', 20)->delete();

db助手函数在V5.0.9之前版本默认会每次重新连接数据库，因此应当尽量避免多次调用。

关于更多的查询条件和查询语法，会在后面一章查询语言中详细讲述。

（5）链式操作

链式操作

使用链式操作可以完成复杂的数据库查询操作，例如：

// 查询十个满足条件的数据 并按照id倒序排列

$list = Db::name('data')

->where('status', 1)

->field('id,name')

->order('id', 'desc')

->limit(10)

->select();

dump($list);

链式操作不分先后，只要在查询方法（这里是select方法）之前调用就行，所以，下面的查询是等效的：

// 查询十个满足条件的数据 并按照id倒序排列

$list = Db::name('data')

->field('id,name')

->order('id', 'desc')

->where('status', 1)

->limit(10)

->select();

dump($list);

支持链式操作的查询方法包括：

| **方法名** | **描述** |
| --- | --- |
| select | 查询数据集 |
| find | 查询单个记录 |
| insert | 插入记录 |
| update | 更新记录 |
| delete | 删除记录 |
| value | 查询值 |
| column | 查询列 |
| chunk | 分块查询 |
| count等 | 聚合查询 |

更多的链式操作方法可以参考官方的完全开发手册。

（6）事务支持

事务支持

注意：

由于需要用到事务的功能，请先修改数据表的类型为InnoDB，而不是MyISAM。

对于事务的支持，最简单的方法就是使用transaction方法，只需要把需要执行的事务操作封装到闭包里面即可自动完成事务，例如：

Db::transaction(function () {

Db::table('think\_user')

->delete(1);

Db::table('think\_data')

->insert(['id' => 28, 'name' => 'thinkphp', 'status' => 1]);

});

一旦think\_data表写入失败的话，系统会自动回滚，写入成功的话系统会自动提交当前事务。

也可以手动控制事务的提交，上面的实现代码可以改成：

// 启动事务

Db::startTrans();

try {

Db::table('think\_user')

->delete(1);

Db::table('think\_data')

->insert(['id' => 28, 'name' => 'thinkphp', 'status' => 1]);

// 提交事务

Db::commit();

} catch (\Exception $e) {

// 回滚事务

Db::rollback();

}

注意：

事务操作只对支持事务的数据库，并且设置了数据表为事务类型才有效，在Mysql数据库中请设置表类型为InnoDB。并且事务操作必须使用同一个数据库连接。

五、查询语言

快速入门（五）：查询语言

（1）查询表达式

查询表达式

最普通的查询就是判断某个字段是否等于某个值，例如，我们查询think\_data数据表中id等于1的数据，用法如下：

$result = Db::name('data')

->where('id', 1)

->find();

dump($result);

新手注意，如果没有使用use引入Db类的话 需要使用：

$result = \think\Db::name('data')

->where('id', 1)

->find();

dump($result);

提示：

find方法用于查找满足条件第一个记录（即使你的查询条件有多个符合的数据），如果查询成功，返回的是一个一维数组，没有满足条件的话则默认返回null（也支持设置是否抛出异常）。

生成的SQL语句是：

SELECT \* FROM `think\_data` WHERE `id` = 1

上述的查询其实等同于：

$result = Db::name('data')

->where('id', '=', 1)

->find();

dump($result);

使用表达式查询的时候，where方法的参数依次为：

where( 字段名，条件表达式，查询值 )

可以支持的查询表达式包括如下：

| **表达式** | **含义** |
| --- | --- |
| EQ、= | 等于（=） |
| NEQ、<> | 不等于（<>） |
| GT、> | 大于（>） |
| EGT、>= | 大于等于（>=） |
| LT、< | 小于（<） |
| ELT、<= | 小于等于（<=） |
| LIKE | 模糊查询 |
| [NOT] BETWEEN | （不在）区间查询 |
| [NOT] IN | （不在）IN 查询 |
| [NOT] NULL | 查询字段是否（不）是NULL |
| [NOT] EXISTS | EXISTS查询 |
| EXP | 表达式查询，支持SQL语法 |

其中条件表达式不区分大小写

下面就来查询id大于等于1的数据，使用如下代码：

$result = Db::name('data')

->where('id', '>=', 1)

->limit(10)

->select();

dump($result);

因为这里需要返回多条记录，因此这里我们使用了select方法，并且使用limit方法限制了返回的最多记录数。

提示：

select方法用于查询数据集，如果查询成功，返回的是一个二维数组，如果没有满足条件的话则返回空数组（也支持设置是否需要抛出异常）。

生成的SQL语句是：

SELECT \* FROM `think\_data` WHERE `id` >= 1 LIMIT 10

如果使用EXP条件表达式的话，表示后面是原生的SQL语句表达式，例如上面的查询可以改成：

$result = Db::name('data')

->where('id', 'exp', '>= 1')

->limit(10)

->select();

dump($result);

生成的SQL语句和前面是一样的。

如果要查询id的范围，可以使用

$result = Db::name('data')

// id 是 1、2、3 其中的数字

->where('id', 'in', [1, 2, 3])

->select();

dump($result);

生成的SQL语句是：

SELECT \* FROM `think\_data` WHERE `id` IN (1,2,3)

或者

$result = Db::name('data')

// id 在 5到8之间的

->where('id', 'between', [5, 8])

->select();

dump($result);

生成的SQL语句是：

SELECT \* FROM `think\_data` WHERE `id` BETWEEN 5 AND 8

接下来，使用多个字段查询：

$result = Db::name('data')

// id 在 1到3之间的

->where('id', 'between', [1, 3])

// name 中包含think

->where('name', 'like', '%think%')

->select();

dump($result);

这样生成的查询语句为：

SELECT \* FROM `think\_data` WHERE `id` BETWEEN 1 AND 3 AND `name` LIKE '%think%'

如果要查询某个字段是否为NULL，可以使用：

$result = Db::name('data')

->where('name', 'null')

->select();

dump($result);

这样生成的查询语句为：

SELECT \* FROM `think\_data` WHERE `name` IS NULL

（2）批量查询

批量查询

我们可以使用一个方法完成多个查询条件，例如上面的查询可以改成：

$result = Db::name('data')

->where([

'id' => ['between', '1,3'],

'name' => ['like', '%think%'],

])->select();

dump($result);

这样生成的查询语句还是和之前一样：

SELECT \* FROM `think\_data` WHERE `id` BETWEEN '1' AND '3' AND `name` LIKE '%think%'

我们再来看一些复杂的用法，使用OR和AND混合条件查询，例如：

$result = Db::name('data')

// name 中包含think

->where('name', 'like', '%think%')

->where('id', ['in', [1, 2, 3]], ['between', '5,8'], 'or')

->limit(10)

->select();

dump($result);

生成的SQL语句为：

SELECT \* FROM `think\_data` WHERE ( `id` IN (1,2,3) or `id` BETWEEN '5' AND '8' ) AND `name` LIKE '%think%' LIMIT 10

【 5.1 】使用须知

5.1的数组查询方式有所改变，必须使用下面的方式

$result = Db::name('data')

->where([

['id', 'between', '1,3'],

[ 'name', 'like', '%think%'],

])->select();

dump($result);

（3）快捷查询

快捷查询

如果你有多个字段需要使用相同的查询条件，可以使用快捷查询。例如，我们要查询id和status都大于0的数据，可以使用：

$result = Db::name('data')

->where('id&status', '>', 0)

->limit(10)

->select();

dump($result);

生成的SQL语句为：

SELECT \* FROM `think\_data` WHERE ( `id` > 0 AND `status` > 0 ) LIMIT 10

也可以使用or方式查询，例如：

$result = Db::name('data')

->where('id|status', '>', 0)

->limit(10)

->select();

dump($result);

生成的SQL语句为：

SELECT \* FROM `think\_data` WHERE ( `id` > 0 OR `status` > 0 ) LIMIT 10

（4）视图查询

视图查询

如果需要快捷查询多个表的数据，可以使用视图查询，相当于在数据库创建了一个视图，但仅仅支持查询操作，例如：

$result = Db::view('user','id,name,status')

->view('profile',['name'=>'truename','phone','email'],'profile.user\_id=user.id')

->where('status',1)

->order('id desc')

->select();

dump($result);

生成的SQL语句为：

SELECT user.id,user.name,user.status,profile.name AS truename,profile.phone,profile.email FROM think\_user user INNER JOIN think\_profile profile ON profile.user\_id=user.id WHERE user.status = 1 order by user.id desc

（5）闭包查询

闭包查询

find和select方法可以直接使用闭包查询：

$result = Db::name('data')->select(function ($query) {

$query->where('name', 'like', '%think%')

->where('id', 'in', '1,2,3')

->limit(10);

});

dump($result);

生成的SQL语句是：

SELECT \* FROM `think\_data` WHERE `name` LIKE '%think%' AND `id` IN ('1','2','3') LIMIT 10

（6）获取值和列

获取数值

如果仅仅是需要获取某行表的某个值，可以使用value方法：

// 获取id为8的data数据的name字段值

$name = Db::name('data')

->where('id', 8)

->value('name');

dump($name);

name的结果为：thinkphp

获取列数据

也支持获取某个列的数据，使用column方法，例如：

// 获取data表的name列

$list = Db::name('data')

->where('status', 1)

->column('name');

dump($list);

返回的结果类似下面：

array (size=4)

0 => string 'thinkphp'

1 => string 'onethink'

2 => string 'topthink'

3 => string 'kancloud'

如果希望返回以id为索引的name列数据，可以改成：

// 获取data表的name列 并且以id为索引

$list = Db::name('data')

->where('status', 1)

->column('name', 'id');

dump($list);

返回的结果类似下面：

array (size=4)

1 => string 'thinkphp'

2 => string 'onethink'

3 => string 'topthink'

4 => string 'kancloud'

如果需要返回以主键为索引的数据集，可以使用：

// 获取data表的name列 并且以id为索引

$list = Db::name('data')

->where('status', 1)

->column('\*', 'id');

dump($list);

返回的结果类似下面：

array (size=4)

1 => array (size=3)

'id' => int 1

'name' => string 'thinkphp'

'status'=> int 1

2 => array (size=3)

'id' => int 1

'name' => string 'onethink'

'status'=> int 1

3 => array (size=3)

'id' => int 1

'name' => string 'topthink'

'status'=> int 1

4 => array (size=3)

'id' => int 1

'name' => string 'kancloud'

'status'=> int 1

（7）聚合查询

聚合查询

thinkphp为聚合查询提供了更便捷的方法，如下：

// 统计data表的数据

$count = Db::name('data')

->where('status', 1)

->count();

dump($count);

// 统计user表的最高分

$max = Db::name('user')

->where('status', 1)

->max('score');

dump($max);

支持的聚合查询方法包括：

| **方法** | **说明** | **参数** |
| --- | --- | --- |
| count | 统计数量 | 统计的字段名（可选） |
| max | 获取最大值 | 统计的字段名（必须） |
| min | 获取最小值 | 统计的字段名（必须） |
| avg | 获取平均值 | 统计的字段名（必须） |
| sum | 获取总分 | 统计的字段名（必须） |

（8）时间查询

时间（日期）查询

首先需要在think\_data数据表新增create\_time字段，用于日期查询的字段类型推荐使用datetime类型。

ThinkPHP5.0的查询语言强化了对时间日期查询的支持，例如：

// 查询创建时间大于2016-1-1的数据

$result = Db::name('data')

->whereTime('create\_time', '>', '2016-1-1')

->select();

dump($result);

// 查询本周添加的数据

$result = Db::name('data')

->whereTime('create\_time', '>', 'this week')

->select();

dump($result);

// 查询最近两天添加的数据

$result = Db::name('data')

->whereTime('create\_time', '>', '-2 days')

->select();

dump($result);

// 查询创建时间在2016-1-1~2016-7-1的数据

$result = Db::name('data')

->whereTime('create\_time', 'between', ['2016-1-1', '2016-7-1'])

->select();

dump($result);

日期查询对create\_time字段类型没有要求，可以是int/string/timestamp/datetime/date中的任何一种，系统会自动识别进行处理。

还可以使用下面的人性化日期查询方式，例如：

// 获取今天的数据

$result = Db::name('data')

->whereTime('create\_time', 'today')

->select();

dump($result);

// 获取昨天的数据

$result = Db::name('data')

->whereTime('create\_time', 'yesterday')

->select();

dump($result);

// 获取本周的数据

$result = Db::name('data')

->whereTime('create\_time', 'week')

->select();

dump($result);

// 获取上周的数据

$result = Db::name('data')

->whereTime('create\_time', 'last week')

->select();

dump($result);

（9）字符串查询

字符串查询

在必要的时候，仍然可以使用原生的字符串查询，但建议是**配合参数绑定**一起使用，可以避免注入问题，例如：

$result = Db::name('data')

->where('id > :id AND name IS NOT NULL', ['id' => 10])

->select();

dump($result);

可以直接在where方法中使用字符串查询条件，并支持第二个参数传入参数绑定，上面这个查询生成的SQL语句是：

SELECT \* FROM `think\_data` WHERE ( id > '10' AND name IS NOT NULL )

（10）分块查询

分块查询

分块查询是为查询大量数据的需要而设计的，假如think\_data表已经有超过1万条记录，但是一次性取那么大的数据会导致内存开销非常之大，但确实又有这个需要（例如查询所有的数据并导出到excel），采用分块查询可以缓解这个问题。

使用分块查询，可以把1万条记录分成100次处理，每次处理100条记录，代码示例如下：

Db::name('data')

->where('status', '>', 0)

->chunk(100, function ($list) {

// 处理100条记录

foreach($list as $data){

}

});

第二个参数可以是有效的callback类型，包括使用闭包函数。

系统会按照主键顺序查询，每次查询100条，如果你不希望使用主键进行查询，或者没有主键的话，则需要指定查询的排序字段（但必须是唯一的），例如：

Db::name('user')

->where('status', '>', 0)

->chunk(100, function ($list) {

// 处理100条记录

foreach($list as $data){

}

}, 'uid');

然后交给callback进行数据处理，处理完毕后继续查询下一个100条记录，如果你需要在中途中断后续的查询，只需要在callback方法调用中返回false即可，例如：

Db::name('data')

->where('status','>',0)

->chunk(100,function($list){

foreach($list as $data){

// 返回false则中断后续查询

return false;

}

});

六、模型和关联

快速入门（六）：模型和关联

本章主要帮助大家学习和掌握模型的使用方法，以及通过模型进行关联操作。

ThinkPHP5.0的模型是一种对象-关系映射（Object/Relation Mapping，简称ORM）的封装，并且提供了简洁的ActiveRecord实现。一般来说，每个数据表会和一个“模型”对应。

ORM的基本特性就是表映射到模型，记录映射到模型对象实例，字段映射到对象属性。

模型是一种对象化的操作封装，而不是简单的CURD操作，简单的CURD操作直接使用前面提过的Db类即可。

模型类和Db类的区别主要在于对象及业务逻辑的封装，Db类的查询默认返回的是数组（或者集合），而模型类返回的是当前的模型对象实例（或者集合），模型是比Db类更高级的数据封装，支持模型关联、模型事件和业务（逻辑）方法。

为了演示需要，我们事先添加一些路由定义如下：

return [

// 全局变量规则定义

'\_\_pattern\_\_' => [

'id' => '\d+',

],

'user/index' => 'index/user/index',

'user/create' => 'index/user/create',

'user/add' => 'index/user/add',

'user/add\_list' => 'index/user/addList',

'user/update/:id' => 'index/user/update',

'user/delete/:id' => 'index/user/delete',

'user/:id' => 'index/user/read',

];

（1）模型定义

模型定义

为了更好的理解，我们首先在数据库创建一个think\_user表如下：

CREATE TABLE IF NOT EXISTS `think\_user`(

`id` int(8) unsigned NOT NULL AUTO\_INCREMENT,

`nickname` varchar(50) NOT NULL COMMENT '昵称',

`email` varchar(255) NULL DEFAULT NULL COMMENT '邮箱',

`birthday` int(11) UNSIGNED NOT NULL DEFAULT '0' COMMENT '生日',

`status` tinyint(2) NOT NULL DEFAULT '0' COMMENT '状态',

`create\_time` int(11) UNSIGNED NOT NULL DEFAULT '0' COMMENT '注册时间',

`update\_time` int(11) UNSIGNED NOT NULL DEFAULT '0' COMMENT '更新时间',

PRIMARY KEY (`id`)

) ENGINE=MyISAM DEFAULT CHARSET=utf8 ;

数据库配置文件定义如下：

return [

// 数据库类型

'type' => 'mysql',

// 服务器地址

'hostname' => '127.0.0.1',

// 数据库名

'database' => 'demo',

// 数据库用户名

'username' => 'root',

// 数据库密码

'password' => '',

// 数据库连接端口

'hostport' => '',

// 数据库连接参数

'params' => [],

// 数据库编码默认采用utf8

'charset' => 'utf8',

// 数据库表前缀

'prefix' => 'think\_',

// 数据库调试模式

'debug' => true,

];

我们为think\_user表定义一个User模型（位于application/index/model/User.php）如下：

<?php

namespace app\index\model;

use think\Model;

class User extends Model

{

}

大多情况下，我们无需为模型定义任何的属性和方法即可完成基础的操作。

设置数据表

模型会自动对应一个数据表，规范是：

数据库前缀+当前的模型类名（不含命名空间）

因为模型类命名是驼峰法，所以获取实际的数据表的时候会自动转换为小写+下划线命名的数据表名称。

如果你的模型命名不符合这一数据表对应规范，可以给当前模型定义单独的数据表，包括两种方式。

设置完整数据表：

<?php

namespace app\index\model;

use think\Model;

class User extends Model

{

// 设置完整的数据表（包含前缀）

protected $table = 'think\_user';

}

设置不带前缀的数据表名：

<?php

namespace app\index\model;

use think\Model;

class User extends Model

{

// 设置数据表（不含前缀）

protected $name = 'member';

}

设置数据库连接

如果当前模型类需要使用不同的数据库连接，可以定义模型的connection属性，例如：

<?php

namespace app\index\model;

use think\Model;

class User extends Model

{

// 设置单独的数据库连接

protected $connection = [

// 数据库类型

'type' => 'mysql',

// 服务器地址

'hostname' => '127.0.0.1',

// 数据库名

'database' => 'test',

// 数据库用户名

'username' => 'root',

// 数据库密码

'password' => '',

// 数据库连接端口

'hostport' => '',

// 数据库连接参数

'params' => [],

// 数据库编码默认采用utf8

'charset' => 'utf8',

// 数据库表前缀

'prefix' => 'think\_',

// 数据库调试模式

'debug' => true,

];

}

一般来说，一个应用的模型都是公用的，不区分模块，所以不必在每个模块下面定义模型。

# （2）基础操作

完成基本的模型定义后，我们就可以进行基础的模型操作了，我们来领略下模型的对象化操作的魅力，主要内容包含：

* [新增数据](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/147285#_4)
* [批量新增](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/147285#_113)
* [查询数据](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/147285#_145)
* [数据列表](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/147285#_266)
* [更新数据](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/147285#_357)
* [删除数据](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/147285#_420)

## 新增数据

我们先来看下如何写入模型数据，创建一个User控制器并增加add操作方法如下：

<?php

namespace app\index\controller;

use app\index\model\User as UserModel;

class User

{

// 新增用户数据

public function add()

{

$user = new UserModel;

$user->nickname = '流年';

$user->email = 'thinkphp@qq.com';

$user->birthday = strtotime('1977-03-05');

if ($user->save()) {

return '用户[ ' . $user->nickname . ':' . $user->id . ' ]新增成功';

} else {

return $user->getError();

}

}

}

### 提示：

在当前文件中给app\index\model\User模型定义了一个别名UserModel是为了避免和当前的app\index\controller\User产生冲突，如果你当前的控制器类不是User的话可以不需要定义UserModel别名。

有一种方式可以让你省去别名定义，系统支持统一对控制器类添加Controller后缀，修改配置参数：

// 是否启用控制器类后缀

'controller\_suffix' => true,

然后，控制器类文件改为UserController.php，并且修改控制器类的定义如下：

<?php

namespace app\index\controller;

use app\index\model\User;

class UserController

{

// 新增用户数据

public function add()

{

$user = new User;

$user->nickname = '流年';

$user->email = 'thinkphp@qq.com';

$user->birthday = strtotime('1977-03-05');

if ($user->save()) {

return '用户[ ' . $user->nickname . ':' . $user->id . ' ]新增成功';

} else {

return $user->getError();

}

}

}

接下来，我们访问

http://tp5.com/user/add

如果看到输出

用户[ 流年:1 ]新增成功

表示用户模型写入成功了。

默认情况下，实例化模型类后**第一次执行**的save操作都是执行的数据库insert（新增数据）操作，如果你需要实例化执行save执行数据库的update（更新数据）操作，请确保在save方法之前调用isUpdate方法：

// 强制执行数据更新操作

$user->isUpdate()->save();

如果你觉得上面的方式给User对象一个个赋值太麻烦，可以改为下面的方式：

// 新增用户数据

public function add()

{

$user['nickname'] = '看云';

$user['email'] = 'kancloud@qq.com';

$user['birthday'] = strtotime('2015-04-02');

if ($result = UserModel::create($user)) {

return '用户[ ' . $result->nickname . ':' . $result->id . ' ]新增成功';

} else {

return '新增出错';

}

}

create方法可以传入数组或者标准对象，你可以在外部统一赋值后传入，当然也可以直接传入表单数据（我们后面会有专门的描述）。

我们刷新刚才的访问地址后，页面输出结果为：

用户[ 看云:2 ]新增成功

## 批量新增

也可以直接进行数据的批量新增，给控制器添加如下addList操作方法：

// 批量新增用户数据

public function addList()

{

$user = new UserModel;

$list = [

['nickname' => '张三', 'email' => 'zhanghsan@qq.com', 'birthday' => strtotime('1988-01-15')],

['nickname' => '李四', 'email' => 'lisi@qq.com', 'birthday' => strtotime('1990-09-19')],

];

if ($user->saveAll($list)) {

return '用户批量新增成功';

} else {

return $user->getError();

}

}

访问URL地址

http://tp5.com/user/add\_list

最后的输出结果为：

用户批量新增成功

## 查询数据

接下来添加User模型的查询功能，给User控制器增加如下read操作方法：

// 读取用户数据

public function read($id='')

{

$user = UserModel::get($id);

echo $user->nickname . '<br/>';

echo $user->email . '<br/>';

echo date('Y/m/d', $user->birthday) . '<br/>';

}

模型的get方法用于获取数据表的数据并返回当前的模型对象实例，通常只需要传入主键作为参数，如果没有传入任何值的话，则表示获取第一条数据。

访问如下URL地址

http://tp5.com/user/1

输出结果是：

流年

thinkphp@qq.com

1977/03/05

访问如下URL地址

http://tp5.com/user/2

输出结果是：

看云

kancloud@qq.com

2016/04/02

模型的get方法和Db类的find方法返回结果的区别在于，Db类默认返回的只是数组（注意这里说的默认，其实仍然可以设置为对象），而模型的get方法查询返回的一定是当前的模型对象实例。

但是系统为模型实现了ArrayAccess接口，因此仍然可以通过数组的方式访问对象实例，把控制器的read操作方法改成如下：

// 读取用户数据

public function read($id = '')

{

$user = UserModel::get($id);

echo $user['nickname'] . '<br/>';

echo $user['email'] . '<br/>';

echo date('Y/m/d', $user['birthday']) . '<br/>';

}

再次访问URL地址

http://tp5.com/user/1

不但没有报错，而且最终的输出结果和之前是一样的：

流年

thinkphp@qq.com

1977/03/05

在此提醒一点，如果你是在模型的内部获取数据，请不要使用$this->nickname，而应该使用$this->getAttr('nickname')方式替代。

如果我想通过用户的email来查询模型数据的话，应该如何操作呢？下面是一个查询的例子：

// 根据email读取用户数据

public function read()

{

$user = UserModel::getByEmail('thinkphp@qq.com');

echo $user->nickname . '<br/>';

echo $user->email . '<br/>';

echo date('Y/m/d', $user->birthday) . '<br/>';

}

输出的结果是：

流年

thinkphp@qq.com

1977/03/05

如果不是根据主键查询的话，可以传入数组作为查询条件，例如：

// 根据nickname读取用户数据

public function read()

{

$user = UserModel::get(['nickname'=>'流年']);

echo $user->nickname . '<br/>';

echo $user->email . '<br/>';

echo date('Y/m/d', $user->birthday) . '<br/>';

}

更复杂的查询则可以使用闭包并配合查询构建器来完成，例如：

// 根据nickname读取用户数据

public function read()

{

$user = UserModel::get(function($query){

$query->where('nickname', '流年')->where('id', '>', 10)->order('id','desc');

});

echo $user->nickname . '<br/>';

echo $user->email . '<br/>';

echo date('Y/m/d', $user->birthday) . '<br/>';

}

## 数据列表

如果要查询多个数据，可以使用模型的all方法，我们在控制器中添加index操作方法用于获取用户列表：

// 获取用户数据列表

public function index()

{

$list = UserModel::all();

foreach ($list as $user) {

echo $user->nickname . '<br/>';

echo $user->email . '<br/>';

echo date('Y/m/d', $user->birthday) . '<br/>';

echo '----------------------------------<br/>';

}

}

然后访问

http://tp5.com/user/index

就可以看到输出结果为：

流年

thinkphp@qq.com

1977/03/05

-------------------------------------

看云

kancloud@qq.com

2015/04/02

-------------------------------------

张三

zhanghsan@qq.com

1988/01/15

-------------------------------------

李四

lisi@qq.com

1990/09/19

-------------------------------------

如果不是使用主键查询，可以直接传入数组条件查询，例如：

// 获取用户数据列表

public function index()

{

$list = UserModel::all(['status'=>1]);

foreach ($list as $user) {

echo $user->nickname . '<br/>';

echo $user->email . '<br/>';

echo date('Y/m/d', $user->birthday) . '<br/>';

echo '----------------------------------<br/>';

}

}

同样也可以使用闭包并配合查询构建器完成更多的条件查询，例如：

// 获取用户数据列表

public function index()

{

$list = UserModel::all(function($query){

$query->where('id', '<', 3)->order('id', 'desc');

});

foreach ($list as $user) {

echo $user->nickname . '<br/>';

echo $user->email . '<br/>';

echo date('Y/m/d', $user->birthday) . '<br/>';

echo '----------------------------------<br/>';

}

}

刷新页面访问输出的结果是：

看云

kancloud@qq.com

2015/04/02

-------------------------------------

流年

thinkphp@qq.com

1977/03/05

-------------------------------------

## 更新数据

我们可以对查询出来的数据进行更新操作，下面添加一个update操作方法：

// 更新用户数据

public function update($id)

{

$user = UserModel::get($id);

$user->nickname = '刘晨';

$user->email = 'liu21st@gmail.com';

$user->save();

return '更新用户成功';

}

访问下面的URL地址

http://tp5.com/user/update/1

会输出

更新用户成功

然后我们再次访问

http://tp5.com/user/1

会看到输出结果变成：

刘晨

liu21st@gmail.com

1977/03/05

说明我们的更新操作已经生效了。

默认情况下，查询返回的模型实例如果执行save操作都是执行的数据库update（更新数据）操作，如果你需要实例化执行save执行数据库的insert（新增数据）操作，请确保在save方法之前调用isUpdate方法：

// 强制进行数据新增操作

$user->isUpdate(false)->save();

ActiveRecord模式的更新数据方式需要首先读取对应的数据，如果需要更高效的方法可以把update方法改成：

// 更新用户数据

public function update($id)

{

$user['id'] = (int) $id;

$user['nickname'] = '刘晨';

$user['email'] = 'liu21st@gmail.com';

UserModel::update($user);

return '更新用户成功';

}

## 删除数据

我们给User控制器添加delete方法用于删除用户。

// 删除用户数据

public function delete($id)

{

$user = UserModel::get($id);

if ($user) {

$user->delete();

return '删除用户成功';

} else {

return '删除的用户不存在';

}

}

然后访问

http://tp5.com/user/delete/1

输出结果为：

删除用户成功

如果刷新页面后输出结果为：

删除的用户不存在

同样我们也可以直接使用destroy方法删除模型数据，例如把上面的delete方法改成如下：

// 删除用户数据

public function delete($id)

{

$result = UserModel::destroy($id);

if ($result) {

return '删除用户成功';

} else {

return '删除的用户不存在';

}

}

目前为止，你已经掌握了最基本的模型操作，后面会引申一些高级的用法。

（3）读取器和修改器

读取器

前面读取用户生日的时候，使用了date方法进行日期的格式处理输出，但是每次读取数据后都需要这样处理就显得非常麻烦。

使用读取器功能就可以简化类似的数据处理操作，例如，我们给User模型添加读取器的定义方法。

<?php

namespace app\index\model;

use think\Model;

class User extends Model

{

// birthday读取器

protected function getBirthdayAttr($birthday)

{

return date('Y-m-d', $birthday);

}

}

这里，我们添加了一个getBirthdayAttr读取器方法用于读取User模型的birthday属性的值，该方法会在读取birthday属性值的时候自动执行。

读取器方法的命名规范是：

get + 属性名的驼峰命名+ Attr

所以，getBirthdayAttr读取器读取的是birthday属性，而getUserBirthdayAttr 读取器读取的则是user\_birthday属性。

定义完获取器后，修改控制器的read操作方法如下：

// 读取用户数据

public function read($id='')

{

$user = UserModel::get($id);

echo $user->nickname . '<br/>';

echo $user->email . '<br/>';

echo $user->birthday . '<br/>';

}

访问URL地址

http://tp5.com/user/1

最后的输出结果为：

流年

thinkphp@qq.com

1977-03-05

读取器还可以定义读取数据表中不存在的属性，例如把原始生日和转换的格式分开两个属性birthday和user\_birthday，我们只需定义user\_birthday属性的读取器方法：

<?php

namespace app\index\model;

use think\Model;

class User extends Model

{

// user\_birthday读取器

protected function getUserBirthdayAttr($value,$data)

{

return date('Y-m-d', $data['birthday']);

}

}

这里的读取器方法使用了第二个参数，表示传入所有的属性数据。因为原始的user\_birthday属性数据是不存在的，所以我们需要通过data参数获取。

read操作方法修改为：

// 读取用户数据

public function read($id='')

{

$user = UserModel::get($id);

echo $user->nickname . '<br/>';

echo $user->email . '<br/>';

echo $user->birthday . '<br/>';

echo $user->user\_birthday . '<br/>';

}

当刷新页面的时候，最终输出的结果为：

流年

thinkphp@qq.com

226339200

1977-03-05

修改器

由于birthday属性是时间戳（整型）格式的，因此我们必须在写入数据前进行时间戳转换，前面使用的方法是每次赋值的时候进行转换处理：

$user['birthday'] = strtotime('2015-04-02');

为了避免每次都进行日期格式的转换操作，可以定义修改器方法来自动处理，修改User模型如下：

<?php

namespace app\index\model;

use think\Model;

class User extends Model

{

// 读取器

protected function getUserBirthdayAttr($birthday, $data)

{

return date('Y-m-d', $data['birthday']);

}

// birthday修改器

protected function setBirthdayAttr($value)

{

return strtotime($value);

}

}

修改器方法的命名规范是：

set + 属性名的驼峰命名+ Attr

所以，setBirthdayAttr方法修改的是birthday属性，而setUserBirthdayAttr 方法修改的则是user\_birthday属性。

控制器的add操作方法修改如下：

// 新增用户数据

public function add()

{

$user = new UserModel;

$user->nickname = '流年';

$user->email = 'thinkphp@qq.com';

$user->birthday = '1977-03-05';

if ($user->save()) {

return '用户[ ' . $user->nickname . ':' . $user->id . ' ]新增成功';

} else {

return $user->getError();

}

}

访问URL地址：

http://tp5.com/user/add

最后的输出结果为：

用户[ 流年:10 ]新增成功

接着我们访问

http://tp5.com/user/10

页面输出结果为：

流年

thinkphp@qq.com

1977-03-05

通过定义修改器和读取器，完成了时间戳方式存储的birthday属性的写入和读取的自动处理。

（4）类型转换和自动完成

类型转换

对于前面的时间戳birthday的例子，还可以进行进一步的简化，这里需要用到类型强制转换的功能，在User模型类中添加定义：

<?php

namespace app\index\model;

use think\Model;

class User extends Model

{

protected $dateFormat = 'Y/m/d';

protected $type = [

// 设置birthday为时间戳类型（整型）

'birthday' => 'timestamp',

];

}

不需要定义任何修改器和读取器，我们完成了相同的功能。

对于timestamp和datetime类型，如果不设置模型的dateFormat属性，默认的日期显示格式为：Y-m-d H:i:s，或者也可以显示的设置日期格式，例如：

<?php

namespace app\index\model;

use think\Model;

class User extends Model

{

protected $type = [

// 设置birthday为时间戳类型（整型）

'birthday' => 'timestamp:Y/m/d',

];

}

提示：

对于简单的数据格式转换之类的处理，设置类型转换比定义修改器和读取器更加方便。

ThinkPHP5.0支持的转换类型包括：

| **类型** | **描述** |
| --- | --- |
| integer | 整型 |
| float | 浮点型 |
| boolean | 布尔型 |
| array | 数组 |
| json | JSON类型 |
| object | 对象 |
| datetime | 日期时间 |
| timestamp | 时间戳（整型） |
| serialize | 序列化 |

自动时间戳

对于固定的时间戳和时间日期型的字段，比如文章的创建时间、修改时间等字段，还有比设置类型转换更简单的方法，尤其是所有的数据表统一处理的话，只需要在数据库配置文件中添加设置：

// 开启自动写入时间戳字段

'auto\_timestamp' => true,

再次访问

http://tp5.com/user/add

会发现系统已经自动写入了think\_user数据表中的的create\_time、update\_time字段，如果自动写入的时间戳字段不是这两个的话，需要修改模型类的属性定义，例如：

<?php

namespace app\index\model;

use think\Model;

class User extends Model

{

// 定义类型转换

protected $type = [

'birthday' => 'timestamp:Y/m/d',

];

// 定义时间戳字段名

protected $createTime = 'create\_at';

protected $updateTime = 'update\_at';

}

如果个别数据表不需要自动写入时间戳字段的话，也可以在模型里面直接关闭：

<?php

namespace app\index\model;

use think\Model;

class User extends Model

{

// 定义类型转换

protected $type = [

'birthday' => 'timestamp:Y/m/d',

];

// 关闭自动写入时间戳

protected $autoWriteTimestamp = false;

}

关闭自动写入时间戳后，我们再次访问URL地址：

http://tp5.com/user/add

重新生成的数据已经没有自动写入时间戳了，而是数据库默认值写入。

默认的时间戳字段类型是整型，如果需要使用其它的时间字段类型，可以做如下设置：

<?php

namespace app\index\model;

use think\Model;

class User extends Model

{

// 定义类型转换

protected $type = [

'birthday' => 'timestamp:Y/m/d',

];

// 指定自动写入时间戳的类型为dateTime类型

protected $autoWriteTimestamp = 'datetime';

}

如果全局的自动时间戳的类型是统一的，也可以直接在数据库配置文件中设置：

// 开启自动写入时间戳字段

'auto\_timestamp' => 'datetime',

如上设置后，你的think\_user数据表中的create\_time和 update\_time 字段类型就必须更改为datetime类型的格式。

支持设置的时间戳类型包含：datetime、date和timestamp。

自动完成

系统已经自动写入了think\_user数据表中的的create\_time、update\_time字段，如果我们希望自动写入其它的字段，则可以使用自动完成功能，例如下面实现新增的时候自动写入status字段。

<?php

namespace app\index\model;

use think\Model;

class User extends Model

{

// 定义类型转换

protected $type = [

'birthday' => 'timestamp:Y/m/d',

];

// 定义自动完成的属性

protected $insert = ['status' => 1];

}

除了insert属性之外，自动完成共有三个属性定义，分别是：

| **属性** | **描述** |
| --- | --- |
| auto | 新增及更新的时候自动完成的属性数组 |
| insert | 仅新增的时候自动完成的属性数组 |
| update | 仅更新的时候自动完成的属性数组 |

自动完成属性里面一般来说仅仅需要定义属性的名称，然后配合修改器或者类型转换来一起完成，如果写入的是一个固定的值，就无需使用修改器。status属性的自动写入可以直接使用：

'status' => 1

完成后，我们访问URL地址：

http://tp5.com/user/add

最后的输出结果为：

用户[ 流年:12 ]新增成功

为了便于看到效果，我们修改控制器的read操作方法输出更多的属性：

// 读取用户数据

public function read($id='')

{

$user = UserModel::get($id);

echo $user->nickname . '<br/>';

echo $user->email . '<br/>';

echo $user->birthday . '<br/>';

echo $user->status . '<br/>';

echo $user->create\_time . '<br/>';

echo $user->update\_time . '<br/>';

}

然后，访问URL地址：

http://tp5.com/user/12

最后的输出结果为：

流年

thinkphp@qq.com

1977/03/05

1

2016-05-02 16:21:33

2016-05-02 16:21:33

可以看到status、create\_time和update\_time都实现了自动写入。

如果你的status属性的值不是固定的，而是需要条件判断，那么我们可以定义修改器来配合自动完成。

<?php

namespace app\index\model;

use think\Model;

class User extends Model

{

// 定义类型转换

protected $type = [

'birthday' => 'timestamp:Y/m/d',

];

// 定义自动完成的属性

protected $insert = ['status'];

// status属性修改器

protected function setStatusAttr($value, $data)

{

return '流年' == $data['nickname'] ? 1 : 2;

}

// status属性读取器

protected function getStatusAttr($value)

{

$status = [-1 => '删除', 0 => '禁用', 1 => '正常', 2 => '待审核'];

return $status[$value];

}

}

我们访问下面的URL地址进行批量新增

http://tp5.com/user/add\_list

之后，访问

http://tp5.com/user/18

最后的输出结果为：

张三

zhanghsan@qq.com

1988/01/15

待审核

2016-05-02 16:40:57

2016-05-02 16:40:57

（5）查询范围

查询范围

对于一些常用的查询条件，我们可以封装成查询范围来进行方便的调用。

例如，邮箱地址为thinkphp@qq.com和status为1这两个常用查询条件，可以定义为模型类的两个查询范围方法：

<?php

namespace app\index\model;

use think\Model;

class User extends Model

{

// 定义类型转换

protected $type = [

'birthday' => 'timestamp:Y/m/d',

];

// 定义自动完成的属性

protected $insert = ['status'];

// status修改器

protected function setStatusAttr($value, $data)

{

return '流年' == $data['nickname'] ? 1 : 2;

}

// status读取器

protected function getStatusAttr($value)

{

$status = [-1 => '删除', 0 => '禁用', 1 => '正常', 2 => '待审核'];

return $status[$value];

}

// email查询

public function scopeEmail($query)

{

$query->where('email', 'thinkphp@qq.com');

}

// status查询

public function scopeStatus($query)

{

$query->where('status', 1);

}

}

查询范围方法的定义规范为：

scope + 查询范围名称

我们修改控制器的index方法如下：

// 根据查询范围获取用户数据列表

public function index()

{

$list = UserModel::scope('email,status')->select();

foreach ($list as $user) {

echo $user->nickname . '<br/>';

echo $user->email . '<br/>';

echo $user->birthday . '<br/>';

echo $user->status . '<br/>';

echo '-------------------------------------<br/>';

}

}

最后查询的SQL语句是：

SELECT \* FROM `think\_user` WHERE `email` = 'thinkphp@qq.com' AND `status` = 1

scope方法支持使用闭包，例如：

// 根据查询范围获取用户数据列表

public function index()

{

//

$list = UserModel::scope(function ($query) {

$query->where('email','thinkphp@qq.com')

->where('status',1)

->order('id', 'desc');

})

->select();

foreach ($list as $user) {

echo $user->nickname . '<br/>';

echo $user->email . '<br/>';

echo $user->birthday . '<br/>';

echo $user->status . '<br/>';

echo '-------------------------------------<br/>';

}

}

上面的scope方法使用了闭包，闭包里面支持所有的链式操作方法。

最后生成的SQL语句是：

SELECT \* FROM `think\_user` WHERE `email` = 'thinkphp@qq.com' AND `status` = 1 ORDER BY `id` desc

查询范围方法支持额外的参数，例如scopeEmail方法改为：

// email查询

public function scopeEmail($query, $email = '')

{

$query->where('email', $email);

}

查询范围的方法的第一个参数必须是查询对象，并且支持多个额外参数。

然后，使用下面的方式调用即可：

$list = UserModel::scope('email','thinkphp@qq.com')->select();

全局查询范围

可以给模型定义全局的查询范围，在模型类添加一个静态的base方法即可，例如我们给模型类增加一个全局查询范围，用于查询状态为1的数据：

<?php

namespace app\index\model;

use think\Model;

class User extends Model

{

// 定义类型转换

protected $type = [

'birthday' => 'timestamp:Y/m/d',

];

// 定义自动完成的属性

protected $insert = ['status'];

// status修改器

protected function setStatusAttr($value, $data)

{

return '流年' == $data['nickname'] ? 1 : 2;

}

// status读取器

protected function getStatusAttr($value)

{

$status = [-1 => '删除', 0 => '禁用', 1 => '正常', 2 => '待审核'];

return $status[$value];

}

// email查询

public function scopeEmail($query)

{

$query->where('email', 'thinkphp@qq.com');

}

// 全局查询范围

protected static function base($query)

{

// 查询状态为1的数据

$query->where('status',1);

}

}

当使用下面的查询操作

UserModel::scope('email')->select();

最后生成的SQL语句是：

SELECT \* FROM `think\_user` WHERE `email` = 'thinkphp@qq.com' AND `status` = 1 ORDER BY `id` desc

每次查询都会自动带上全局查询范围的查询条件。

# （6）输入和验证

现在我们来进一步使用表单提交数据完成模型的对象操作，主要内容包含：

* [表单提交](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/147289#_4)
* [表单验证](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/147289#_145)
  + [【 5.1 】使用须知](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/147289#_51__242)
* [错误提示](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/147289#_262)
* [自定义验证规则](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/147289#_328)
* [控制器验证](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/147289#_389)

## 表单提交

首先创建一个视图模板文件 application/index/view/user/create.html，内容如下：

<!doctype html>

<html>

<head>

<meta charset="UTF-8">

<title>创建用户</title>

<style>

body {

font-family:"Microsoft Yahei","Helvetica Neue",Helvetica,Arial,sans-serif;

font-size:16px;

padding:5px;

}

.form{

padding: 15px;

font-size: 16px;

}

.form .text {

padding: 3px;

margin:2px 10px;

width: 240px;

height: 24px;

line-height: 28px;

border: 1px solid #D4D4D4;

}

.form .btn{

margin:6px;

padding: 6px;

width: 120px;

font-size: 16px;

border: 1px solid #D4D4D4;

cursor: pointer;

background:#eee;

}

a{

color: #868686;

cursor: pointer;

}

a:hover{

text-decoration: underline;

}

h2{

color: #4288ce;

font-weight: 400;

padding: 6px 0;

margin: 6px 0 0;

font-size: 28px;

border-bottom: 1px solid #eee;

}

div{

margin:8px;

}

.info{

padding: 12px 0;

border-bottom: 1px solid #eee;

}

.copyright{

margin-top: 24px;

padding: 12px 0;

border-top: 1px solid #eee;

}

</style>

</head>

<body>

<h2>创建用户</h2>

<FORM method="post" class="form" action="{:url('index/user/add')}">

昵 称：<INPUT type="text" class="text" name="nickname"><br/>

邮 箱：<INPUT type="text" class="text" name="email"><br/>

生 日：<INPUT type="text" class="text" name="birthday"><br/>

<input type="hidden" name="\_\_token\_\_" value="{$Request.token}" />

<INPUT type="submit" class="btn" value=" 提交 ">

</FORM>

<div class="copyright">

<a title="官方网站" href="http://www.thinkphp.cn">ThinkPHP</a>

<span>V5</span>

<span>{ 十年磨一剑-为API开发设计的高性能框架 }</span>

</div>

</body>

</html>

User控制器增加新的操作方法create如下：

// 创建用户数据页面

public function create()

{

return view();

}

view方法是系统封装的助手函数用于快速渲染模板文件，这里没有传入模板文件，则按照系统默认的解析规则会自动渲染当前操作方法对应的模板文件，也就是默认视图目录（application/index/view）下面的user/create.html文件，所以如果改成下面的方式是相同的：

// 创建用户数据页面

public function create()

{

return view('user/create');

}

并且修改之前的add方法如下：

// 新增用户数据

public function add()

{

$user = new UserModel;

if ($user->allowField(true)->save(input('post.'))) {

return '用户[ ' . $user->nickname . ':' . $user->id . ' ]新增成功';

} else {

return $user->getError();

}

}

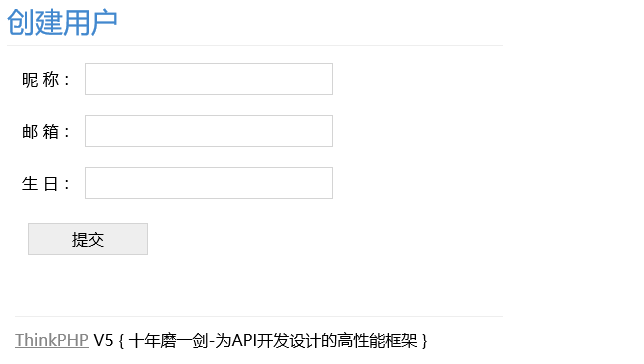
#### 注意

这里使用allowField(true)是为了避免表单令牌验证的字段被写入数据表，如果你已经在模型里面定义了field属性的话，可以不需要。

我们访问URL地址：

http://tp5.com/user/create

页面输出如图：



输入用户信息后，点击提交按钮：



页面显示结果为：

用户[ 流年:30 ]新增成功

## 表单验证

永远不要相信用户的数据，所以现在给表单提交添加数据验证。

我们添加一个User验证器（位于application/index/validate/User.php），代码如下：

<?php

namespace app\index\validate;

use think\Validate;

class User extends Validate

{

// 验证规则

protected $rule = [

'nickname' => 'require|min:5|token',

'email' => 'require|email',

'birthday' => 'dateFormat:Y-m-d',

];

}

User验证器添加了三个属性的验证规则，分别表示：

* 昵称必须，而且最小长度为5
* 邮箱必须，而且必须是合法的邮件地址
* 生日可选，如果填写的话必须为 Y-m-d格式的日期格式

对属性可以使用多个验证规则，除非使用了require开头的规则，否则所有的验证都是可选的（也就是说有值才验证），多个验证之间用|分割，并且按照先后顺序依次进行验证，一旦某个规则验证失败，后续的规则就不会再进行验证（除非设置批量验证方式则统一返回所有的错误信息）。

更多的内置规则可以参考完全开发手册的[内置规则](http://www.kancloud.cn/manual/thinkphp5/129356)一节。

如果我们的验证规则里面使用了|，为了避免混淆则必须用数组方式定义验证规则，验证规则定义修改如下：

<?php

namespace app\index\validate;

use think\Validate;

class User extends Validate

{

// 验证规则

protected $rule = [

'nickname' => ['require', 'min'=>5, 'token'],

'email' => ['require', 'email'],

'birthday' => ['dateFormat' => 'Y|m|d'],

];

}

然后对控制器的add方法则稍加修改，在save方法之前添加一个validate方法即可：

// 新增用户数据

public function add()

{

$user = new UserModel;

if ($user->allowField(true)->validate(true)->save(input('post.'))) {

return '用户[ ' . $user->nickname . ':' . $user->id . ' ]新增成功';

} else {

return $user->getError();

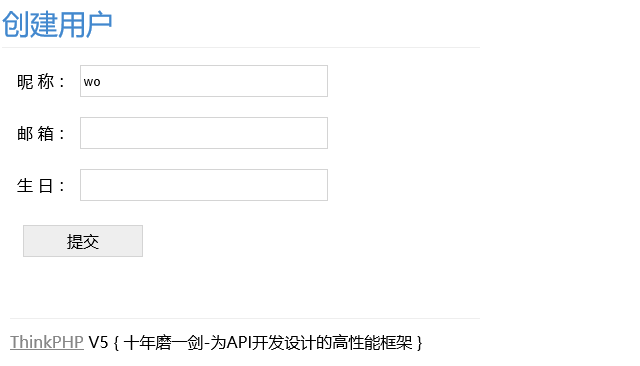
}

}

当我们没有输入任何表单数据就直接提交的话，页面会输出结果：

nickname不能为空

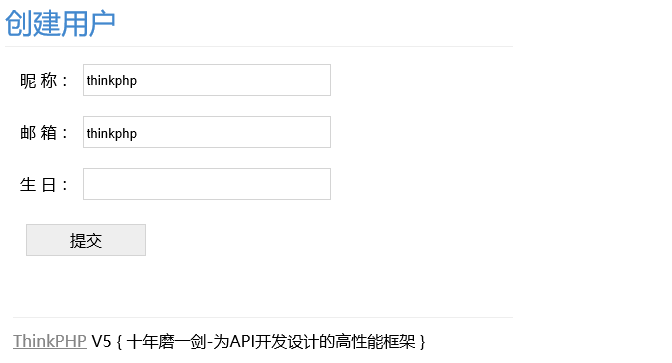
当我们输入昵称为wo的时候点击提交



页面输出结果为：

nickname长度不能小于 5

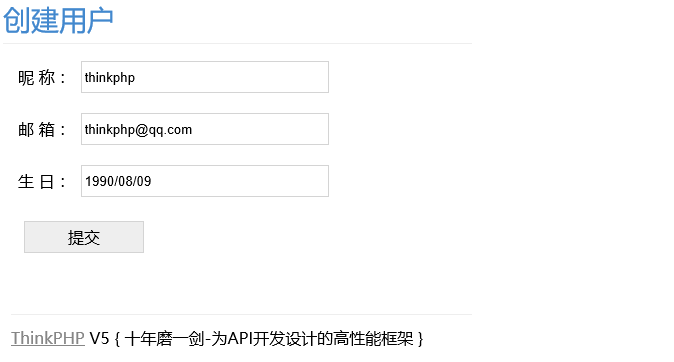
当输入一个错误的邮箱格式后提交



页面提示错误信息为：

email格式不符

当输入一个 1990/08/09 的生日时候



页面提示的错误信息是：

birthday必须使用日期格式：Y-m-d

### 【 5.1 】使用须知

5.1取消了模型验证功能，请在控制器中进行数据验证

上面的add方法可以改成

// 新增用户数据

public function add()

{

$check = $this->validate(input('post.'),'User');

if (true === $check) {

$user = new UserModel;

$user->allowField(true)->save(input('post.'));

return '用户[ ' . $user->nickname . ':' . $user->id . ' ]新增成功';

} else {

return $check;

}

}

## 错误提示

目前为止，提示的错误信息都是系统默认的，接下来我们来定义提示信息。

首先，如果只是希望修改属性名称的话，可以直接使用下面的验证规则定义方式：

<?php

namespace app\index\validate;

use think\Validate;

class User extends Validate

{

// 验证规则

protected $rule = [

'nickname|昵称' => 'require|min:5',

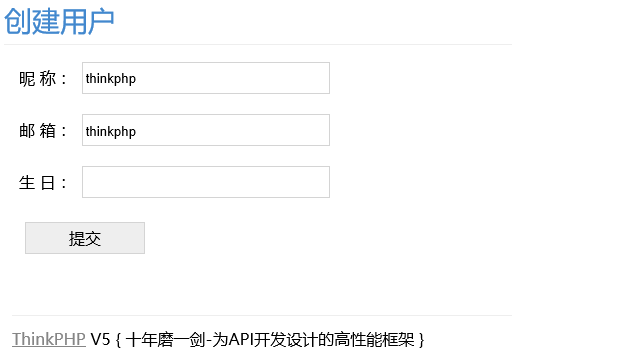
'email|邮箱' => 'require|email',

'birthday|生日' => 'dateFormat:Y-m-d',

];

}

现在我们提交一个错误的邮箱，



提示的错误信息为：

邮箱格式不符

输入错误的生日格式的时候，提示的错误信息为：

生日必须使用日期格式：Y-m-d

如果希望完整定义错误提示信息的话，可以使用：

<?php

namespace app\index\validate;

use think\Validate;

class User extends Validate

{

// 验证规则

protected $rule = [

['nickname', 'require|min:5', '昵称必须|昵称不能短于5个字符'],

['email', 'email', '邮箱格式错误'],

['birthday', 'dateFormat:Y-m-d', '生日格式错误'],

];

}

现在我们提交一个错误的邮箱，提示的错误信息为：

邮箱格式错误

提交一个错误的生日格式后，提示的错误信息变成：

生日格式错误

系统提供了丰富的内置验证规则，具体可以参考完全开发手册。

## 自定义验证规则

系统的验证规则可以满足大部分的验证场景，但有时候我们也需要自定义特殊的验证规则，例如我们需要验证邮箱必须为thinkphp.cn域名的话，可以在User验证器中添加验证规则如下：

<?php

namespace app\index\validate;

use think\Validate;

class User extends Validate

{

// 验证规则

protected $rule = [

['nickname', 'require|min:5', '昵称必须|昵称不能短于5个字符'],

['email', 'checkMail:thinkphp.cn', '邮箱格式错误'],

['birthday', 'dateFormat:Y-m-d', '生日格式错误'],

];

// 验证邮箱格式 是否符合指定的域名

protected function checkMail($value, $rule)

{

return 1 === preg\_match('/^\w+([-+.]\w+)\*@' . $rule . '$/', $value);

}

}

自定义验证规则也支持返回动态的错误信息，只需要在验证方法里面返回错误信息字符串即可，例如：

<?php

namespace app\index\validate;

use think\Validate;

class User extends Validate

{

// 验证规则

protected $rule = [

['nickname', 'require|min:5', '昵称必须|昵称不能短于5个字符'],

['email', 'checkMail:thinkphp.cn', '邮箱格式错误'],

['birthday', 'dateFormat:Y-m-d', '生日格式错误'],

];

// 验证邮箱格式 是否符合指定的域名

protected function checkMail($value, $rule)

{

$result = preg\_match('/^\w+([-+.]\w+)\*@' . $rule . '$/', $value);

if (!$result) {

return '邮箱只能是' . $rule . '域名';

} else {

return true;

}

}

}

如果输入了一个不是thinkphp.cn域名的邮箱地址，会提示如下错误信息：

邮箱只能是thinkphp.cn域名

## 控制器验证

前面我们讲了在模型中使用验证器进行数据验证的方法，下面来讲下如何在控制器中进行数据验证。

验证器类的定义不变，现在修改下控制器类：

namespace app\index\controller;

use app\index\model\User as UserModel;

use think\Controller;

class User extends Controller

{

// 创建用户数据页面

public function create()

{

return view();

}

public function add()

{

$data = input('post.');

// 数据验证

$result = $this->validate($data,'User');

if (true !== $result) {

return $result;

}

$user = new UserModel;

// 数据保存

$user->allowField(true)->save($data);

return '用户[ ' . $user->nickname . ':' . $user->id . ' ]新增成功';

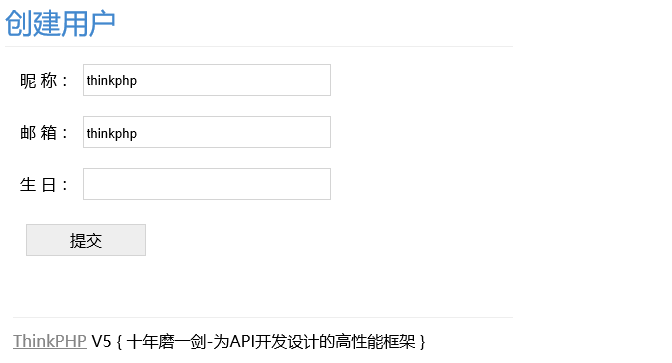
}

}

然后访问

http://tp5.com/user/create

当输入一个错误的邮箱格式后提交



页面提示错误信息为：

email格式不符

如果有一些个别的验证没有在验证器里面定义，也可以使用静态方法单独处理，例如下面对birthday字段单独验证是否是一个有效的日期格式：

namespace app\index\controller;

use app\index\model\User as UserModel;

use think\Controller;

use think\Validate;

class User extends Controller

{

// 创建用户数据页面

public function create()

{

return view();

}

public function add()

{

$data = input('post.');

// 验证birthday是否有效的日期

$check = Validate::is($data['birthday'],'date');

if (false === $check) {

return 'birthday日期格式非法';

}

$user = new UserModel;

// 数据保存

$user->save($data);

return '用户[ ' . $user->nickname . ':' . $user->id . ' ]新增成功';

}

}

# （7）关联

本篇为您讲述ThinkPHP5.0的最常用的几个关联的定义和使用，主要包括：

* [基本定义](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/145257#_4)
* [一对一关联](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/145257#_49)
  + [关联定义](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/145257#_79)
  + [关联写入](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/145257#_166)
  + [关联查询](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/145257#_237)
  + [关联更新](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/145257#_277)
  + [关联删除](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/145257#_305)
* [一对多关联](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/145257#_334)
  + [关联定义](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/145257#_352)
  + [关联新增](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/145257#_438)
  + [关联查询](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/145257#_467)
  + [关联更新](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/145257#_518)
  + [关联删除](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/145257#_540)
* [多对多关联](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/145257#_563)
  + [关联定义](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/145257#_585)
  + [关联新增](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/145257#_649)
  + [关联删除](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/145257#_701)
  + [关联查询](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/145257#_730)

## 基本定义

ThinkPHP5.0的关联采用了对象化的操作模式，你无需继承不同的模型类，只是把关联定义成一个方法，并且直接通过当前模型对象的属性名获取定义的关联数据。

举个例子，有一个用户模型User，有一个关联的模型对象Book，每个用户有多本书，User模型定义如下：

<?php

namespace app\index\model;

use think\Model;

class User extends Model

{

// 定义关联方法

public function books()

{

return $this->hasMany('Book');

}

}

#### 提示：

User模型的books方法就是一个关联定义方法，方法名可以随意命名（使用驼峰法命名），但注意要避免和User模型对象的字段属性冲突。

实际获取关联数据的时候，就是采用下面的方式：

$user = User::get(5);

// 获取User对象的nickname属性

echo $user->nickname;

// 获取User对象的Book关联对象

dump($user->books);

// 执行关联的Book对象的查询

$user->books()->where('name','thinkphp')->find();

对于涉及的代码用法，目前不用深究，后面会详细描述。

一般来说，关联关系包括：

* 一对一关联：HAS\_ONE 以及相对的 BELONGS\_TO
* 一对多关联：HAS\_MANY 以及相对的 BELONGS\_TO
* 多对多关联：BELONGS\_TO\_MANY

后面会详细讲解每一种关联的用法。

## 一对一关联

一对一关联是一种最简单的关联，例如每个用户都有一份档案，每个公司都有一个营业执照等等。

在这之前，我们先创建数据表如下：

DROP TABLE IF EXISTS `think\_user`;

CREATE TABLE IF NOT EXISTS `think\_user` (

`id` int(6) UNSIGNED NOT NULL AUTO\_INCREMENT,

`nickname` varchar(25) NOT NULL,

`name` varchar(25) NOT NULL,

`password` varchar(50) NOT NULL,

`create\_time` int(11) UNSIGNED NOT NULL,

`update\_time` int(11) UNSIGNED NOT NULL,

`status` tinyint(1) DEFAULT 0,

PRIMARY KEY (`id`)

) ENGINE=MyISAM DEFAULT CHARSET=utf8;

DROP TABLE IF EXISTS `think\_profile`;

CREATE TABLE IF NOT EXISTS `think\_profile` (

`id` int(6) UNSIGNED NOT NULL AUTO\_INCREMENT,

`truename` varchar(25) NOT NULL,

`birthday` int(11) NOT NULL,

`address` varchar(255) DEFAULT NULL,

`email` varchar(255) DEFAULT NULL,

`user\_id` int(6) UNSIGNED NOT NULL,

PRIMARY KEY (`id`)

) ENGINE=MyISAM DEFAULT CHARSET=utf8;

### 关联定义

我们以用户和档案的一对一关联为例，在User模型类中添加关联定义方法，然后在方法中调用hasOne方法即可：

<?php

namespace app\index\model;

use think\Model;

class User extends Model

{

// 开启自动写入时间戳

protected $autoWriteTimestamp = true;

// 定义自动完成的属性

protected $insert = ['status' => 1];

// 定义关联方法

public function profile()

{

// 用户HAS ONE档案关联

return $this->hasOne('Profile');

}

}

hasOne方法有5个参数，依次分别是：

#### hasOne('关联模型名','关联外键','主键','别名定义','join类型')

默认的外键是：**当前模型名\_id**，主键则是自动获取，如果你的表设计符合这一规范的话，只需要设置关联的模型名即可.

通常关联模型和当前模型都是相同的命名空间，如果关联模型在不同的命名空间，需要指定完整的类名，例如：

// 关联admin模块下面的模型对象

return $this->hasOne('\app\admin\Profile');

在关联查询的时候，默认使用当前模型的名称（小写）作为数据表别名，可以指定查询使用的数据表别名，例如：

// 用户HAS ONE档案关联

return $this->hasOne('Profile','user\_id','id',['user'=>'member','profile'=>'info']);

要进行模型的关联操作，我们必须同时定义好关联模型，Profile模型定义如下：

<?php

namespace app\index\model;

use think\Model;

class Profile extends Model

{

protected $type = [

'birthday' => 'timestamp:Y-m-d',

];

}

可以看到Profile模型中并没有定义关联方法。如果你的关联操作都是基于User模型的话，Profile模型中并不需要定义关联方法。

如果你需要基于Profile模型来进行关联操作，则需要在Profile模型中定义对应的BELONGS\_TO关联，如下：

<?php

namespace app\index\model;

use think\Model;

class Profile extends Model

{

protected $type = [

'birthday' => 'timestamp:Y-m-d',

];

public function user()

{

// 档案 BELONGS TO 关联用户

return $this->belongsTo('User');

}

}

belongsTo方法和hasOne一样，也有5个参数：

#### belongsTo('关联模型名','关联外键','关联模型主键','别名定义','join类型')

### 关联写入

首先来看下如何进行关联数据的写入，创建User控制器的add操作方法如下：

<?php

namespace app\index\controller;

use app\index\model\Profile;

use app\index\model\User as UserModel;

class User

{

// 关联新增数据

public function add()

{

$user = new UserModel;

$user->name = 'thinkphp';

$user->password = '123456';

$user->nickname = '流年';

if ($user->save()) {

// 写入关联数据

$profile = new Profile;

$profile->truename = '刘晨';

$profile->birthday = '1977-03-05';

$profile->address = '中国上海';

$profile->email = 'thinkphp@qq.com';

$user->profile()->save($profile);

return '用户新增成功';

} else {

return $user->getError();

}

}

}

关联模型的写入调用了关联方法profile()，该方法返回的是一个Relation对象，执行save方法会自动传入当前模型User的主键作为关联键值，所以不需要手动传入Profile模型的user\_id属性。

save方法也可以直接使用数组而不是Profile对象，例如：

<?php

namespace app\index\controller;

use app\index\model\Profile;

use app\index\model\User as UserModel;

class User

{

// 关联新增数据

public function add()

{

$user = new UserModel;

$user->name = 'thinkphp';

$user->password = '123456';

$user->nickname = '流年';

if ($user->save()) {

// 写入关联数据

$profile['truename'] = '刘晨';

$profile['birthday'] = '1977-03-05';

$profile['address'] = '中国上海';

$profile['email'] = 'thinkphp@qq.com';

$user->profile()->save($profile);

return '用户[ ' . $user->name . ' ]新增成功';

} else {

return $user->getError();

}

}

}

### 关联查询

一对一的关联查询很简单，直接把关联对象当成属性来用即可，例如：

public function read($id)

{

$user = UserModel::get($id);

echo $user->name . '<br/>';

echo $user->nickname . '<br/>';

echo $user->profile->truename . '<br/>';

echo $user->profile->email . '<br/>';

}

访问URL地址：

http://tp5.com/user/1

最后输出结果为：

thinkphp

流年

刘晨

thinkphp@qq.com

以上关联查询的时候，只有在获取关联对象（$user->profile）的时候才会进行实际的关联查询，缺点是会可能进行多次查询，但可以使用预载入查询来提高查询性能，对于一对一关联来说，只需要进行一次查询即可获取关联对象数据，例如：

public function read($id)

{

$user = UserModel::get($id,'profile');

echo $user->name . '<br/>';

echo $user->nickname . '<br/>';

echo $user->profile->truename . '<br/>';

echo $user->profile->email . '<br/>';

}

get方法使用第二个参数就表示进行关联预载入查询。

### 关联更新

一对一的关联更新如下：

public function update($id)

{

$user = UserModel::get($id);

$user->name = 'framework';

if ($user->save()) {

// 更新关联数据

$user->profile->email = 'liu21st@gmail.com';

$user->profile->save();

return '用户[ ' . $user->name . ' ]更新成功';

} else {

return $user->getError();

}

}

访问URL地址：

http://tp5.com/user/update/1

最后输出结果为：

用户更新成功

### 关联删除

关联删除代码如下：

public function delete($id)

{

$user = UserModel::get($id);

if ($user->delete()) {

// 删除关联数据

$user->profile->delete();

return '用户[ ' . $user->name . ' ]删除成功';

} else {

return $user->getError();

}

}

访问URL地址：

http://tp5.com/user/delete/1

页面输出结果为：

用户删除成功

## 一对多关联

每个作者写有多本书就是一个典型的一对多关联，首先创建如下数据表：

DROP TABLE IF EXISTS `think\_book`;

CREATE TABLE IF NOT EXISTS `think\_book` (

`id` int(8) UNSIGNED NOT NULL AUTO\_INCREMENT,

`title` varchar(255) NOT NULL,

`publish\_time` int(11) UNSIGNED DEFAULT NULL,

`create\_time` int(11) UNSIGNED NOT NULL,

`update\_time` int(11) UNSIGNED NOT NULL,

`status` tinyint(1) NOT NULL,

`user\_id` int(6) UNSIGNED NOT NULL,

PRIMARY KEY (`id`)

) ENGINE=MyISAM DEFAULT CHARSET=utf8;

### 关联定义

在User模型类添加Book关联如下：

<?php

namespace app\index\model;

use think\Model;

class User extends Model

{

// 开启自动写入时间戳

protected $autoWriteTimestamp = true;

// 定义自动完成的属性

protected $insert = ['status' => 1];

// 定义关联方法

public function profile()

{

return $this->hasOne('Profile');

}

// 定义关联

public function books()

{

return $this->hasMany('Book');

}

}

hasMany的参数如下：

#### hasMany('关联模型名','关联外键','关联模型主键','别名定义')

Book模型类定义如下：

<?php

namespace app\index\model;

use think\Model;

class Book extends Model

{

protected $type = [

'publish\_time' => 'timestamp:Y-m-d',

];

// 开启自动写入时间戳

protected $autoWriteTimestamp = true;

// 定义自动完成的属性

protected $insert = ['status' => 1];

}

如果需要定义对应的关联，则可以使用belongsTo方法：

<?php

namespace app\index\model;

use think\Model;

class Book extends Model

{

protected $type = [

'publish\_time' => 'timestamp:Y-m-d',

];

// 开启自动写入时间戳

protected $autoWriteTimestamp = true;

// 定义自动完成的属性

protected $insert = ['status' => 1];

// 定义关联方法

public function user()

{

return $this->belongsTo('User');

}

}

### 关联新增

添加addBook方法用于新增关联数据：

public function addBook()

{

$user = UserModel::get(1);

$book = new Book;

$book->title = 'ThinkPHP5快速入门';

$book->publish\_time = '2016-05-06';

$user->books()->save($book);

return '添加Book成功';

}

对于一对多关联，也可以批量增加数据：

public function addBook()

{

$user = UserModel::get(1);

$books = [

['title' => 'ThinkPHP5快速入门', 'publish\_time' => '2016-05-06'],

['title' => 'ThinkPHP5开发手册', 'publish\_time' => '2016-03-06'],

];

$user->books()->saveAll($books);

return '添加Book成功';

}

### 关联查询

可以直接调用模型的属性获取全部关联数据，例如：

public function read()

{

$user = UserModel::get(1);

$books = $user->books;

dump($books);

}

一对多查询同样可以使用预载入查询，例如：

public function read()

{

$user = UserModel::get(1,'books');

$books = $user->books;

dump($books);

}

一对多预载入查询会在原先延迟查询的基础上增加一次查询，可以解决典型的N+1次查询问题。

如果要过滤查询，可以调用关联方法：

public function read()

{

$user = UserModel::get(1);

// 获取状态为1的关联数据

$books = $user->books()->where('status',1)->select();

dump($books);

// 获取作者写的某本书

$book = $user->books()->getByTitle('ThinkPHP5快速入门');

dump($book);

}

还可以根据关联数据来查询当前模型数据，例如：

public function read()

{

// 查询有写过书的作者列表

$user = UserModel::has('books')->select();

// 查询写过三本书以上的作者

$user = UserModel::has('books', '>=', 3)->select();

// 查询写过ThinkPHP5快速入门的作者

$user = UserModel::hasWhere('books', ['title' => 'ThinkPHP5快速入门'])->select();

}

### 关联更新

下面来进行关联数据的更新

public function update($id)

{

$user = UserModel::get($id);

$book = $user->books()->getByTitle('ThinkPHP5开发手册');

$book->title = 'ThinkPHP5快速入门';

$book->save();

}

或者使用查询构建器的update方法进行更新（但可能无法触发关联模型的事件）。

public function update($id)

{

$user = UserModel::get($id);

$user->books()->where('title', 'ThinkPHP5快速入门')->update(['title' => 'ThinkPHP5开发手册']);

}

### 关联删除

删除部分关联数据：

public function delete($id){

$user = UserModel::get($id);

// 删除部分关联数据

$book = $user->books()->getByTitle('ThinkPHP5开发手册');

$book->delete();

}

删除所有的关联数据：

public function delete($id){

$user = UserModel::get($id);

if($user->delete()){

// 删除所有的关联数据

$user->books()->delete();

}

}

## 多对多关联

一个用户会有多个角色，同时一个角色也会包含多个用户，这就是一个典型的多对多关联，先创建一个角色表think\_role结构如下：

DROP TABLE IF EXISTS `think\_role`;

CREATE TABLE IF NOT EXISTS `think\_role` (

`id` int(5) UNSIGNED NOT NULL AUTO\_INCREMENT,

`name` varchar(25) NOT NULL,

`title` varchar(50) NOT NULL,

PRIMARY KEY (`id`)

) ENGINE=MyISAM DEFAULT CHARSET=utf8;

多对多关联通常一定会有一个中间表，也称为枢纽表，所以需要创建一个用户角色的中间表，这里创建了一个think\_access表，结构如下：

DROP TABLE IF EXISTS `think\_access`;

CREATE TABLE IF NOT EXISTS `think\_access` (

`user\_id` int(6) UNSIGNED NOT NULL,

`role\_id` int(5) UNSIGNED NOT NULL

) ENGINE=MyISAM DEFAULT CHARSET=utf8;

### 关联定义

给User模型添加多对多关联方法定义

<?php

namespace app\index\model;

use think\Model;

class User extends Model

{

// 开启自动写入时间戳

protected $autoWriteTimestamp = true;

// 定义自动完成的属性

protected $insert = ['status' => 1];

// 定义一对一关联

public function profile()

{

return $this->hasOne('Profile');

}

// 定义一对多关联

public function books()

{

return $this->hasMany('Book');

}

// 定义多对多关联

public function roles()

{

// 用户 BELONGS\_TO\_MANY 角色

return $this->belongsToMany('Role', 'think\_access');

}

}

belongsToMany的参数如下：

#### belongsToMany('关联模型名','中间表名称','关联外键','关联模型主键','别名定义')

Role模型定义如下：

<?php

namespace app\index\model;

use think\Model;

class Role extends Model

{

public function user()

{

// 角色 BELONGS\_TO\_MANY 用户

return $this->belongsToMany('User', 'think\_access');

}

}

如果你使用的版本是5.0.8以上，中间表无需添加表前缀。

对于枢纽表并不需要创建模型类，在多对多关联关系中，并不需要直接操作枢纽表。

### 关联新增

给某个用户增加编辑角色，并且由于这个角色还没创建过，所以可以使用下面的方式：

// 关联新增数据

public function add()

{

$user = UserModel::getByNickname('张三');

// 新增用户角色 并自动写入枢纽表

$user->roles()->save(['name' => 'editor', 'title' => '编辑']);

return '用户角色新增成功';

}

也可以批量新增用户的角色如下：

// 关联新增数据

public function add()

{

$user = UserModel::getByNickname('张三');

// 给当前用户新增多个用户角色

$user->roles()->saveAll([

['name' => 'leader', 'title' => '领导'],

['name' => 'admin', 'title' => '管理员'],

]);

return '用户角色新增成功';

}

现在给另外一个用户增加编辑角色，由于该角色已经存在了，所以只需要使用attach方法增加枢纽表的关联数据：

// 关联新增数据

public function add()

{

$user = UserModel::getByNickname('张三');

$role = Role::getByName('editor');

// 添加枢纽表数据

$user->roles()->attach($role);

return '用户角色添加成功';

}

或者直接使用角色Id添加关联数据

// 关联新增数据

public function add()

{

$user = UserModel::getByNickname('张三');

$user->roles()->attach(1);

return '用户角色添加成功';

}

### 关联删除

如果需要解除用户的管理角色，可以使用detach方法删除关联的枢纽表数据，但不会删除关联模型数据，例如：

// 关联删除数据

public function delete()

{

$user = UserModel::get(2);

$role = Role::getByName('admin');

// 删除关联数据 但不删除关联模型数据

$user->roles()->detach($role);

return '用户角色删除成功';

}

如果有必要，也可以删除枢纽表的同时删除关联模型，下面的例子会解除用户的编辑角色并且同时删除编辑这个角色身份：

// 关联删除数据

public function delete()

{

$user = UserModel::getByNickname('张三');

$role = Role::getByName('editor');

// 删除关联数据 并同时删除关联模型数据

$user->roles()->detach($role,true);

return '用户角色删除成功';

}

### 关联查询

获取用户张三的所有角色的话，直接使用：

// 关联查询

public function read()

{

$user = UserModel::getByNickname('张三');

dump($user->roles);

}

同样支持对多对多关联使用预载入查询：

// 关联查询

public function read()

{

// 预载入查询

$user = UserModel::get(2,'roles');

dump($user->roles);

}

到目前为止，我们已经掌握了关联的基础用法，更多的关联使用请参考《[掌握5.0数据库和模型](http://www.kancloud.cn/thinkphp/master-database-and-model)》一书的第八章：模型关联的内容。

# （8）模型输出

模型的输出主要是指把查询和操作的模型数据输出到模板或者返回客户端，以User模型为例，模型定义如下：

<?php

namespace app\index\model;

use think\Model;

class User extends Model

{

}

可以输出模型实例对象为数组或者JSON。

* + [模板输出](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/147377#_15)
  + [输出数组](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/147377#_113)
    - [隐藏属性](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/147377#_142)
    - [指定属性](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/147377#_165)
    - [追加属性](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/147377#_186)
  + [输出JSON](https://www.kancloud.cn/thinkphp/thinkphp5_quickstart/147377#JSON_230)

## 模板输出

通常在控制器中读取模型数据并渲染模板输出，方法如下：

<?php

namespace app\index\controller;

use app\index\model\User as UserModel;

use think\Controller;

class User

{

public function read($id='')

{

$user = UserModel::get($id);

return view('read',['user'=>$user]);

}

}

在模板中可以直接输出模型对象的数据，例如创建application/index/view/user/read.html，内容如下：

用户ID：{$user.id}<br/>

昵称：{$user.nickname}<br/>

邮箱：{$user.email}<br/>

生日：{$user.birthday}<br/>

访问

http://tp5.com/user/1

页面输出结果为：

用户ID：1

昵称：流年

邮箱：thinkphp@qq.com

生日：1977/03/05

如果输出的是模型数据列表，

<?php

namespace app\index\controller;

use app\index\model\User as UserModel;

use think\Controller;

class User

{

public function index()

{

$list = UserModel::all();

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

}

}

模板修改为：

{volist name="list" id="user"}

用户ID：{$user.id}<br/>

昵称：{$user.nickname}<br/>

邮箱：{$user.email}<br/>

生日：{$user.birthday}<br/>

-------------------------------------<br/>

{/volist}

访问

http://tp5.com/user/index

页面输出结果为：

用户ID：1

昵称：流年

邮箱：thinkphp@qq.com

生日：1977/03/05

-------------------------------------

用户ID：2

昵称：看云

邮箱：kancloud@qq.com

生日：2015/04/02

-------------------------------------

用户ID：3

昵称：张三

邮箱：zhanghsan@qq.com

生日：1988/01/15

-------------------------------------

用户ID：4

昵称：李四

邮箱：lisi@qq.com

生日：1990/09/19

-------------------------------------

## 输出数组

可以使用toArray方法把当前的模型对象输出为数组。  
修改User控制器的read操作方法如下：

// 读取用户数据并输出数组

public function read($id = '')

{

$user = UserModel::get($id);

dump($user->toArray());

}

访问 http://tp5.com/user/20 页面输出结果为：

array (size=7)

'id' => int 20

'nickname' => string '张三' (length=6)

'email' => string 'zhanghsan@qq.com' (length=16)

'birthday' => string '1988/01/15' (length=10)

'status' => string '待审核' (length=9)

'create\_time' => string '2016-05-02 16:40:57' (length=19)

'update\_time' => string '2016-05-02 16:40:57' (length=19)

可以看到，User模型的数据已经经过了读取器方法处理。

### 隐藏属性

如果输出的时候需要隐藏某些属性，可以使用：

// 读取用户数据并输出数组

public function read($id = '')

{

$user = UserModel::get($id);

dump($user->hidden(['create\_time','update\_time'])->toArray());

}

再次访问 http://tp5.com/user/20 页面输出结果变成：

array (size=5)

'id' => int 20

'nickname' => string '张三' (length=6)

'email' => string 'zhanghsan@qq.com' (length=16)

'birthday' => string '1988/01/15' (length=10)

'status' => int 2

### 指定属性

或者指定一些属性输出，则可以用：

// 读取用户数据并输出数组

public function read($id = '')

{

$user = UserModel::get($id);

dump($user->visible(['id','nickname','email'])->toArray());

}

再次访问 http://tp5.com/user/20 页面输出结果变成：

array (size=3)

'id' => int 20

'nickname' => string '张三' (length=6)

'email' => string 'zhanghsan@qq.com' (length=16)

### 追加属性

如果读取器定义了一些非数据库字段的读取，例如：

<?php

namespace app\index\model;

use think\Model;

class User extends Model

{

// status修改器

protected function getUserStatusAttr($value,$data)

{

$status = [-1 => '删除', 0 => '禁用', 1 => '正常', 2 => '待审核'];

return $status[$data['status']];

}

}

而我们如果需要输出user\_status属性数据的话，可以使用append方法，用法如下：

// 读取用户数据并输出数组

public function read($id = '')

{

$user = UserModel::get($id);

dump($user->append(['user\_status'])->toArray());

}

再次访问 http://tp5.com/user/20 页面输出结果变成：

array (size=8)

'id' => int 20

'nickname' => string '张三' (length=6)

'email' => string 'zhanghsan@qq.com' (length=16)

'birthday' => string '1988/01/15' (length=10)

'status' => int 2

'create\_time' => string '2016-05-02 16:40:57' (length=19)

'update\_time' => string '2016-05-02 16:40:57' (length=19)

'user\_status' => string '待审核' (length=9)

## 输出JSON

对于API开发而言，经常需要返回JSON格式的数据，修改read操作方法改成JSON输出：

// 读取用户数据输出JSON

public function read($id = '')

{

$user = UserModel::get($id);

return $user->toJson();

}

访问 http://tp5.com/user/3 页面输出结果为：

{"id":3,"nickname":"张三","email":"zhanghsan@qq.com","birthday":"1988\/01\/15","status":2,"create\_time":"2016-05-02 16:40:57","update\_time":"2016-05-02 16:40:57"}

toJson输出方法仍然支持 hidden、visible和append方法。

或者采用更简单的方法输出JSON，下面的方式是等效的：

// 读取用户数据直接输出JSON

public function read($id = '')

{

return json(UserModel::get($id));

}