ThinkPHP 5.0 快速入门

目　录

零、序言

> **< ThinkPHP官方出品，入门 TP5 必读系列 >**

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（更新中）
- 附录A、常见问题集（更新中）
- 附录B、3.2和5.0区别 √

阅读须知

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。**

ThinkPHP V5.0 官方权威QQ群

> 新手群（369126686 开放制） 允许扯谈 [已满员]
> 高级群（50546480 收费制）禁止闲聊
> 专家群（416914496 邀请制）比较安静

快速入门用户专用交流群

> **已购买快速入门的用户请扫描加群和TP5开发者一起交流（群号：484416938）**

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和命名空间的概念，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。**

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，从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的代码的同时，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。

> `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。

官网下载

ThinkPHP最新的稳定版本可以在（官方网站下载页）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。

Composer安装和更新

ThinkPHP `5.0` 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 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。

> **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），看云上有 Composer 的中文版本。

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。

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 tp5  --prefer-dist
```

如果出现错误提示，请根据提示操作或者参考Composer中文文档。

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。

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，请耐心等待，或者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
> ```

Git下载和更新

`ThinkPHP` 使用 `Git` 版本库进行更新，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及国内的仓库地址（官方扩展只能通过Composer安装）：

[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://git.oschina.net/liu21st/thinkphp5.git`
- 核心框架：`https://git.oschina.net/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`

> **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）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。

目录结构

`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
```

> **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/exend	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
```

运行环境

> ThinkPHP5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

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（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。

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
```

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。

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。

调试模式

`ThinkPHP`支持调试模式，默认情况下是开启状态。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。

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。

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。

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其实目录为 `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：

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`，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!
```

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`

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，主要包含：

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://serverName/index.php/模块/控制器/操作
```

> **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（或者要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
> ```

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。
但下面的方式仍然可以得到正确的结果：

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

> **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。

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;
    }
}
```

> **提示：**
>
> 后面的示例访问地址，我们都假设配置了隐藏 `index.php` 。

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。

[0] HttpException in Route.php line 1176

非法请求:index/index/hello

```
1167.                 $controller = !empty($path) ? array_shift($path) : null;
1168.             }
1169.             // 解析操作
1170.             $action = !empty($path) ? array_shift($path) : null;
1171.             // 解析额外参数
1172.             self::parseUrlParams(empty($path) ? '' : implode('/', $path));
1173.             // 封装路由
1174.             $route = [$module, $controller, $action];
1175.             if (isset(self::$name[implode($depr, $route)])) {
1176.                 throw new HttpException(404, 'invalid request:' . $url);
1177.             }
```

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，

[0] HttpException in App.php line 290

模块不存在 :hello

```
281.                 $available = true;
282.             }
283.
284.             // 模块初始化
285.             if ($module && $available) {
286.                 // 初始化模块
287.                 $request->module($module);
288.                 $config = self::init($module);
289.             } else {
290.                 throw new HttpException(404, 'module not exists:' . $module);
291.             }
292.         } else {
293.             // 单一模块部署
294.             $module = '';
295.             $request->module($module);
296.         }
297.         // 当前模块路径
298.         App::$modulePath = APP_PATH . ($module ? $module . DS : '');
299.
```

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}', 'm
onth' => '\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}', 'mo
nth' => '\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',
];
```

生成URL地址

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');
```

> 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ThinkPHP`5.0`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。

总结

看完本篇内容后，你应该了解如下的知识：

- URL访问规范
- 路由定义
- URL地址生成

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，主要内容包括：

`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。

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，因为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：

```
<?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，在其它的控制器中就可以直接使用

```
<?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。

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!
```

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：

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-revali
date']);
    }
}
```

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!';
        }
    }

}
```

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：

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/
```

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。

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，分别用于查询和写入，下面我们来实现数据表`think_user`的CURD操作。

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);
```

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行数。

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：

```
$result = Db::connect('mysql://root:123456@127.0.0.1:3306/thinkphp#utf8')->query('selec
t * from think_data where id = 1');
dump($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');
```

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);
```

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')
    ->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。

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。

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

本章带你领略 `ThinkPHP5.0` 的查询语法，以及如何使用查询构建器进行查询操作，主要包括：

- 查询表达式
- 批量查询
- 快捷查询
- 视图查询
- 闭包查询
- 使用Query对象
- 获取数值
- 获取列数据
- 聚合查询
- 字符串查询
- 时间（日期）查询
- 分块查询

> 本章查询内容均配置了数据表前缀 `think_` ，因此统一使用 `Db` 类的 `name` 方法代替 `table` 方法进行举例说明。

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
```

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);
```

或者使用批量方式：

```
$result = Db::name('data')
    ->where([
        'id'   => [['in', [1, 2, 3]], ['between', '5,8'], 'or'],
        'name' => ['like', '%think%'],
    ])->limit(10)->select();
dump($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 `n
ame` LIKE '%think%' LIMIT 10
```

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
```

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.ema
il FROM think_user user INNER JOIN think_profile profile ON profile.user_id=user.id WHE
RE user.status = 1 order by user.id desc
```

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
```

使用Query对象

也可以事先封装Query对象，并传入select方法，例如：

```
$query = new \think\db\Query;
$query->name('city')->where('name', 'like', '%think%')
    ->where('id', 'in', '1,2,3')
    ->limit(10);
$result = Db::select($query);
dump($result);
```

使用 `Query` 对象的话，`select` 方法之前调用的任何的链式操作都是无效。

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=5)
  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=5)
  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=5)
  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
```

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）	

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 )
```

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);
```

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记录，记录映射到对象，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。

（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,
];
```

并添加路由定义（`application/route.php`）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',
];
```

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

```
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：

```
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：

```
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：

```
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,
    ];
}
```

（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

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，实例化模型类后执行`save`操作都是执行的数据库`insert`操作，如果你需要实例化执行`save`执行数据库的`update`操作，请确保在save方法之前调用`isUpdate`方法：

```
$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;
    $list = [
        ['nickname' => '张三', 'email' => 'zhanghsan@qq.com', 'birthday' => strtotime('1
988-01-15')],
        ['nickname' => '李四', 'email' => 'lisi@qq.com', 'birthday' => strtotime('1990-0
9-19')],
    ];
    if ($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
```

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查询构建器来完成，例如：

```
// 根据nickname读取用户数据
public function read()
{
    $user = UserModel::where('nickname', '流年')->find();
    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/>';
    }
}
```

我们也可以使用数据库的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()
{
    $list = UserModel::where('id','<',3)->select();
    foreach ($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：

```
流年
thinkphp@qq.com
1977/03/05
------------------------------------
看云
kancloud@qq.com
2015/04/02
------------------------------------
```

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';
    if (false !== $user->save()) {
        return '更新用户成功';
    } else {
        return $user->getError();
    }
}
```

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，查询模型数据后返回的模型示例执行`save`操作都是执行的数据库`update`操作，如果你需要实例化执行`save`执行数据库的`insert`操作，请确保在save方法之前调用`isUpdate`方法：

```
$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';
    $result           = UserModel::update($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。

定义完修改器后，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
    protected function scopeEmail($query)
    {
        $query->where('email', 'thinkphp@qq.com');
    }

    // status查询
    protected 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()
{
    $list = UserModel::scope('email,status')->all();
    foreach ($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
```

支持多次调用 `scope` 方法，并且可以追加新的查询及链式操作，例如：

```
// 根据查询范围获取用户数据列表
public function index()
{
    //
    $list = UserModel::scope('email')
        ->scope('status')
        ->scope(function ($query) {
            $query->order('id', 'desc');
        })
        ->all();
    foreach ($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
    protected 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')->all();
```

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
    protected 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')->all();
```

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
- 表单验证
- 错误提示
- 自定义验证规则
- 控制器验证

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
```

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';
    }
}
```

（6）输入和验证

（7）关联

本篇为您讲述ThinkPHP5.0的关联定义和基础用法，主要包括：

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，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 extends Controller
{
    // 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
```

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');
    }

}
```

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，更多的关联使用请关注后续相关的专题。

（8）模型输出

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`。

- 输出数组
- 隐藏属性
- 指定属性
- 追加属性
- 输出JSON

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)
    {
        $status = [-1 => '删除', 0 => '禁用', 1 => '正常', 2 => '待审核'];
        return $status[$value];
    }
```

```
}
```

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);
    echo $user->toJson();
}
```

访问`http://tp5.com/user/20`页面输出结果为：

```
{"id":22,"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"}
```

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($id = '')
{
    echo UserModel::get($id);
}
```

> toJson输出方法仍然支持`hidden`、`visible`和`append`方法。

（8）模型输出

七、视图和模板

快速入门（七）：视图和模板

本章主要来学习视图和模板的用法。

前面只是在控制器方法里面直接输出而没有使用视图模板功能，从现在开始来了解下如何把变量赋值到模板，并渲染输出，主要内容包括：

- 模板输出
- 分页输出
- 公共模板
- 模板定位
- 模板布局
- 标签定制
- 输出替换
- 渲染内容
- 助手函数

模板输出

首先来看如何输出一个数据集，我们修改`User`控制器的`index`方法如下：

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

use app\index\model\User as UserModel;
use think\Controller;

class User extends Controller
{
    // 获取用户数据列表并输出
    public function index()
    {
        $list = UserModel::all();
        $this->assign('list', $list);
        $this->assign('count', count($list));
        return $this->fetch();
    }
}
```

这里的 User 控制器和之前模型章节有所不同，继承了系统的 \think\Controller 类，该类对视图类的方法进行了封装，所以可以在无需实例化视图类的情况下，直接调用视图类的相关方法，包括：	方法	描述
assign	模板变量赋值	
fetch	渲染模板	

	文件
display	渲染内容
engine	初始化模板引擎

这里用到的其中两个方法`assign`和`fetch`，也是最常用的两个方法。

`assign`方法可以把任何类型的变量赋值给模板，关键在于模板中如何输出，不同的变量类型需要采用不同的标签输出。

前面我们已经学习过，`fetch`方法默认渲染输出的模板文件应该是当前控制器和操作对应的模板，也就是：

```
application/index/view/user/index.html
```

接下来，定义视图文件的内容，采用`volist`标签输出数据集：

```
<!doctype html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>查看用户列表</title>
<style>
body{
    color: #333;
    font: 16px Verdana, "Helvetica Neue", helvetica, Arial, 'Microsoft YaHei', sans-ser
if;
    margin: 0px;
    padding: 20px;
}

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>用户列表（{$count}）</h2>
{volist name="list" id="user" }
<div class="info">
ID：{$user.id}<br/>
昵称：{$user.nickname}<br/>
邮箱：{$user.email}<br/>
生日：{$user.birthday}<br/>
</div>
{/volist}
<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>
```

`ThinkPHP5.0`默认使用的是一个内置的编译型模板引擎，包含了一系列的模板标签，我们会陆续介绍一些常用的标签用法。

index方法给模板赋值了两个变量`count`和`list`，分别是标量和二维数组，标量的输出很简单，使用：`{$count}`便可，一看就明白。

二维数组通常使用`volist`标签输出。

```
{volist name="list" id="user"}
ID：{$user.id}<br/>
昵称：{$user.nickname}<br/>
邮箱：{$user.email}<br/>
生日：{$user.birthday}<br/>
------------------------<br/>
{/volist}
```

`volist`标签的`name`属性就是模板变量的名称，`id`属性则是定义每次循环输出的变量，在volist标签中间使用`{$user.id}`表示输出当前用户的id属性，以此类推下面的内容则依次输出用户的相关属性。

```
ID：{$user.id}<br/>
昵称：{$user.nickname}<br/>
邮箱：{$user.email}<br/>
生日：{$user.birthday}<br/>
```

来看下实际的输出效果，访问URL地址：

```
http://tp5.com/user/index
```

就可以看到页面输出结果如图：

当然，实际看到的数据可能有所出入。

分页输出

可以很简单的输出用户的分页数据，控制器`index`方法修改为：

```
// 获取用户数据列表
public function index()
{
    // 分页输出列表 每页显示3条数据
    $list = UserModel::paginate(3);
    $this->assign('list',$list);
    return $this->fetch();
}
```

模板文件修改为：

```
<link rel="stylesheet" href="/static/bootstrap/css/bootstrap.min.css" />
<h2>用户列表（{$list->total()}）</h2>
{volist name="list" id="user"}
ID：{$user.id}<br/>
昵称：{$user.nickname}<br/>
邮箱：{$user.email}<br/>
生日：{$user.birthday}<br/>
------------------------<br/>
{/volist}
```

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

> 注意，由于演示需要，在模板中引入了bootstrap样式文件。请按照所示位置存放。

多插入一些数据后，访问下面的地址：

```
http://tp5.com/user/index
```

后可以看到输出如图所示：

公共模板

加上之前定义的创建用户的模板，现在已经有两个模板文件了，为了避免重复定义模板，可以把模板的公共头部和尾部分离出来，便于维护。

我们把模板文件拆分为三部分：

```
application/index/view/user/header.html
```

```
application/index/view/user/index.html
application/index/view/user/footer.html
```

`header.html`内容为：

```
<!doctype html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>查看用户列表</title>
<style>
body{
    color: #333;
    font: 16px Verdana, "Helvetica Neue", helvetica, Arial, 'Microsoft YaHei', sans-ser
if;
    margin: 0px;
    padding: 20px;
}

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>
```

`footer.html`内容为：

```
<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>
```

`index.html` 内容为：

```
{include file="user/header" /}
<h2>用户列表（{$count}）</h2>
{volist name="list" id="user" }
<div class="info">
ID：{$user.id}<br/>
昵称：{$user.nickname}<br/>
邮箱：{$user.email}<br/>
生日：{$user.birthday}<br/>
</div>
{/volist}
{include file="user/footer" /}
```

公共头部模板文件中可能存在一些变量，例如这里的页面标题不同的页面会有不同，可以使用

```
{include file="user/header" title="查看用户列表" /}
```

然后把头部模板文件中的

```
<title>查看用户列表</title>
```

改为：

```
<title>[title]</title>
```

再次访问页面可以看到输出结果：

如果需要传递多个变量，则使用多个属性即可，例如：

```
{include file="user/header" title="查看用户列表" keywords="thinkphp" /}
```

也可以支持传递动态变量的方式，例如：

```
{include file="user/header" title="$title" /}
```

模板定位

`fetch`方法的第一个参数表示渲染的模板文件或者模板表达式。

通常我们都是使用的模板表达式，而不需要使用完整的文件名。

模板文件名可以随意命名，如果把`index.html`文件改成：

```
application/index/view/user/list.html
```

index操作方法中的fetch方法需要改成：

```
return $this->fetch('list');
```

而如果fetch方法改成：

```
return $this->fetch('index/list');
```

那么实际渲染的模板文件则是

```
application/index/view/index/list.html
```

当然，你可以设置更多的目录级别，例如：

```
return $this->fetch('one/two/three/list');
```

那么实际渲染的模板文件则是

```
application/index/view/one/two/three/list.html
```

有一些和模板定位相关的设置参数能够帮助你调整模板文件的位置和名称。
通常来说，模板相关的参数可以直接在配置文件中配置 `template` 参数，默认的配置如下：

```
'template'              => [
    // 模板引擎类型 支持 php think 支持扩展
    'type'          => 'Think',
    // 模板路径
    'view_path'     => '',
    // 模板后缀
    'view_suffix'   => '.html',
    // 模板文件名分隔符
    'view_depr'     => DS,
    // 模板引擎普通标签开始标记
    'tpl_begin'     => '{',
    // 模板引擎普通标签结束标记
    'tpl_end'       => '}',
    // 标签库标签开始标记
    'taglib_begin' => '{',
    // 标签库标签结束标记
    'taglib_end'    => '}',
],
```

`view_path` 参数决定了你的模板文件的根目录，如果没有设置的话系统会默认使用当前模块的视图目录 `view` 。

如果希望自定义模板文件的位置、命名和后缀，可以对模板参数稍加修改如下：

```
'template'              => [
    // 模板引擎类型 支持 php think 支持扩展
    'type'          => 'Think',
    // 模板路径
    'view_path'     => '../template/index/',
    // 模板后缀
    'view_suffix'   => '.tpl',
    // 模板文件名分隔符
    'view_depr'     => '_',
],
```

通过配置我们把当前渲染的模板文件移动到了

```
ROOT_PATH/template/index/user_index.tpl
```

模板布局

现在使用模板布局来进一步简化模板定义。

首先需要定义一个布局模板文件，放到 `application/index/view/layout.html` 内容如下：

```
{include file="user/header" /}
 {__CONTENT__}
{include file="user/footer" /}
```

`application/index/view/user/index.html` 改成：

```
{layout name="layout" /}
<h2>用户列表（{$count}）</h2>
{volist name="list" id="user" }
<div class="info">
ID：{$user.id}<br/>
昵称：{$user.nickname}<br/>
邮箱：{$user.email}<br/>
生日：{$user.birthday}<br/>
</div>
{/volist}
```

在index模板文件中开头定义 `layout` 标签 ，表示当前模板使用了布局，布局模板文件为 `layout.html` ，布局模板中的 `{__CONTENT__}` 会自动替换为解析后的index.html内容。

如果你的布局模板中不是使用 `{__CONTENT__}` 的话，可以改成：

```
{layout name="layout/newlayout" replace="[__REPLACE__]" /}
```

那么回自动读取模板文件

`application/index/view/layout/newlayout.html` ，内容如下：

```
{include file="user/header" /}
[__REPLACE__]
{include file="user/footer" /}
```

如果你所有的模板文件都统一使用布局，并且都是有同一个布局模板，那么可以统一配置而不需要在模板文件中使用`layout`标签定义。

在应用配置或者模块配置中添加如下设置参数：

```
'template'  =>  [
    'layout_on'     =>  true,
    'layout_name'   =>  'layout',
    'layout_item'   =>  '[__REPLACE__]'
]
```

模板文件中只需要定义如下：

```
<h2>用户列表（{$count}）</h2>
{volist name="list" id="user" mod="2" }
<div class="info">
ID：{$user.id}<br/>
昵称：{$user.nickname}<br/>
邮箱：{$user.email}<br/>
生日：{$user.birthday}<br/>
</div>
{/volist}
```

访问

```
http://tp5.com/user/index
```

可以看到页面输出结果和之前一样：

如果想动态控制模板文件使用布局，则可以在控制器中使用：

```
// 获取用户数据列表
public function index()
{
    $list = UserModel::all();
    $this->assign('list', $list);
    $this->assign('count', count($list));
    // 动态使用布局
    $this->view->engine->layout('layout','[__REPLACE__]');
    return $this->fetch();
}
```

> 注意：这里调用的是 `this->view->engine` 对象的 `layout` 方法，并不是所有的模板引擎都支持布局功能，如果你使用的是其它的模板引擎，可能不提供 `layout` 方法。

如果使用配置方式开启了布局模板，也可以使用该方法临时关闭布局，例如：

```
// 获取用户数据列表
public function index()
{
    $list = UserModel::all();
    $this->assign('list', $list);
    $this->assign('count', count($list));
    // 临时关闭布局
    $this->view->engine->layout(false);
    return $this->fetch();
}
```

或者直接在模板文件的开头加上 `{__NOLAYOUT__}` 标签：

```
{__NOLAYOUT__}
<h2>用户列表（{$count}）</h2>
{volist name="list" id="user" mod="2" }
<div class="info">
ID：{$user.id}<br/>
昵称：{$user.nickname}<br/>
邮箱：{$user.email}<br/>
生日：{$user.birthday}<br/>
</div>
{/volist}
```

标签定制

可以设置模板标签的定界符：

```
'template'              => [
    // 模板引擎类型 支持 php think 支持扩展
    'type'         => 'Think',
    // 模板路径
    'view_path'    => '../template/index/',
    // 模板后缀
    'view_suffix'  => '.tpl',
    // 模板文件名分隔符
    'view_depr'    => '_',
    // 模板引擎普通标签开始标记
    'tpl_begin'    => '{',
    // 模板引擎普通标签结束标记
    'tpl_end'      => '}',
    // 标签库标签开始标记
    'taglib_begin' => '<',
    // 标签库标签结束标记
    'taglib_end'   => '>',
],
```

并且修改 `index.html` 模板中的标签修改如下：

```
<h2>用户列表（{$count}）</h2>
<div class="info">
<volist name="list" id="user" >
ID：{$user.id}<br/>
```

```
昵称：{$user.nickname}<br/>
邮箱：{$user.email}<br/>
生日：{$user.birthday}<br/>
</volist>
</div>
```

输出替换

为了更加清晰，需要把资源文件独立出来，并在模板文件中引入，例如增加`public/static/common.css`文件：

```
body{
    color: #333;
    font: 16px Verdana, "Helvetica Neue", helvetica, Arial, 'Microsoft YaHei', sans-ser
if;
    margin: 0px;
    padding: 20px;
}

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;
}
```

我们在header.html文件中引入资源文件

```
<html>
<head>
<meta charset="UTF-8">
<title>[title]</title>
<link charset="utf-8" rel="stylesheet" href="/static/common.css">
</head>
<body>
```

但这样有一个问题，如果部署的目录变化的话，资源文件的路径就会跟着变化，这里我们采用输出替换功能，使得资源文件的引入动态化
可以在输出之前对解析后的内容进行替换，使用：

```
// 读取用户数据
public function read($id='')
{
    $user = UserModel::get($id);
    $this->assign('user',$user);
    $this->view->replace([
        '__PUBLIC__'    =>  '/static',
    ]);
    return $this->fetch();
}
```

模板文件改为：

```
<html>
<head>
<meta charset="UTF-8">
<title>[title]</title>
<link charset="utf-8" rel="stylesheet" href="__PUBLIC__/common.css">
</head>
<body>
```

最终输出的时候，会自动进行 `__PUBLIC__` 替换。

渲染内容

有时候，并不需要模板文件，而是直接渲染内容或者读取数据库中存储的内容，控制器方法修改如下：

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

use app\index\model\User as UserModel;
use think\Controller;

class User extends Controller
{
    // 获取用户数据列表并输出
    public function index()
    {
        $list = UserModel::all();
        $this->assign('list', $list);
        $this->assign('count', count($list));
        // 关闭布局
        $this->view->engine->layout(false);
        $content = <<<EOT
<style>
body{
    color: #333;
    font: 16px Verdana, "Helvetica Neue", helvetica, Arial, 'Microsoft YaHei', sans-ser
if;
    margin: 0px;
    padding: 20px;
}
```

```
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>
<h2>用户列表（{\$count}）</h2>
<div>
{volist name="list" id="user"  }
ID：{\$user.id}<br/>
昵称：{\$user.nickname}<br/>
邮箱：{\$user.email}<br/>
生日：{\$user.birthday}<br/>
------------------------<br/>
{/volist}
</div>
<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>

EOT;
        return $this->display($content);
    }
}
```

`display` 方法用于渲染内容而不是模板文件输出，和直接使用 `echo` 输出的区别是 `display` 方法输出的内容支持模板标签的解析。

助手函数

可以使用系统提供的助手函数view简化模板渲染输出（注意不适用于内容渲染输出）：

前面的模板渲染代码可以改为：

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

use app\index\model\User as UserModel;

class User
{
    // 读取用户数据
    public function read($id='')
    {
        $user = UserModel::get($id);
        return view('', ['user' => $user], ['__PUBLIC__'    => '/static']);
    }
}
```

> 使用`view`助手函数，不需要继承`think\Controller`类。该方法的第一个参数就是渲染的模板表达式。

八、调试和日志

快速入门（八）：调试和日志

项目开发的时候，出现错误在所难免，最大的困惑在于发现问题所在，其次才是如何解决问题。因此懂得如何调试和跟踪问题非常之关键，`5.0`版本提供了非常方便的调试工具和手段，让你更容易定位和发现问题。

本篇授你`TP5`调试大法之独门五式，稍加修炼，则可让你的TP5调试功力独步天下。

- 第一式：未雨绸缪——页面Trace
- 第二式：初见端倪——异常页面
- 第三式：拨云见日——断点调试
- 第四式：欲穷千里——日志分析
- 第五式：运筹帷幄——远程调试

第一式：未雨绸缪——页面Trace

> 第一式是基本式，简单易学，但往往容易被忽视。
> 难度系数：0
> 实用指数：6

页面`Trace`是`ThinkPHP`经典的调试手段，`ThinkPHP5.0`继续发扬光大，已经可以支持不依赖页面显示。

页面`Trace`的主要作用包括：

- 查看运行数据；
- 查看文件加载情况；
- 查看运行流程；
- 查看当前执行SQL；
- 跟踪调试数据；
- 查看页面错误信息；

系统默认不开启页面`Trace`，开启页面`Trace`是在应用配置文件中设置下面的参数：

```
// 开启应用Trace调试
'app_trace' =>  true,
// 设置Trace显示方式
'trace'     =>  [
    // 在当前Html页面显示Trace信息
    'type'  =>  'html',
],
```

我们用默认自带的欢迎页面来测试，访问后就可以看到页面`Trace`信息按钮。

ThinkPHP V5

十年磨一剑 - 为API开发设计的高性能框架

[V5.0 版本由 七牛云 独家赞助发布]

新手快速入门 1元试用Azure云 完全开发手册

注意看在右下角可以看到一个 `ThinkPHP` 的 `LOGO` 和当前页面的运行时间， 

点击会弹出详细的 `Trace` 信息，如图：

这是系统默认的 `Trace` 信息显示效果，包含了基本、文件、流程、错误、 `SQL` 和调试信息。

基本信息一栏显示了当前请求的运行信息，包括运行时间、吞吐率、内存开销和文件加载等基本信息，通过这个页面可以对当前的请求有一个直观的了解，例如当前请求的内存开销是否过大，查询次数是否在合理的范围之内等等。

文件信息一栏则按加载顺序显示了当前请求加载的文件列表。
流程信息一栏则会显示当前请求做了哪些操作，大家可以在开发的过程中经常关注下不同页面的请求和区别，该栏的内容开启调试模式后可见。

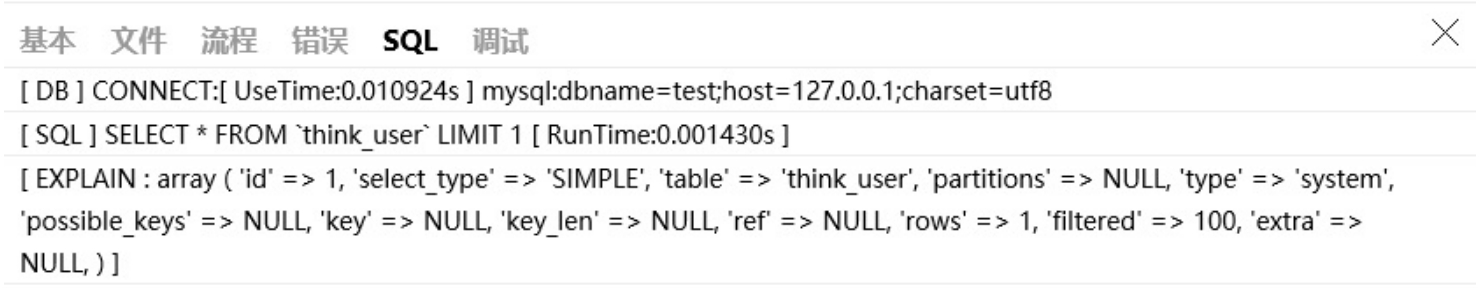
错误信息一栏会显示页面执行过程中的相关错误，包括警告错误（由于ThinkPHP5.0默认情况下对错误零容忍，所以你在正常情况下基本看不到任何错误，因为有任何错误都会直接抛出异常）。

如果你当前请求使用了数据库操作和查询的话，并且开启了数据库调试模式（注意，数据库调试模式可以单独开启，在数据库配置文件中配置开启`debug`参数）会在`SQL`一栏显示相关的`SQL`连接和查询信息，如图：

对于一些性能不高的查询尤其要引起注意，及早进行优化，如果要查看每个`SQL`查询的`EXPLAIN`信息，可以在数据库配置文件中设置`sql_explain`参数如下：

```
// 是否需要进行SQL性能分析
'sql_explain'    => true,
```

开启后，我们就可以看到SQL语句的下面增加了`EXPLAIN`分析信息：



最后一栏是用于开发过程的调试输出，使用`trace`方法调试输出的信息不会在页面直接显示，而是在页面`Trace`的调试一栏显示输出。

我们在控制器的方法中增加

```
trace('这是测试调试信息');
trace([1,2,3]);
```

然后刷新页面会看到：

基本 文件 流程 错误 SQL **调试**

这是测试调试信息

Array ([0] => 1 [1] => 2 [2] => 3)

如果你不希望影响页面的输出效果，可以启用浏览器Trace调试信息，设置如下：

```
// 开启应用Trace调试
'app_trace' =>  true,
// 设置Trace显示方式
'trace' =>  [
    // 使用浏览器console显示页面trace信息
    'type'  =>  'console',
],
```

设置后，用`Chrome`浏览器打开控制台切换到`console`可以看到如下显示：

控制台方式依然可以查看基本、文件、流程、错误、SQL和调试栏的相关信息。

并且支持颜色显示，例如：

现在你已经基本了解了页面 `Trace` 的作用了，在开发过程中，开启页面 `Trace` 显示可以让你及早了解系统运行状况和及时排查隐患，起到未雨绸缪的作用。

> Ajax方式请求的信息不会在页面Trace中显示，还包括部分页面Trace之后执行的日志信息也无法在Trace中查看到。

第二式：初见端倪——异常页面

> 第二式是临危式，一旦遇"敌"方出此招，必须有波澜不惊的气势方能驾驭自如。
> 难度系数：3
> 实用指数：8

前面已经提到，`ThinkPHP5.0` 对错误非常严谨，默认情况下，任何的错误（包括警告错误）系统都会抛出异常。

为了培养实战经验，让我们来和异常进行一次亲密接触吧，下面的代码用了一个未定义索引 `$_GET['name']` ：

```
namespace app\index\controller;

class Index
{
    public function index()
    {
        return 'hello,'.$_GET['name'];
    }
}
```

访问 `http://tp5.com` 后显示：

[8] ErrorException in Index.php line 10

未定义索引：name

```
1.  <?php
2.  namespace app\index\controller;
3.
4.  use app\index\model\User;
5.
6.  class Index
7.  {
8.      public function index()
9.      {
10.         return 'hello,'.$_GET['name'];
11.     }
12. }
```

Call Stack

```
1.  in Index.php line 10
2.  at Error::appError(8, 'Undefined index: nam...', 'E:\www\tp\applicatio...', 10, [])
    in Index.php line 10
3.  at Index->index()
4.  at ReflectionMethod->invokeArgs(object(Index), []) in App.php line 219
5.  at App::invokeMethod([object(Index), 'index']) in App.php line 338
6.  at App::module(['', null, null], ['app_namespace' => 'app', 'app_debug' => true,
```

如果你的系统是英文的话，会自动显示英文提示：

[8] ErrorException in Index.php line 10

Undefined index: name

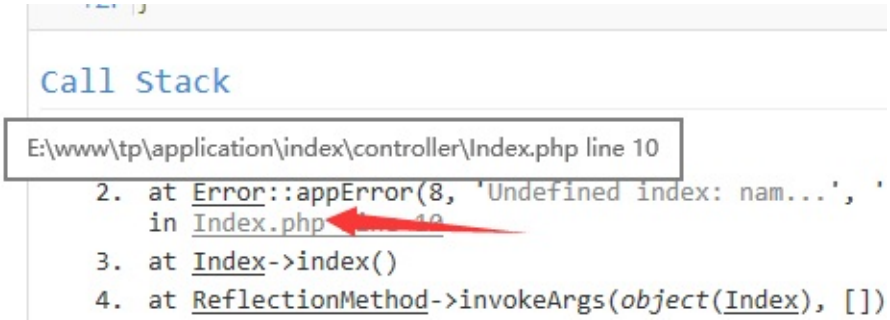
```php
<?php
namespace app\index\controller;

use app\index\model\User;

class Index
{
    public function index()
    {
        return 'hello,'.$_GET['name'];
    }
}
```

Call Stack

1. in Index.php line 10
2. at Error::appError(8, 'Undefined index: nam...', 'E:\www\tp\applicatio...', 10, [])
   in Index.php line 10
3. at Index->index()
4. at ReflectionMethod->invokeArgs(*object*(Index), []) in App.php line 219
5. at App::invokeMethod([*object*(Index), 'index']) in App.php line 338
6. at App::module(['', *null*, *null*], ['app_namespace' => 'app', 'app_debug' => *true*,

异常页面除了显示错误信息之外，还有很多的额外的信息能够帮助你快速定位和发现问题所在。

第一步是查看异常所在的文件源码，错误行会用红色标识，如果要继续一探究竟，那么就需要查看下面的`Call Stack`信息，这个是显示了当前异常的详细Trace信息，带下划线样式的地方，鼠标移上去会停留会显示该文件的详细位置



这些信息还不够，不要急，仔细查看完整的异常页面，其实还包含了当前请求的全局变量和常量，如图：

[8] ErrorException in Index.php line 10

未定义索引：name

```
1.  <?php
2.  namespace app\index\controller;
3.
4.  use app\index\model\User;
5.
6.  class Index
7.  {
8.      public function index()
9.      {
10.         return 'hello,'.$_GET['name'];
11.     }
12. }
```

Call Stack

1. in Index.php line 10
2. at Error::appError(8, 'Undefined index: nam...', 'E:\www\tp\applicatio...', 10, []) in Index.php line 10
3. at Index->index()
4. at ReflectionMethod->invokeArgs(*object*(Index), []) in App.php line 219
5. at App::invokeMethod([*object*(Index), 'index']) in App.php line 338
6. at App::module(['', *null*, *null*], ['app_namespace' => 'app', 'app_debug' => *true*, 'app_trace' => *true*, ...], *false*) in App.php line 117
7. at App::run() in start.php line 18
8. at require('E:\www\tp\thinkphp\s...') in index.php line 16

Environment Variables

GET Data empty

POST Data empty

Files empty

Cookies

pgv_pvi	5043134464
PHPSESSID	ct9bv5nf4l0dcmts8qt0gvldv7
pgv_si	s7539599360
thinkphp_show_page_trace	1|5
think_var	zh-hans-cn

Session empty

Server/Request Data

HTTP_ACCEPT	text/html, application/xhtml+xml, image/jxr, */*
HTTP_ACCEPT_LANGUAGE	zh-Hans-CN,zh-Hans;q=0.5
HTTP_USER_AGENT	Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko LBBROWSER
HTTP_ACCEPT_ENCODING	gzip, deflate
HTTP_HOST	www.tp5.com
HTTP_CONNECTION	Keep-Alive
HTTP_COOKIE	pgv_pvi=5043134464; PHPSESSID=ct9bv5nf4l0dcmts8qt0gvldv7; pgv_si=s7539599360; thinkphp_show_page_trace=1|5; think_var=zh-hans-cn
PATH	C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS\System32\WindowsPowerShell\v1.0\;C:\wamp64\bin\php\php7.0.0;C:\ProgramData\ComposerSetup\bin;C:\Program Files\TortoiseGit\bin;C:\Program Files\Git\bin;C:\WINDOWS\system32\config\systemprofile\AppData\Local\Microsoft\WindowsApps
SystemRoot	C:\WINDOWS
COMSPEC	C:\WINDOWS\system32\cmd.exe
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
WINDIR	C:\WINDOWS
SERVER_SIGNATURE	<address>Apache/2.4.17 (Win64) PHP/7.0.0 Server at www.tp5.com Port 80</address>
SERVER_SOFTWARE	Apache/2.4.17 (Win64) PHP/7.0.0
SERVER_NAME	www.tp5.com
SERVER_ADDR	127.0.0.1
SERVER_PORT	80
REMOTE_ADDR	127.0.0.1
DOCUMENT_ROOT	E:/www/tp/public
REQUEST_SCHEME	http
CONTEXT_PREFIX	
CONTEXT_DOCUMENT_ROOT	E:/www/tp/public
SERVER_ADMIN	webmaster@dummy-host.example.com
SCRIPT_FILENAME	E:/www/tp/public/index.php
REMOTE_PORT	54427
GATEWAY_INTERFACE	CGI/1.1
SERVER_PROTOCOL	HTTP/1.1
REQUEST_METHOD	GET
QUERY_STRING	
REQUEST_URI	/
SCRIPT_NAME	/index.php
PHP_SELF	/index.php
REQUEST_TIME_FLOAT	1470977203.067
REQUEST_TIME	1470977203

Environment Variables empty

ThinkPHP Constants

APP_PATH	E:\www\tp\public/../application/
THINK_VERSION	5.0.0 RC4
THINK_START_TIME	1470977203.0721
THINK_START_MEM	360672
EXT	.php
DS	\
THINK_PATH	E:\www\tp\thinkphp\
LIB_PATH	E:\www\tp\thinkphp\library\
CORE_PATH	E:\www\tp\thinkphp\library\think\
TRAIT_PATH	E:\www\tp\thinkphp\library\traits\
ROOT_PATH	E:\www\tp\public/..\
EXTEND_PATH	E:\www\tp\public/..\extend\
VENDOR_PATH	E:\www\tp\public/..\vendor\
RUNTIME_PATH	E:\www\tp\public/..\runtime\
LOG_PATH	E:\www\tp\public/..\runtime\log\
CACHE_PATH	E:\www\tp\public/..\runtime\cache\
TEMP_PATH	E:\www\tp\public/..\runtime\temp\
CONF_PATH	E:\www\tp\public/../application/
CONF_EXT	.php
ENV_PREFIX	PHP_
IS_CLI	false
IS_WIN	true

ThinkPHP V5.0.0 RC4 { 十年磨一剑-为API开发设计的高性能框架 }

遇到异常页面后要沉着冷静的分析错误信息，根据经验，第二式可以解决80%的问题定位，调试大神一般会在这个阶段把问题找出。

如若仍然无从下手的，请继续修炼第三式。

第三式：拨云见日——断点调试

> 第三式为破敌式，不断设置断点缩小和定位错误范围，直到找到错误位置。虽然简单粗暴，但对于复杂的情况下排查错误非常实用，并且如何设置断点有很多的经验。
> 难度系数：6
> 实用指数：9

系统为断点调试提供了几个有用的方法，诀窍就是通过调试信息不断缩小问题的范围以及诊断变量的变化。

`dump` 变量调试输出

在你需要的断点位置，调用`dump`方法，可以输出浏览器友好的变量信息，支持任何变量类型，例如：

```
dump('测试');
dump(['a', 'b', 'c']);
dump(User::get());
```

页面会输出类似下面的信息：

```
E:\www\tp\thinkphp\library\think\Debug.php:168:string '测试' (length=6)

E:\www\tp\thinkphp\library\think\Debug.php:168:
array (size=3)
  0 => string 'a' (length=1)
  1 => string 'b' (length=1)
  2 => string 'c' (length=1)

E:\www\tp\thinkphp\library\think\Debug.php:168:
object(app\index\model\User)[10]
  protected 'table' => string 'think_user' (length=10)
  protected 'pk' => string 'id' (length=2)
  protected 'field' =>
    array (size=7)
      'id' => string 'int' (length=3)
      'birthday' => string 'int' (length=3)
      'status' => string 'int' (length=3)
      'create_time' => string 'int' (length=3)
      'update_time' => string 'int' (length=3)
      0 => string 'nickname' (length=8)
      1 => string 'email' (length=5)
  protected 'connection' =>
    array (size=0)
      empty
  protected 'name' => string 'User' (length=4)
  protected 'class' => string 'app\index\model\User' (length=20)
  protected 'error' => null
  protected 'validate' => null
  protected 'visible' =>
    array (size=0)
      empty
  protected 'hidden' =>
    array (size=0)
      empty
  protected 'append' =>
    array (size=0)
      empty
  protected 'data' =>
    array (size=9)
      'id' => int 2
      'nickname' => string 'thinkphp' (length=8)
```

`halt` 变量调试并中断输出

`halt` 方法的作用和 `dump` 一样，只是在输出变量之后会中断当前程序的执行，下面是示例代码：

```
dump('测试');
halt(['a', 'b', 'c']);
halt('这里的信息是不会输出的');
```

页面会输出：

```
E:\www\tp\thinkphp\library\think\Debug.php:168:string '测试' (length=6)
E:\www\tp\thinkphp\library\think\Debug.php:168:
array (size=3)
  0 => string 'a' (length=1)
  1 => string 'b' (length=1)
  2 => string 'c' (length=1)
```

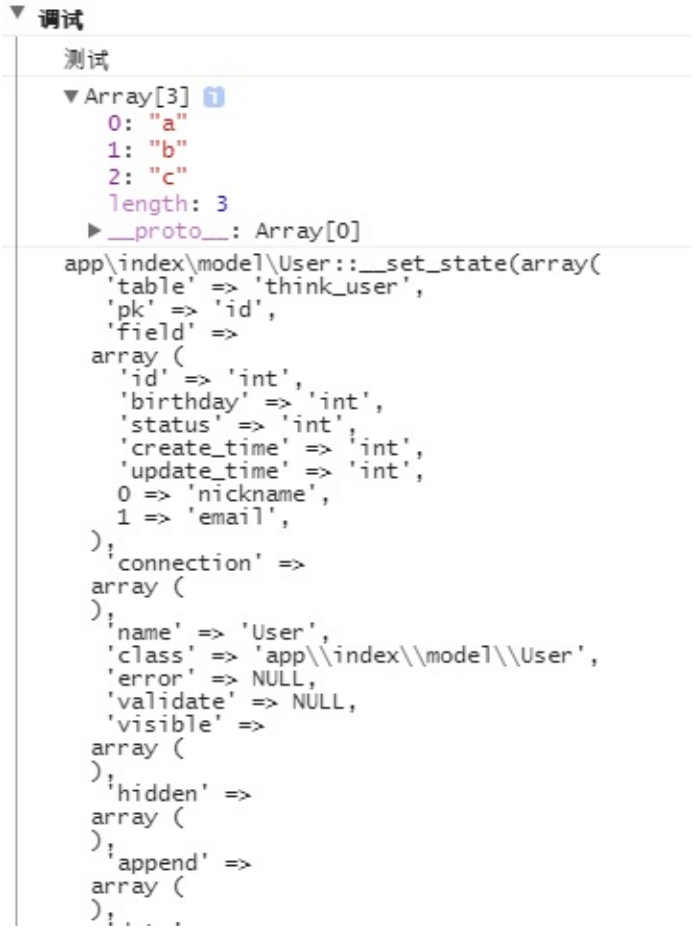
第二个 `halt` 方法不会被执行，因为 `halt` 方法在执行后就中止程序了。

`trace` 控制台输出

如果你不希望在页面输出调试信息，可以使用 `trace` 方法，该方法输出的信息会在页面 `Trace` 或者浏览器 `Console` 中显示，使用方法和 `dump` 是一样的。

```
trace('测试');
trace(['a', 'b', 'c']);
trace(User::get());
```

输出效果如图：



利用好这三个调试方法，足以临场杀敌，百万"大军"中取 `BUG` 首级！

第四式：欲穷千里——日志分析

> 第四式为诊断式，意为经常追溯日志信息，协同分析错误原因。
> 难度系数：6
> 实用指数：7

系统记录的日志信息在很多时候能够发挥意想不到的作用，默认情况下，系统使用文件方式记录日志，并且按照日期自动分开子目录保存，文件结构如下：

```
runtime/log
```

```
    └─201608
        05.log
        06.log
        08.log
        09.log
        10.log
        11.log
        12.log
```

每天会生成一个当天的日志文件，日志文件的内容和页面 `Trace` 信息的内容类似，但区别在于所有的请求都会记录，因此日志信息包含了大量的信息，更加有利于我们分析问题。

ThinkPHP对系统的日志按照级别来分类，并且这个日志级别完全可以自己定义，系统内部使用的级别包括：

- **log** 常规日志，用于记录日志
- **error** 错误，一般会导致程序的终止
- **notice** 警告，程序可以运行但是还不够完美的错误
- **info** 信息，程序输出信息
- **debug** 调试，用于调试信息
- **sql** SQL语句，用于SQL记录，只在数据库的调试模式开启时有效

系统提供了不同日志级别的快速记录方法，例如：

```
Log::error('错误信息');
Log::info('日志信息');
```

或者使用 `trace` 方法记录日志

```
trace('错误信息','error');
trace('日志信息','info');
```

为了便于分析，还支持设置某些级别的日志信息单独文件记录，例如：

```
'log'   => [
    'type'          => 'file',
    // error和sql日志单独记录
    'apart_level'   =>  ['error','sql'],
],
```

设置后，就会单独生成 `error` 和 `sql` 两个类型的日志文件，主日志文件中将不再包含这两个级别的日志信息，日志文件结构类似于：

```
runtime/log
    └─201608
        05.log
        06.log
        08.log
        09.log
```

```
        10.log
        11.log
        12.log
        12_error.log
        12_sql.log
```

定期查看系统的日志文件便于及时发现一些可能存在的隐患，以及给已有的问题提供更多的参考依据。

日志文件可以使用`Socket`方式记录到远程服务器，这就是后面要讲的远程调试功能，当然，你还可以扩展自己的日志记录方式来满足更多的要求。

第五式：运筹帷幄——远程调试

> 第五式为独步式，一旦修炼完成，千里之外任何错误尽在掌握。
> 难度系数：6
> 实用指数：8

远程调试功能暂且不表，我们将会在下一章`API`开发中为你详细讲述。

勤加修炼并掌握好这`ThinkPHP5.0`的五式调试大法，任何错误都会被你秒破！

九、API开发

快速入门（九）：API开发

使用 `ThinkPHP5.0` 可以更简单的进行 `API` 开发，并且最大程度的满足 `API` 对性能的要求，下面就 `API` 开发中的几个主要问题描述下。

`5.0` 对 `API` 开发的支持包括架构、功能和性能方面的良好支持。

API版本

我们以一个用户信息读取的接口为例，包含两个版本 `V1` 和 `V2` ， `v2` 版本的接口包括用户的档案信息，统一使用 `json` 格式数据输出到客户端。

在 `application` 目录下面创建 `api` 模块目录，并创建 `controller` 和 `model` 子目录，因为 `api` 接口无需视图，所以不需要创建 `view` 目录。

`api` 版本号的传入方式有很多，包括设置头信息、请求参数传入以及路由方式，这里我们采请求参数传入的方式，设置路由如下：

```
Route::rule(':version/user/:id','api/:version.User/read');
```

不同版本的URL访问地址为：

```
http://tp5.com/v1/user/10
http://tp5.com/v2/user/10
```

> 版本号中不能包含 `.` 符号。

`v1` 版本控制器（类文件位置为 `application/api/controller/v1/User.php` ）代码如下：

```
namespace app\api\controller\v1;

use app\api\model\User as UserModel;

class User
{
    // 获取用户信息
    public function read($id = 0)
    {
        $user = UserModel::get($id);
        if ($user) {
            return json($user);
        } else {
            return json(['error' => '用户不存在'], 404);
        }
    }

}
```

`v2` 版本的控制器（类文件位置为 `application/api/controller/v2/User.php` ）代码如下：

```
namespace app\api\controller\v2;

use app\api\model\User as UserModel;

class User
{
    // 获取用户信息
    public function read($id = 0)
    {
        $user = UserModel::get($id, 'profile');
        if ($user) {
            return json($user);
        } else {
            return json(['error' => '用户不存在'], 404);
        }
    }

}
```

v2版本和v1版本的接口区别在于v2的接口用户信息包含了用户的关联档案信息。

> 除了使用 `json` 格式返回客户端之外，系统还支持 `xml` 、 `jsonp` 格式，只需要把上面的 `json` 函数更改为 `xml` 和 `jsonp` 即可。

`User` 模型代码如下：

```
namespace app\api\model;

use think\Model;

class User extends Model
```

```
{
    // 定义一对一关联
    public function profile()
    {
        return $this->hasOne('Profile');
    }

}
```

`Profile` 模型代码：

```
namespace app\api\model;

use think\Model;

class Profile extends Model
{
    protected $type       = [
        'birthday' => 'timestamp:Y-m-d',
    ];

}
```

访问 `http://tp5.com/v1/user/10` 返回的数据是：

▼ {id: 10, nickname: "流年", name: "thinkphp", password: "123456", create_time: "2016-07-02 23:35:07",…}
create_time: "2016-07-02 23:35:07"
id: 10
name: "thinkphp"
nickname: "流年"
password: "123456"
status: 1
update_time: "2016-07-02 23:35:07"

访问 `http://tp5.com/v2/user/10` 返回的数据是：

▼ {id: 10, nickname: "流年", name: "thinkphp", password: "123456", create_time: "2016-07-02 23:35:07",…}
create_time: "2016-07-02 23:35:07"
id: 10
name: "thinkphp"
nickname: "流年"
password: "123456"
▼ profile: {id: 3, truename: "刘晨", birthday: "1977-03-05", address: "中国上海", email: "thinkphp@qq.com", user_id: 10}
address: "中国上海"
birthday: "1977-03-05"
email: "thinkphp@qq.com"
id: 3
truename: "刘晨"
user_id: 10
status: 1
update_time: "2016-07-02 23:35:07"

异常处理

当发生异常的时候，通常我们返回不同的 `HTTP` 状态码来标识，控制器的 `read` 方法中当请求的用户不存在的时候，系统发送了 `404` 状态码来表示用户数据不存在，代码为：

```
return json(['error' => '用户不存在'], 404);
```

并返回了错误信息如下：

客户端通常可以直接通过HTTP状态码来判断接口请求的成功与否而自行进行自定义错误提示，一般`400`以上的状态码都表示请求失败，接口的代码还可以简化成：

```
    // 获取用户信息
    public function read($id = 0)
    {
        $user = UserModel::get($id, 'profile');
        if ($user) {
            return json($user);
        } else {
            // 抛出HTTP异常 并发送404状态码
            abort(404);
        }
    }
```

如果你希望由API后台来处理异常，并且直接接管系统的所有异常信息输出json错误信息，可以自定义一个异常处理类位于（`application/api/exception/Http.php`）：

```
namespace app\api\exception;

use think\exception\Handle;
use think\exception\HttpException;

class Http extends Handle
{

    public function render(\Exception $e)
    {
        if ($e instanceof HttpException) {
            $statusCode = $e->getStatusCode();
        }

        if (!isset($statusCode)) {
            $statusCode = 500;
        }

        $result = [
            'code' => $statusCode,
            'msg'  => $e->getMessage(),
            'time' => $_SERVER['REQUEST_TIME'],
        ];
        return json($result, $statusCode);
    }

}
```

然后，在应用配置文件中修改异常处理handle参数为自定义的异常类：

```
'exception_handle'  => '\app\api\exception\Http',
```

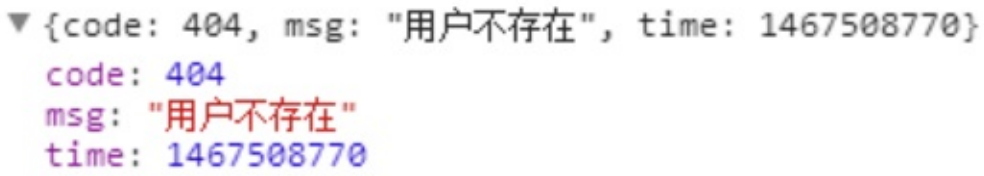
接管`HTTP`异常处理后，我们可以直接在方法中抛出任何的`HTTP`异常，系统自动处理为`json`格式输出到客户端：

```
    // 获取用户信息
    public function read($id = 0)
    {

        $user = UserModel::get($id, 'profile');
        if ($user) {
            return json($user);
        } else {
            // 抛出HTTP异常 并发送404状态码
            abort(404,'用户不存在');
        }
    }
```

当我们请求一个不存在的用户时候

```
http://tp5.com/v2/user/100
```

会看到如下的输出：



如果希望捕获系统的任何异常并转发，可以使用`try catch`如下：

```
    // 获取用户信息
    public function read($id = 0)
    {
        try {
            // 制造一个方法不存在的异常
            $user = UserModel::geet($id, 'profile');
            if ($user) {
                return json($user);
            } else {
                return abort(404, '用户不存在');
            }
        } catch (\Exception $e) {
            // 捕获异常并转发为HTTP异常
            return abort(404, $e->getMessage());
        }
    }
```

执行后会看到系统输出了

▼{code: 404, msg: "method not exist:think\db\Query->geet", time: 1467476927}
code: 404
msg: "method not exist:think\db\Query->geet"
time: 1467476927

RESTFul

`REST`(Representational State Transfer表述性状态转移)是一种针对网络应用的设计和开发方式，可以降低开发的复杂性，提高系统的可伸缩性。`REST`提出了一些设计概念和准则：

1、网络上的所有事物都被抽象为资源（resource）；
2、每个资源对应一个唯一的资源标识（resource identifier）；
3、通过通用的连接器接口（generic connector interface）对资源进行操作；
4、对资源的各种操作不会改变资源标识；
5、所有的操作都是无状态的（stateless）。

`REST`通常基于使用`HTTP`，`URI`，和`JSON`以及`HTML`这些现有的广泛流行的协议和标准。

传统的请求模式和`REST`模式的请求模式区别：

作用	传统模式	REST模式
列举出所有的用户	GET /users/list	GET /users
列出ID为1的用户信息	GET /users/show/id/1	GET /users/1
插入一个新的用户	POST /users/add	POST /users
更新ID为1的用户信息	POST /users/update/id/1	PUT /users/1
删除ID为1的用户	POST /users/delete/id/1	DELETE /users/1

关于更多的REST信息，可以参考：http://zh.wikipedia.org/wiki/REST

下面以一个博客的`REST`接口开发为例，先创建`think_blog`数据表如下：

```
CREATE TABLE IF NOT EXISTS `think_blog` (
  `id` int(10) UNSIGNED NOT NULL AUTO_INCREMENT COMMENT 'ID',
  `name` char(40) NOT NULL DEFAULT '' COMMENT '标识',
  `title` char(80) NOT NULL DEFAULT '' COMMENT '标题',
  `content` text COMMENT '内容',
  `create_time` int(10) UNSIGNED NOT NULL DEFAULT '0' COMMENT '创建时间',
  `update_time` int(10) UNSIGNED NOT NULL DEFAULT '0' COMMENT '更新时间',
  `status` tinyint(1) NOT NULL DEFAULT '0' COMMENT '数据状态',
  PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COMMENT='博客表';
```

> 为了演示需要，该数据表做了一定程度的简化，并不一定符合实际的博客设计，因此仅供测试学习。

为了支持`RESTFul`请求的路由规则，只需要在路由配置文件中添加下面的代码：

```
Route::resource('blogs','index/blog');
```

该方法注册了一个名为 `blogs` 的资源路由，其实内部会自动注册 `7` 个路由规则，包括：

标识	请求类型	生成路由规则	对应操作方法（默认）	描述
index	GET	`blogs`	index	显示博客列表
create	GET	`blogs/create`	create	新增博客页面
save	POST	`blogs`	save	保存博客内容
read	GET	`blogs/:id`	read	查看博客内容
edit	GET	`blogs/:id/edit`	edit	编辑博客页面
update	PUT	`blogs/:id`	update	更新博客内容
delete	DELETE	`blogs/:id`	delete	删除博客

> 上面的7个路由规则，在本示例中仅使用了5个（其中create和edit页面由于本API接口测试暂时不需要使用）。

创建Blog模型如下：

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

use think\Model;

class Blog extends Model
{
    protected $autoWriteTimestamp = true;
    protected $insert             = [
        'status' => 1,
    ];

    protected $field = [
        'id'          => 'int',
        'create_time' => 'int',
        'update_time' => 'int',
        'name', 'title', 'content',
    ];
}
```

为了快速生成控制器类，我们进入命令行，切换到应用根目录下面，执行下面的指令：

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

会自动生成一个 `Blog` 资源控制器类，并且会自动生成如前所述的资源路由对应的7个（空白）方法，我们给每个方法添加一些简单的代码，并去掉了 `create` 和 `edit` 两个方法（对于 `api` 开发而言可以不需要），最终控制器代码如下：

```
<?php

namespace app\index\controller;

use app\index\model\Blog as Blogs;
use think\Controller;
use think\Request;

class Blog extends Controller
{
    /**
     * 显示资源列表
     *
     * @return \think\Response
     */
    public function index()
    {
        $list = Blogs::all();
        return json($list);
    }

    /**
     * 保存新建的资源
     *
     * @param  \think\Request  $request
     * @return \think\Response
     */
    public function save(Request $request)
    {
        $data   = $request->param();
        $result = Blogs::create($data);
        return json($result);
    }

    /**
     * 显示指定的资源
     *
     * @param  int  $id
     * @return \think\Response
     */
    public function read($id)
    {
        $data = Blogs::get($id);
        return json($data);
    }

    /**
     * 保存更新的资源
     *
     * @param  \think\Request  $request
     * @param  int  $id
     * @return \think\Response
     */
    public function update(Request $request, $id)
    {
        $data   = $request->param();
        $result = Blogs::update($data, ['id' => $id]);
        return json($result);
    }

    /**
     * 删除指定资源
```

```
     *
     * @param  int  $id
     * @return \think\Response
     */
    public function delete($id)
    {
        $result = Blogs::destroy($id);
        return json($result);
    }
}
```

> 作为学习用途，Blog接口没有添加数据验证，大家可以结合之前学习的模型验证功能自己添加。

由于`API`开发没有实际的页面显示，加上`REST`请求类型复杂，我们需要通过特殊的方式才能在开发过程中进行`REST`请求的测试。

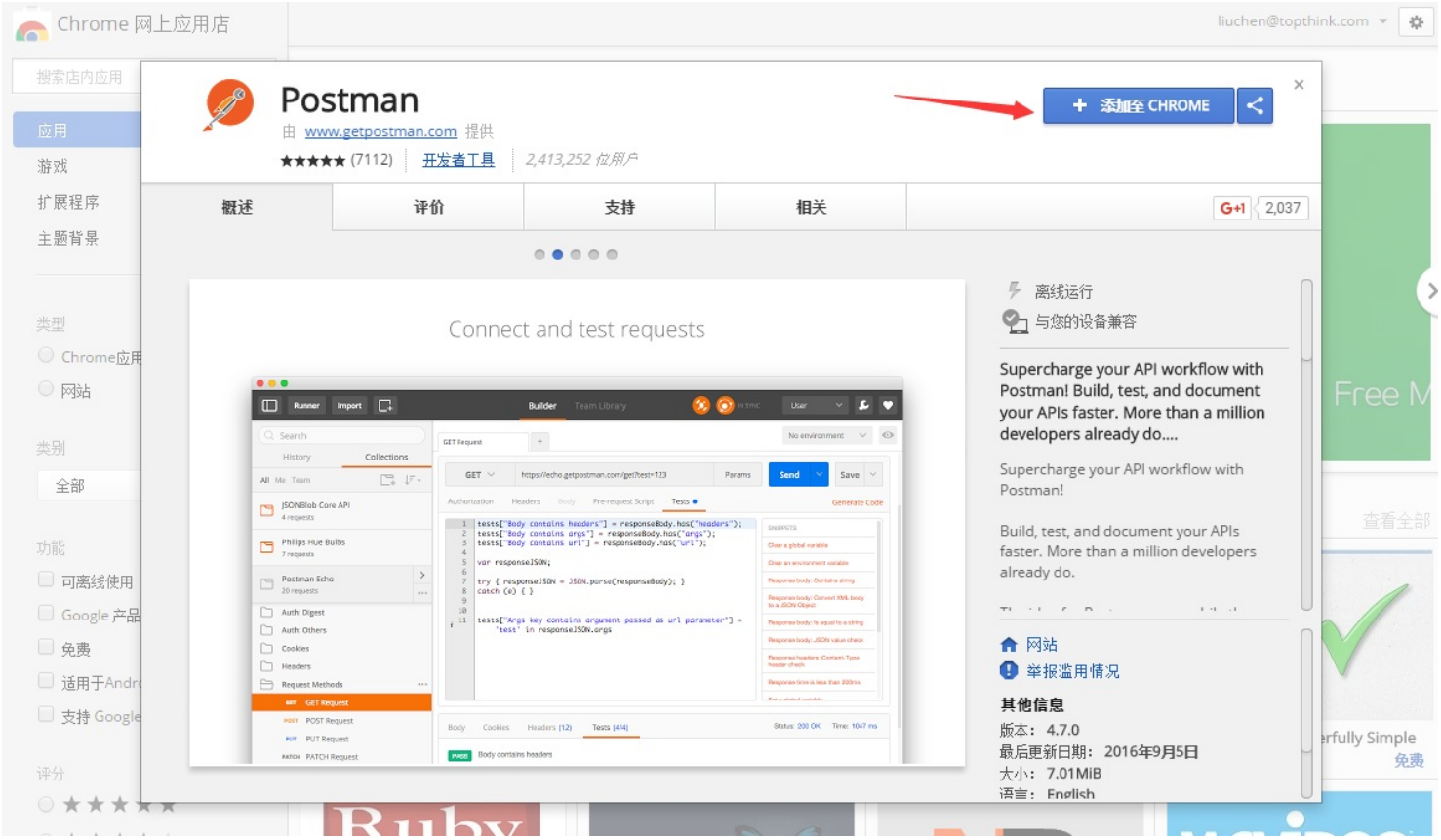
`REST`请求测试

`REST`请求的测试方式有很多，下面介绍两种常用的方式：

Postman

最方便的方法就是通过Postman来测试接口，给Chrome浏览器安装一个`postman`扩展，访问下面地址获取官方扩展：https://www.getpostman.com/

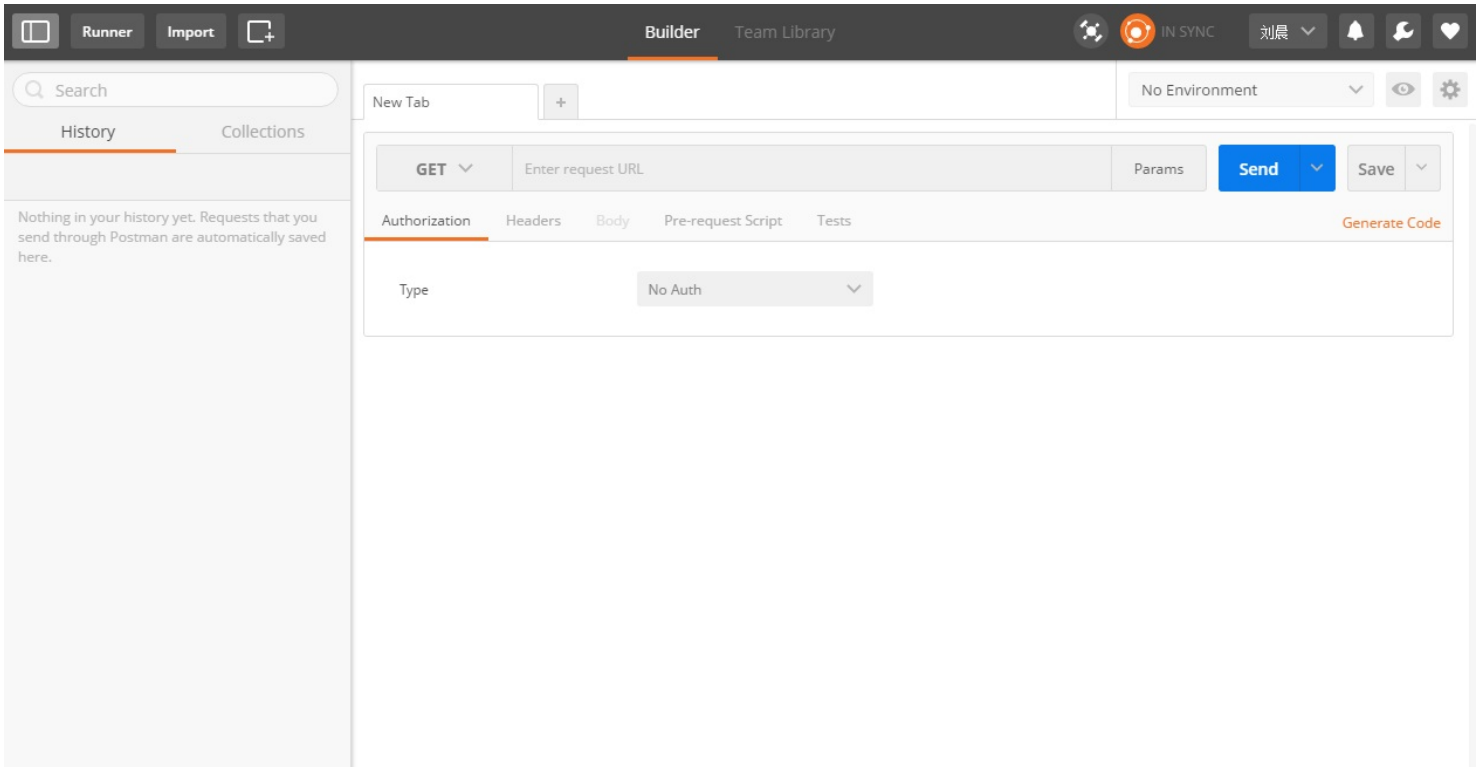
或者直接中chrome的应用商店搜索`postman`（由于众所周知的原因，可能无法正常访问）。



安装完成后会打开

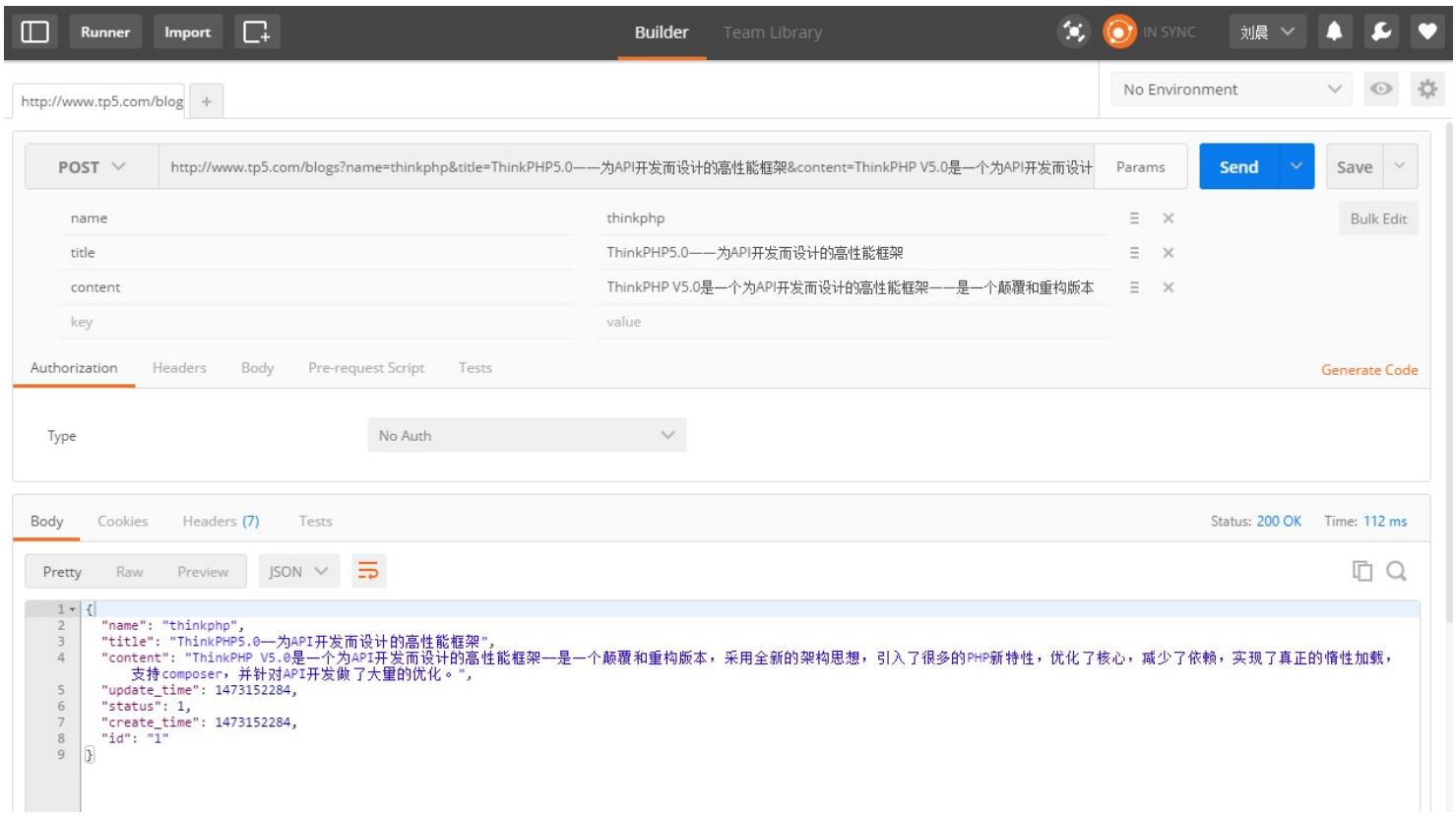
点击`Postman`应用图标就可以打开应用。

如果是首次使用的话，会首先要求注册用户，完成后会进入主界面：



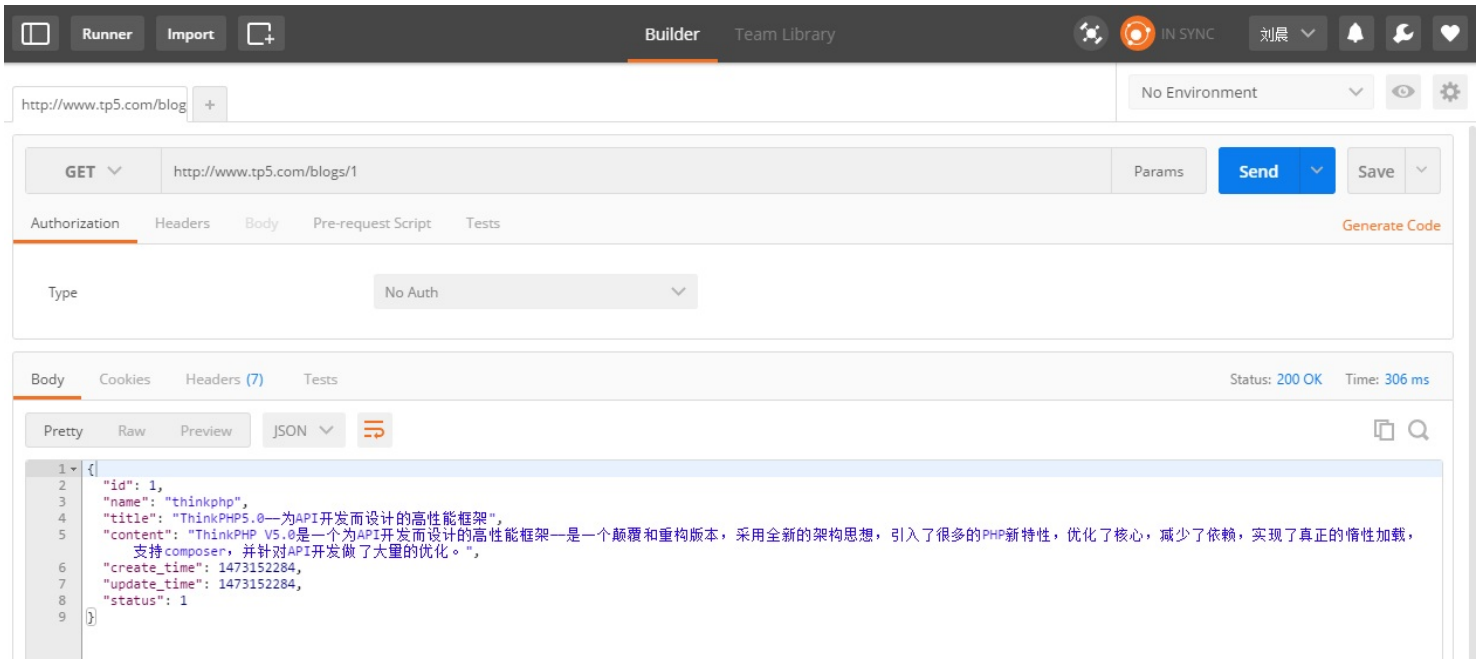
下面我们就来测试下前面的`REST`应用的接口，测试之前选择对应的请求类型，并输入我们的接口地址，如果需要传入参数，则点击`Send`按钮之前的`Params`，依次输入`key`（参数名称）和`value`（参数值），然后点击`Send`按钮即可，下面依次测试博客的接口。

博客添加接口：post http://tp5.com/blogs

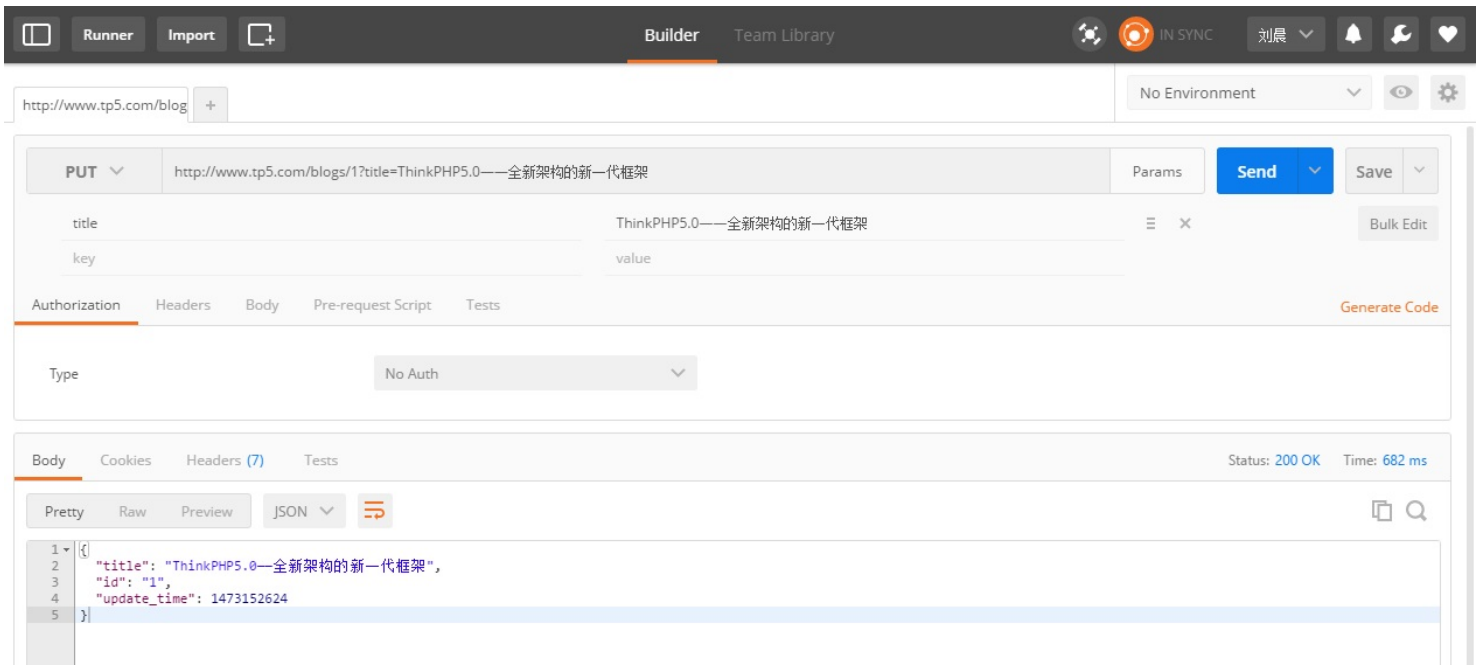


通过添加接口我们写入了两条数据。

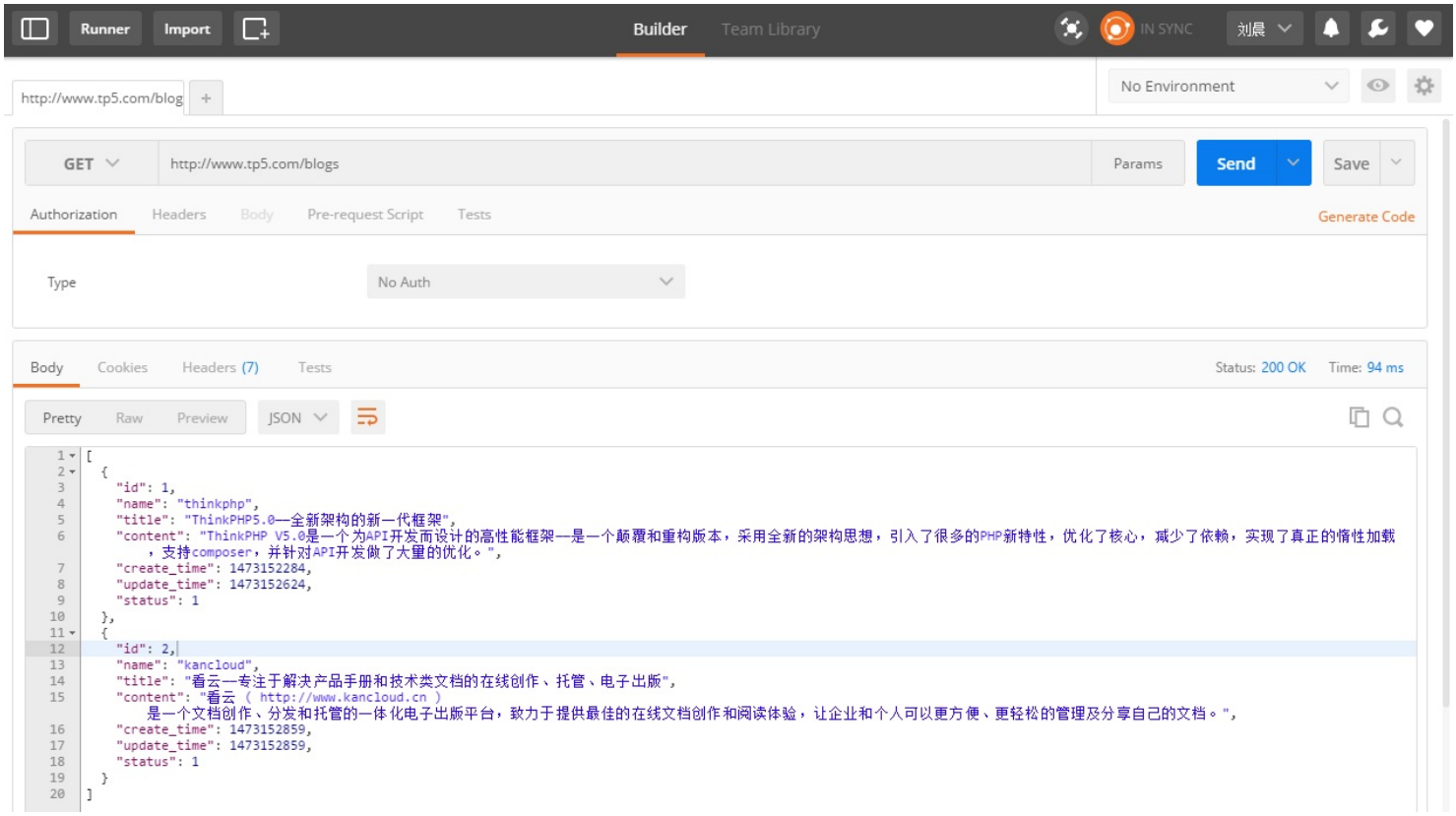
博客读取接口：get http://tp5.com/blogs/1



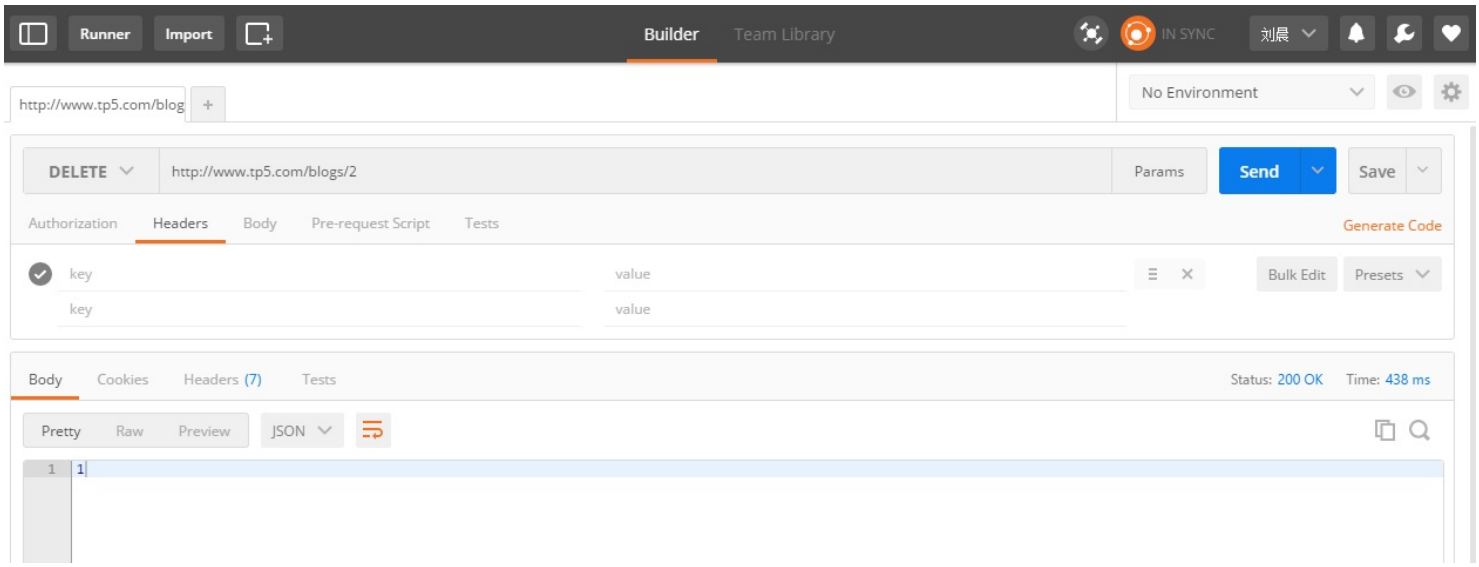
博客更新接口：put http://tp5.com/blogs/1



博客列表接口：get http://tp5.com/blogs



删除博客接口：delete http://tp5.com/blogs/1



如果某个接口出现错误，可以点击 `Preview` 查看错误页面。

REST 请求伪装

除了使用 `Postman` 之外，可以通过一个 `post` 表单来伪装 `REST` 的请求类型。首先创建一个普通的 `post` 表单，把请求参数都作为表单的项目，并且在表单最后添加一个隐藏域 `_method` ，下面的表单模拟了删除博客的请求接口。

```
<form method="post" class="form" action="/blogs/1">
<input type="submit" class="btn" value=" 删除 ">
<input type="hidden" name="_method" value="DELETE" />
```

```
</form>
```

API调试

可以使用 `ThinkPHP5.0` 的 `Trace` 调试中的 `Socket` 调试功能来解决 `API` 开发的调试问题。

环境安装

如果你首次使用，参考下面的安装办法进行 `SocketLog` 的安装。

> **SocketLog安装方法**
>
> 首先，请在chrome浏览器上安装好插件。
> SocketLog首先需要安装chrome插件，Chrome插件安装页面 （需翻墙）
>
> 安装服务端（如果没有nodejs和npm 请首先安装，安装参考），运行下面指令：
>
> ```
> npm install -g socketlog-server
> ```
>
> 安装完成后, 运行命令
>
> ```
> socketlog-server
> ```
>
> 即可启动服务。 将会在本地起一个websocket服务 ，监听端口是1229 。
>
> 如果想服务后台运行，使用：
>
> ```
> socketlog-server > /dev/null &
> ```

浏览器设置

首次使用的时候，需要点击Chrome扩展进行如下设置：

Client_ID用于标识当前用户，注意不要冲突。

应用配置

接下来，修改应用配置文件，修改如下参数：

```
'log'     =>  [
    'type'                  =>  'socket',
    'host'                  =>  'localhost',
    'show_included_files'   =>  true,
    'force_client_ids'      =>  ['slog_b6d7ef'],
    'allow_client_ids'      =>  ['slog_b6d7ef'],
],
```

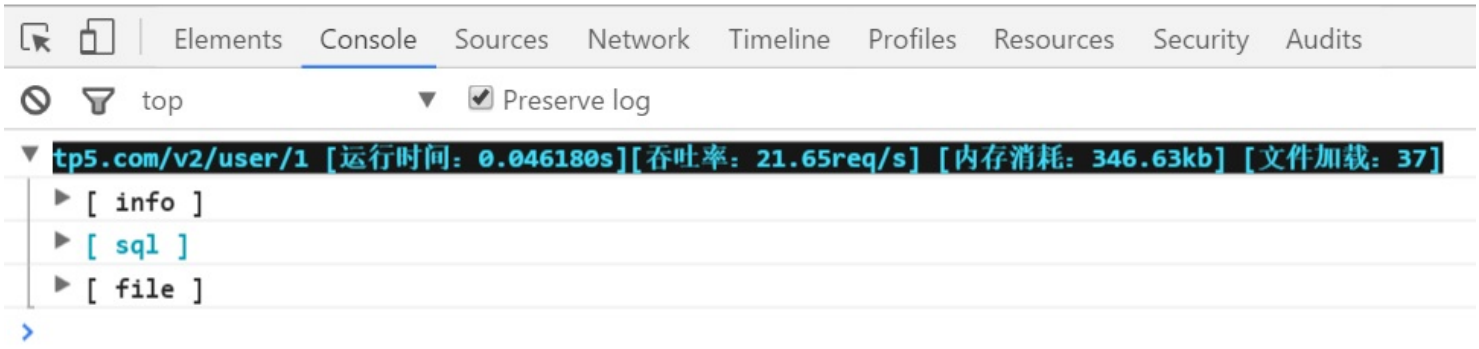
该配置参数把日志类型设置为socket，所有的日志都会写到socket服务器。
如果你的`socket-server`服务器不是`localhost`，配置使用ip地址或者域名即可。

远程调试

下面就可以进行远程调试了，你可以使用任何浏览器访问：

```
http://tp5.com/v2/user/10
```

然后在`Chrome`浏览器中打开`Console`，就可以随时查看该`API`应用的远程调试信息了。



一旦有请求产生，就会自动刷新`Console`信息显示。

还可以支持异常和错误信息的记录，例如：

如果需要指定多个`client_id`进行调试，只需要配置`force_client_ids`和`allow_client_ids`为多个参数即可，例如：

```
'log'     =>  [
    'type'                  =>  'socket',
```

```
    'host'                  =>  'localhost',
    'show_included_files'   =>  true,
    'force_client_ids'      =>  ['slog_b6d7ef','slog_abd89d'],
    'allow_client_ids'      =>  ['slog_b6d7ef','slog_abd89d'],
],
```

安全建议

- 尽量采用HTTPS协议进行接口请求；
- 重要的功能加密传输；
- 做好接口的身份认证；
- 对URL中的参数做好安全过滤；
- 对接口请求做好请求速率限制；
- 重要ID不透明处理；
- 使用JSON格式返回数据；

详细可以参考 《REST API 安全设计指南》

十、命令行工具

快速入门（十）：命令行工具

`ThinkPHP5.0`开始增加了一个命令行工具`think`，并且内置了一些指令，可以帮助开发者在开发或者运维阶段执行一些指令，本章我们来了解下命令行工具的使用和指令开发。

查看指令

> 命令行工具需要在命令行下面执行，请先确保你的`php.exe`已经加入了系统环境变量。

应用的命令行入口文件是应用根目录的`think`文件，其内容如下：

```
// 定义项目路径
define('APP_PATH', './application/');

// 加载框架命令行引导文件
require './thinkphp/console.php';
```

> 你也可以自定义`think`文件，更改应用目录。

要执行命令，首先进入命令行，并切换当前目录到应用的根目录（也就是`think`文件所在目录）下面，执行：

```
php think
```

会显示当前支持的所有指令：

```
>php think
Think Console version 0.1
```

```
Usage:
  command [options] [arguments]

Options:
  -h, --help            Display this help message
  -V, --version         Display this console version
  -q, --quiet           Do not output any message
      --ansi            Force ANSI output
      --no-ansi         Disable ANSI output
  -n, --no-interaction  Do not ask any interactive question
  -v|vv|vvv, --verbose  Increase the verbosity of messages: 1 for normal output, 2 for
more verbose output and 3 for debug

Available commands:
  build              Build Application Dirs
  help               Displays help for a command
  list               Lists commands
 make
  make:controller    Create a new controller class
  make:model         Create a new model class
 optimize
  optimize:autoload  Optimizes PSR0 and PSR4 packages to be loaded with classmaps too,
good for production.
```

> **注意**
>
> 在win10的周年版中运行会出现乱码，请使用 `PowerShell` 运行。

使用

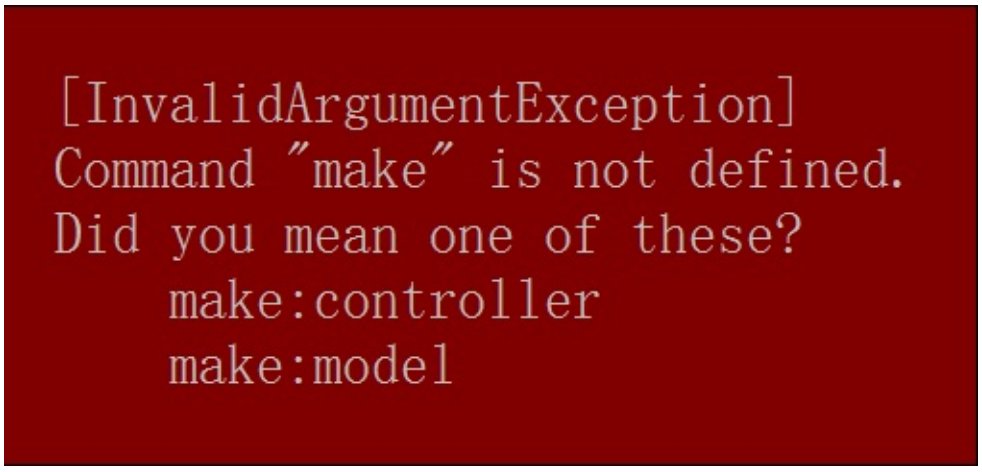
```
>php think list
```

可以获得相同的结果。

如果输入一个不存在的指令，系统会自动搜索相关的指令并提示，例如：

```
>php think make
```

会显示



生成模块

下面我们给应用生成一个新的模块`test`，首先需要在`application`目录下面创建一个`build.php`定义文件，文件内容如下：

```
return [
    // 定义test模块的自动生成
    'test' => [
        '__dir__'    => ['controller', 'model', 'view'],
        'controller' => ['User', 'UserType'],
        'model'      => ['User', 'UserType'],
        'view'       => ['index/index', 'index/test'],
    ],
];
```

然后在命令行下面，执行：

```
>php think build
Successed
```

如果显示Successed则表示生成成功。

> 注意，命令行指令是区分大小写的，所以如果执行
>
> ```
> >php think Build
> ```
>
> 会报错。

我们可以看到`application`目录下面已经生成了一个`test`模块目录，包括下面的子目录及文件：

```
test
├─controller
│     ├─Index.php
│     ├─User.php
│     └─UserType.php
├─model
│     ├─User.php
│     └─UserType.php
├─view
│     └─index
│       ├─index.html
│       └─test.html
├─common.php
└─config.php
```

接下来，我们可以访问

```
http://tp5.com/test/
```

会显示：

我们在 `build.php` 文件中并没有定义 `Index` 控制器，但仍然生成了一个默认的 `Index` 控制器文件以及欢迎页面，这是为了避免模块访问出错。

生成文件

还可以用 `make` 指令单独生成某个应用类库文件，例如：

```
php think make:controller test/Blog
```

会自动为test模块生成一个 Blog控制器文件。

> 注意，默认生成的控制器类是属于资源控制器，并且继承了\think\Controller。

如果希望生成一个空的控制器，可以使用

```
php think make:controller test/Blog --plain
```

又或者生成一个模型文件

```
php think make:model test/Blog
```

生成类库映射文件

在生成类库文件之后，我们强烈建议使用命令行生成类库映射文件，可以提高自动加载的性能，用法如下：

```
>php think optimize:autoload
```

执行完毕，会在`RUNTIME_PATH`目录下面生成一个`classmap.php`文件，包括了系统和应用的所有类库文件的映射列表。

生成路由缓存文件

如果你的应用定义了大量的路由规则，那么建议在实际部署后生成路由缓存文件，可以免去路由注册的开销，从而改善路由的检测效率，用法如下：

```
>php think optimize:route
```

执行完毕，会在`RUNTIME_PATH`目录下面生成一个`route.php`文件，包括了应用的所有路由规则定义列表。

> **注意**
>
> 路由缓存文件只会缓存在application/route.php文件中配置和动态注册的路由规则，因此请确保你没有在其它的文件中进行路由的注册。

生成数据表字段缓存文件

如果你希望提高查询的性能，可以通过生成字段缓存来减少查询。用法如下：

```
>php think optimize:schema
```

执行完毕，会在`RUNTIME_PATH`目录下面创建schema目录，然后在该目录下面按照`database.table.php`的文件命名生成数据表字段缓存文件。

也可以指定数据库生成字段缓存（必须有用户权限），例如，下面指定生成demo数据库下面的所有数据表的字段缓存信息。

```
php think optimize:schema --db demo
```

还可以读取模块的模型类来生成数据表字段缓存（这个适合多数据库连接的情况），如下：

```
php think optimize:schema --module index
```

会读取index模块的模型来生成数据表字段缓存。

> **注意**

> 如果模型类没有继承think\Model或者是抽象类的话，不会生成对应模型的字段缓存。

更新数据表字段缓存也是同样的方式，每次执行都会重新生成缓存。如果需要单独更新某个数据表的缓存，可以使用：

```
php think optimize:schema --table think_user
```

支持指定数据库名称

```
php think optimize:schema --table demo.think_user
```

指令扩展示例

命令行的作用远不止生成文件这么简单，同时应用也可以很方便的扩展自己的指令，我们来扩展一个用于清理缓存目录的指令`clear`。

> **注意**
>
> 最新版本已经内置`clear`指令了，下面的示例可以直接使用内置的`clear`指令进行学习和测试。

首先，我们创建一个指令类`app\console\Clear`，内容如下：

```
<?php
namespace app\console;

use think\console\command\Command;
use think\console\Input;
use think\console\input\Option;
use think\console\Output;

class Clear extends Command
{
    protected function configure()
    {
        // 指令配置
        $this
            ->setName('clear')
            ->addOption('path', 'd', Option::VALUE_OPTIONAL, 'path to clear', null)
            ->setDescription('Clear runtime file');
    }

    protected function execute(Input $input, Output $output)
    {
        $path  = $input->getOption('path') ?: RUNTIME_PATH;
        $files = scandir($path);
        if ($files) {
            foreach ($files as $file) {
                if ('.' != $file && '..' != $file && is_dir($path . $file)) {
                    array_map('unlink', glob($path . $file . '/*.*'));
                } elseif (is_file($path . $file)) {
                    unlink($path . $file);
```

```
                }
            }
        }
        $output->writeln("Clear Successed");
    }
}
```

> 一个合法的指令类，没有固定的目录和命名空间要求，但必须继承 `think\console\command\Command` 或者其子类，并且定义 `configure` 和 `execute` 两个方法。

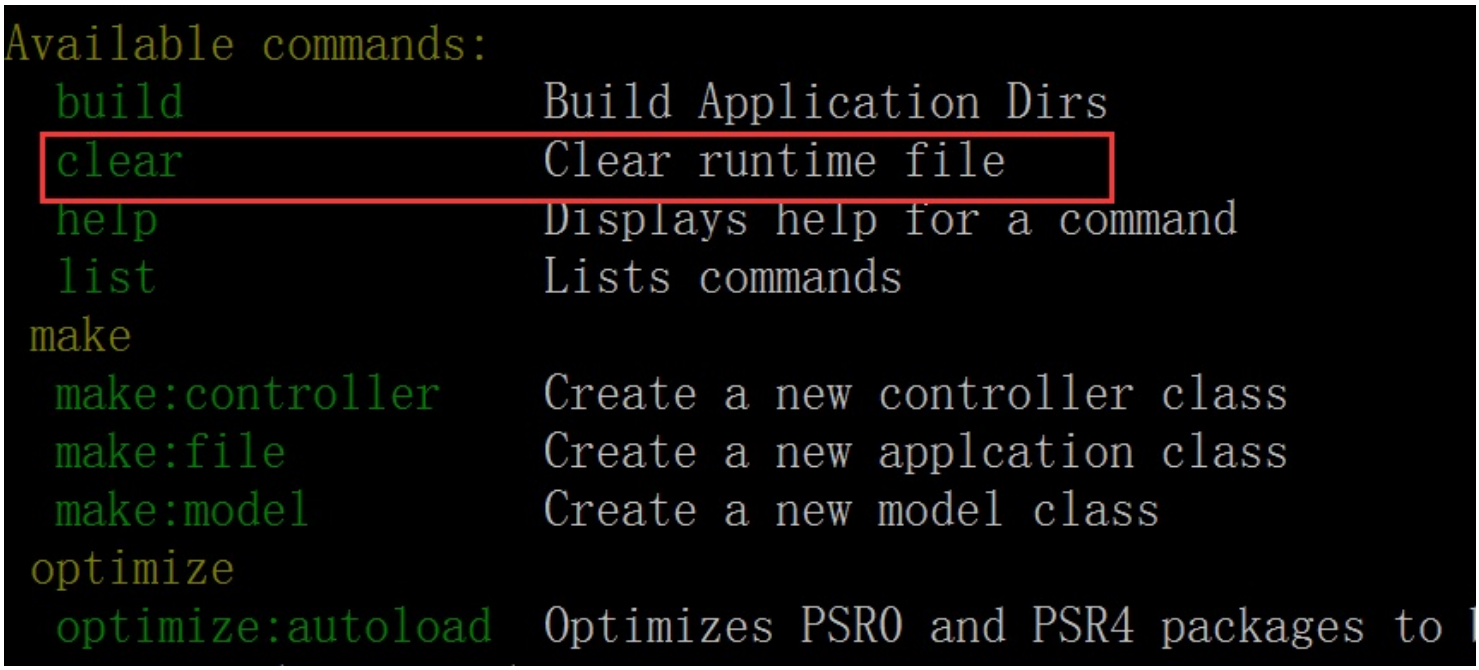
然后，在application目录下面的 `command.php` （如果不存在则创建）文件中添加如下内容：

```
return [
    '\app\console\Clear',
];
```

表示给应用增加一个命令行 `Clear` 指令，我们可以用 `list` 指令来验证是否已经成功注册 `Clear` 指令：

```
>php think list
```

运行后如果看到



表示指令注册成功，接下来可以测试下该指令：

```
>php think clear
Clear Successed
```

> 该指令并不会删除目录，仅仅删除目录下面（包括子目录）的文件。

`clear` 指令还定义了一个 `--path` 参数用于指定目录，下面是一个指定目录删除文件的用法，我们仅仅需要删除日志文件：

```
>php think clear --path d:\www\tp5\runtime\log\
Clear Successed
```

`--path` 参数还有一个简化用法 `-d`

```
>php think clear -d d:\www\tp5\runtime\log\
Clear Successed
```

命令行调试

命令行一旦执行错误，只能看到简单的错误信息，如果需要调试详细的Trace信息，可以使用 `-v` 参数来显示，例如：

假设Clear指令文件中使用了一个未定义的变量 `pathp`，那么我们可以使用

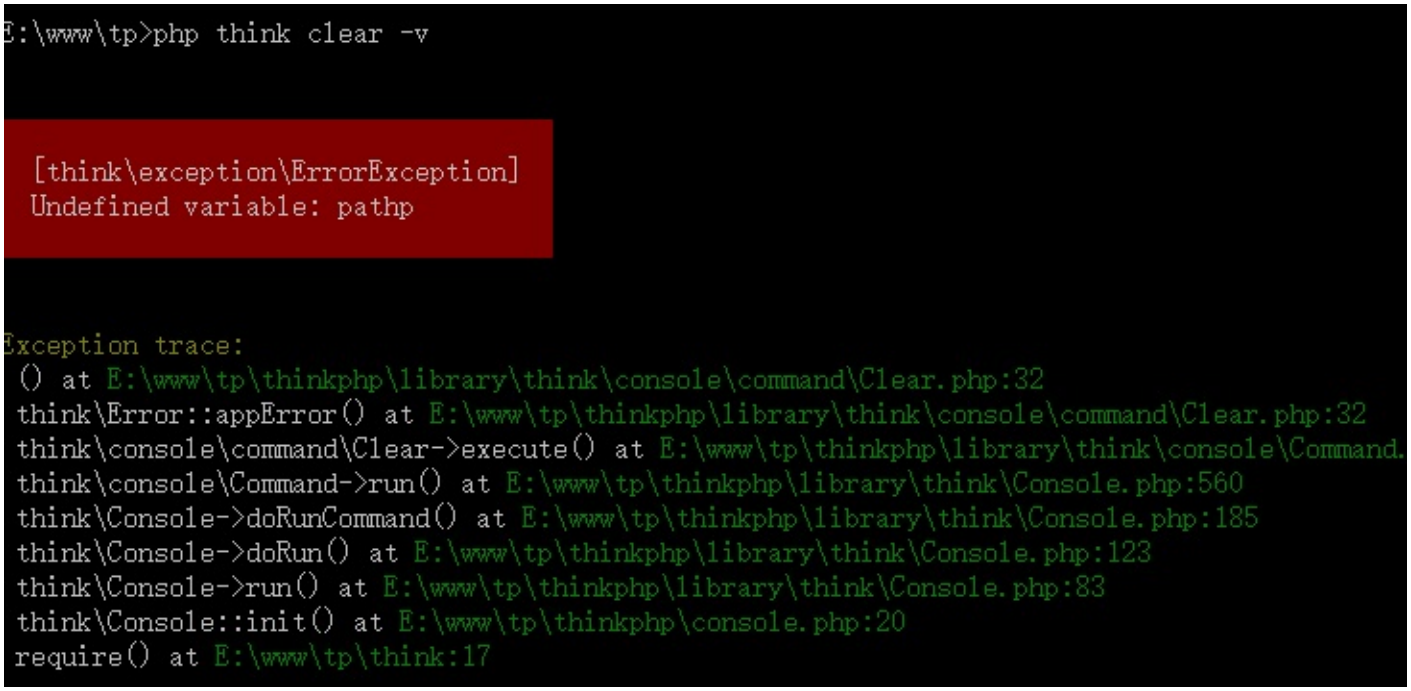
```
>php think clear
```

会显示如下错误：

我们需要查看具体的Trace信息，可以使用

```
>php think clear -v
```

会看到类似下面的错误信息：



命令行颜色支持

为了让命令行工具更加有趣，`think`命令行支持颜色输出，并且内置了几种颜色样式，还可以自己自定义颜色样式输出。

> **windows下面命令行颜色输出支持需要 windows 10.0.10580 以上版本支持**

我们添加一个`color`指令用于演示颜色输出效果，代码如下：

```
<?php
namespace app\console;

use think\console\command\Command;
use think\console\Input;
use think\console\Output;
use think\console\output\formatter\Style;

class Color extends Command
{
    protected function configure()
    {
        $this
            ->setName('color')
            ->setDescription('Show Color text');
    }

    protected function execute(Input $input, Output $output)
    {
        // 输出info样式
        $output->writeln("<info>this is info</info>");
        // 输出error样式
        $output->writeln("<error>this is error</error>");
        // 输出comment样式
        $output->writeln("<comment>this is comment</comment>");
        // 输出question样式
        $output->writeln("<question>this is question</question>");
        // 输出highlight样式
```

```
        $output->writeln("<highlight>this is highlight</highlight>");
        // 输出warning样式
        $output->writeln("<warning>this is warning</warning>");
        // 输出混合样式
        $output->writeln("this is <info>info</info>, this is <error>error</error>,this
is <comment>comment</comment>,this is <question>question</question>,this is <highlight>
highlight</highlight>, this is <warning>warning</warning>");
        // 自定义输出样式
        $output->getFormatter()->setStyle('custom', new Style('black', 'white'));
        $output->writeln("<custom>this is style</custom>");
    }
}
```

`command.php` 定义修改如下：

```
return [
    '\app\console\Clear',
    '\app\console\Color',
];
```

然后执行下面的指令：

```
>php think color
```

运行后就可以看到：

```
this is info
this is error
this is comment
this is question
this is highlight
this is warning
this is info, this is error,this is comment,this is question,this is highlight,
this is warning
this is style
```

调用命令

在代码里面可以直接调用执行命令行的某个命令，例如：

```
namespace app\index\controller;

use think\Console;

class Index
{
    public function index()
    {
        // 调用命令行的指令
        $output = Console::call('make:model',['index/Blog']);
        return $output->fetch();
    }
```

```
}
```

> Console::call方法的第一个参数就是指令名称，后面的第二个参数是一个数组，表示调用的参数。
> 如果我们要创建一个`demo`模块的话，应该是：
>
> ```
> Console::call('build',['--module', 'demo']);
> ```

当访问

```
http://tp5.com
```

页面会输出

```
Model created successfully.
```

可以在`application/index/model/`目录下面发现已经生成了一个`Blog`模型文件。

当我们再次刷新页面的话，会看到页面输出

```
Model already exists!
```

表示模型已经创建过了，无需再次创建。

> 使用`Console::call`方法调用指令执行不会看到最终的输出结果，需要使用fetch方法获取输出信息，一旦发生错误，则会抛出异常。

十一、扩展

快速入门（十一）：扩展

ThinkPHP5 对扩展的支持更加方便和灵活，5.0 版本的扩展原则是不在核心目录添加任何扩展和文件，以确保核心框架的更新。

- 函数扩展
- 类库扩展
- 驱动扩展
- Composer扩展

函数扩展

如果要扩展自己的全局自定义函数，或者重写系统的助手函数，可以直接在应用目录下面的 common.php 文件（ application/common.php ）中定义，例如：

```
// 增加一个新的table助手函数
function table($table, $config = [])
{
    return \think\Db::connect($config)->setTable($table);
}

// 替换框架内置的db助手函数
function db($name, $config = [])
{
    return \think\Db::connect($config)->name($name);
}
```

如果只是定义某个模块需要使用的函数，则可以放到模块的公共文件中，例如定义 index 模块使用的函数可以在 application/index/common.php 中。

类库扩展

如果你需要在核心之外扩展和使用第三方类库，并且该类库不是通过 Composer 安装使用，那么可以直接放入应用根目录下面的 extend 目录下面，该目录是官方建议的第三方扩展类库目录。

extend 目录下面的类库必须遵循 PSR-4 自动加载规范，系统会按照目录自动注册命名空间，下面是一个 org\util\ArrayList 类：

```
<?php
namespace org\util;

/**
 * ArrayList实现类
 * @author    liu21st <liu21st@gmail.com>
 */
class ArrayList implements \IteratorAggregate
{
    /**
```

```
     * 集合元素
     * @var array
     * @access protected
     */
    protected $elements = [];

    /**
     * 架构函数
     * @access public
     * @param string $elements  初始化数组元素
     */
    public function __construct($elements = [])
    {
        if (!empty($elements)) {
            $this->elements = $elements;
        }
    }

    /**
     * 若要获得迭代因子，通过getIterator方法实现
     * @access public
     * @return ArrayObject
     */
    public function getIterator()
    {
        return new \ArrayObject($this->elements);
    }

    /**
     * 增加元素
     * @access public
     * @param mixed $element  要添加的元素
     * @return boolean
     */
    public function add($element)
    {
        return (array_push($this->elements, $element)) ? true : false;
    }

    // 在数组开头插入一个单元
    public function unshift($element)
    {
        return (array_unshift($this->elements, $element)) ? true : false;
    }

    // 将数组最后一个单元弹出（出栈）
    public function pop()
    {
        return array_pop($this->elements);
    }

    /**
     * 增加元素列表
     * @access public
     * @param ArrayList $list  元素列表
     * @return boolean
     */
    public function addAll($list)
    {
        $before = $this->size();
        foreach ($list as $element) {
            $this->add($element);
        }
```

```
        $after = $this->size();
        return ($before < $after);
    }

    /**
     * 清除所有元素
     * @access public
     */
    public function clear()
    {
        $this->elements = [];
    }

    /**
     * 是否包含某个元素
     * @access public
     * @param mixed $element  查找元素
     * @return string
     */
    public function contains($element)
    {
        return (array_search($element, $this->elements) !== false);
    }

    /**
     * 根据索引取得元素
     * @access public
     * @param integer $index 索引
     * @return mixed
     */
    public function get($index)
    {
        return $this->elements[$index];
    }

    /**
     * 查找匹配元素，并返回第一个元素所在位置
     * 注意 可能存在0的索引位置 因此要用===False来判断查找失败
     * @access public
     * @param mixed $element  查找元素
     * @return integer
     */
    public function indexOf($element)
    {
        return array_search($element, $this->elements);
    }

    /**
     * 判断元素是否为空
     * @access public
     * @return boolean
     */
    public function isEmpty()
    {
        return empty($this->elements);
    }

    /**
     * 最后一个匹配的元素位置
     * @access public
     * @param mixed $element  查找元素
     * @return integer
     */
```

```
    public function lastIndexOf($element)
    {
        for ($i = (count($this->elements) - 1); $i > 0; $i--) {
            if ($this->get($i) == $element) {
                return $i;
            }
        }
    }

    public function toJson()
    {
        return json_encode($this->elements);
    }

    /**
     * 根据索引移除元素
     * 返回被移除的元素
     * @access public
     * @param integer $index 索引
     * @return mixed
     */
    public function remove($index)
    {
        $element = $this->get($index);
        if (!is_null($element)) {
            array_splice($this->elements, $index, 1);
        }
        return $element;
    }

    /**
     * 移出一定范围的数组列表
     * @access public
     * @param integer $offset  开始移除位置
     * @param integer $length  移除长度
     */
    public function removeRange($offset, $length)
    {
        array_splice($this->elements, $offset, $length);
    }

    /**
     * 移出重复的值
     * @access public
     */
    public function unique()
    {
        $this->elements = array_unique($this->elements);
    }

    /**
     * 取出一定范围的数组列表
     * @access public
     * @param integer $offset  开始位置
     * @param integer $length  长度
     */
    public function range($offset, $length = null)
    {
        return array_slice($this->elements, $offset, $length);
    }

    /**
     * 设置列表元素
```

```
     * 返回修改之前的值
     * @access public
     * @param integer $index 索引
     * @param mixed $element  元素
     * @return mixed
     */
    public function set($index, $element)
    {
        $previous                = $this->get($index);
        $this->elements[$index]  = $element;
        return $previous;
    }

    /**
     * 获取列表长度
     * @access public
     * @return integer
     */
    public function size()
    {
        return count($this->elements);
    }

    /**
     * 转换成数组
     * @access public
     * @return array
     */
    public function toArray()
    {
        return $this->elements;
    }

    // 列表排序
    public function ksort()
    {
        ksort($this->elements);
    }

    // 列表排序
    public function asort()
    {
        asort($this->elements);
    }

    // 逆向排序
    public function rsort()
    {
        rsort($this->elements);
    }

    // 自然排序
    public function natsort()
    {
        natsort($this->elements);
    }

}
```

我们把类文件放到`extend/org/util/ArrayList.php`，就可以直接使用该类了，注意`ArrayList`类的完整命名空间应该是`org\util\ArrayList`，而不是`extend\org\util\ArrayList`，下面我们来

写个例子测试下是否可以正常使用该扩展类库，控制器代码如下：

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

use org\util\ArrayList;

class Index
{
    public funtion index()
    {
        $list       = ['thinkphp', 'thinkphp', 'onethink', 'topthink'];
        $arrayList  = new ArrayList($list);
        $arrayList->add('kancloud');
        $arrayList->unique();
        dump($arrayList->toArray());
        echo $arrayList->toJson();
    }
}
```

访问输出的结果为：

```
array (size=4)
  0 => string 'thinkphp' (length=8)
  2 => string 'onethink' (length=8)
  3 => string 'topthink' (length=8)
  4 => string 'kancloud' (length=8)
{"0":"thinkphp","2":"onethink","3":"topthink","4":"kancloud"}
```

`extend` 扩展目录通常是放一些全局并且和应用无关的公共类库，如果是应用相关的类库则可以纳入应用目录，例如：

```
<?php
namespace app\common\util;

class Auth
{
    public static funtion check($uid)
    {
        return true;
    }
}
```

类库文件位于`application/common/util/Auth.php`。

驱动扩展

驱动扩展其实是一种比较特别的类库扩展而已，因为针对不同的驱动有不同的方法规范，驱动扩展不需要放入核心目录，你可以在任何位置扩展相关驱动，官方推荐的方式是在`extend`目录下面的`driver`子目录放置所有的驱动扩展。

```
extend/driver
    ├─cache     // 缓存扩展驱动目录
```

```
    ├─db        // 数据库扩展驱动目录
    ├─debug     // 调试扩展驱动目录
    ├─log       // 日志扩展驱动目录
    ├─session   // SESSION扩展驱动目录
    ├─template  // 模板标签扩展驱动目录
    ├─...
```

以日志扩展为例，我们来扩展一个日志驱动满足项目的实际需求，该日志驱动会把指定的日志信息发送到指定邮箱，驱动类实现如下：

```
<?php
namespace driver\log;

/**
 * 本地化调试输出到文件
 */
class Email
{
    protected $config = [
        'time_format' => ' c ',
        'email_addr'  => '',
        'send_level'  => ['error'],
    ];

    // 实例化并传入参数
    public function __construct($config = [])
    {
        if (is_array($config)) {
            $this->config = array_merge($this->config, $config);
        }
    }

    /**
     * 日志写入接口
     * @access public
     * @param array $log 日志信息
     * @return bool
     */
    public function save(array $log = [])
    {
        $now    =   date($this->config['time_format']);
        $info   =   '';
        $server = isset($_SERVER['SERVER_ADDR']) ? $_SERVER['SERVER_ADDR'] : '0.0.0.0';
        $remote = isset($_SERVER['REMOTE_ADDR']) ? $_SERVER['REMOTE_ADDR'] : '0.0.0.0';
        $method = isset($_SERVER['REQUEST_METHOD']) ? $_SERVER['REQUEST_METHOD'] : 'CLI
';
        $uri    = isset($_SERVER['REQUEST_URI']) ? $_SERVER['REQUEST_URI'] : '';
        foreach ($log as $type => $val) {
            if (in_array($type, $this->config['send_level'])) {
                foreach ($val as $msg) {
                    if (!is_string($msg)) {
                        $msg = var_export($msg, true);
                    }
                    $info .= '[ ' . $type . ' ] ' . $msg . "<br/>";
                }
            }
        }
        return error_log("[{$now}] {$server} {$remote} {$method} {$uri}<div>{$info}</di
v>", 1, $this->config['email_addr'],"Subject: Log-{$now}");
    }
```

```
}
```

然后，修改应用配置文件中的`log`参数为：

```
'log'   =>  [
    'type'          =>  'driver\log\Email',
    'email_addr'    =>  'thinkphp@qq.com',
    'send_level'    =>  ['error', 'info'],
],
```

为了确保邮件发送正常，你可能还需要配置邮件服务参数，如果是`windows`服务器，配置`php.ini`文件中的相关参数

```
SMTP = 你的smtp服务器host（域名或者IP）
smtp_port = 端口号（默认25）
sendmail_from = "发送邮件的来源信息"
```

如果是`Linux`系统，一般无需配置或者在`php.ini`设置`sendmail_path`参数：

```
sendmail_path = sendmail 程序的路径（通常是 /usr/sbin/sendmail 或 /usr/lib/sendmail）
```

下面是一个测试例子：

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

use think\Log;

class Index
{
    public function index()
    {
        Log::error('测试错误');
    }
}
```

当运行后，如果你的邮件服务配置正确，会收到一份邮件通知，否则可能会抛出异常页面。

Composer扩展

很多情况下，我们的扩展需要依赖其它的第三方扩展，那么为了有效解决依赖问题，使用Composer扩展包是最方便的开发模式。

关于如何使用`composer`扩展我们会在后面介绍官方扩展的时候给你详细描述。

十二、杂项

快速入门（十二）：工具类库

本章主要给大家讲述一些ThinkPHP5的工具类库的用法和技巧。

> 因为很多工具类库需要使用扩展库，因此在使用工具类库之前，你最好先安装和掌握 `Composer` 的基础用法，从而可以更简单的使用扩展工具类。

Session

会话

`ThinkPHP5.0`使用`think\Session`类进行`Session`的操作管理。

`Session`初始化

大多数情况，我们不需要手动进行Session初始化操作。`ThinkPHP5`会在第一次调用`Session`类的时候按照配置的参数自动初始化和开启`Session`（如果`auto_start`设置为`true`的话），例如，我们在应用配置中添加如下配置：

```
'session'               => [
    'prefix'         => 'think',
    'type'           => '',
    'auto_start'     => true,
],
```

无需任何操作就可以直接调用`Session`类的相关方法，例如：

```
Session::set('name','thinkphp');
Session::get('name');
```

如果你的应用不同模块需要不同的`session`配置参数，那么可以在模块配置文件中重新设置：

```
'session'               => [
    'prefix'         => 'module',
    'type'           => '',
    'auto_start'     => true,
],
```

或者在模块的公共文件中调用`init`方法进行初始化：

```
Session::init([
    'prefix'         => 'module',
    'type'           => '',
    'auto_start'     => true,
```

```
]);
```

> 如果你没有使用Session类进行Session操作的话，例如直接操作`$_SESSION`,必须使用上面的方式手动初始化或者直接调用`session_start()`方法进行`session`初始化，但不建议直接操作`$_SESSION`全局变量。

读取`Session`

建议的读取`Session`数据的方法是通过`Request`请求对象的`session`方法，例如：

```
namespace app\index\controller;

use think\Request;

class User
{
    public function index(Request $request)
    {
        echo $request->session('user_name');
        // 读取二维数组
        echo $request->session('user.name');
    }
}
```

> 通过Request对象读取Session数据支持默认值及过滤方法，因此也更加安全，并且支持多维数组的读取。

当然也支持使用`Session`类直接读取数据：

```
namespace app\index\controller;

use think\Session;

class User
{
    public function index()
    {
        echo Session::get('user_name');
        echo Session::get('user.name');
    }
}
```

> `Session`类的`get`方法只支持最大二维数组的读取，而`Request`对象的`session`方法可以支持任意级别的二维数组获取。

`SESSION`操作

使用`think\Session`类进行`Session`的操作和管理，例如：

```
namespace app\index\controller;
```

```
use think\Session;

class User
{
    public function index()
    {
        // 赋值（当前作用域）
        Session::set('name','thinkphp');
        // 赋值think作用域
        Session::set('name','thinkphp','think');
        // 判断（当前作用域）是否赋值
        Session::has('name');
        // 判断think作用域下面是否赋值
        Session::has('name','think');
        // 取值（当前作用域）
        Session::get('name');
        // 取值think作用域
        Session::get('name','think');
        // 指定当前作用域
        Session::prefix('think');
        // 删除（当前作用域）
        Session::delete('name');
        // 删除think作用域下面的值
        Session::delete('name','think');
        // 清除session（当前作用域）
        Session::clear();
        // 清除think作用域
        Session::clear('think');
        // 赋值（当前作用域）
        Session::set('name.item','thinkphp');
        // 判断（当前作用域）是否赋值
        Session::has('name.item');
        // 取值（当前作用域）
        Session::get('name.item');
        // 删除（当前作用域）
        Session::delete('name.item');
    }
}
```

助手函数

系统也提供了助手函数`session`完成相同的功能，例如：

```
// 初始化session
session([
    'prefix'     => 'module',
    'type'       => '',
    'auto_start' => true,
]);

// 赋值（当前作用域）
session('name', 'thinkphp');
// 赋值think作用域
session('name', 'thinkphp', 'think');
// 判断（当前作用域）是否赋值
session('?name');
// 取值（当前作用域）
session('name');
// 取值think作用域
session('name', '', 'think');
```

```
// 删除（当前作用域）
session('name', null);
// 清除session（当前作用域）
session(null);
// 清除think作用域
session(null, 'think');
```

模板输出

如果需要在模板中输出Session数据，可以使用下面的方法：

```
{$Request.session.user_name}
```

也可以支持二维数组的输出

```
{$Request.session.user.name}
```

`Session` 驱动

支持指定 Session 驱动，配置文件如下：

```
'session' => [
    'prefix'     => 'module',
    'type'       => 'redis',
    'auto_start' => true,
     // redis主机
    'host'       => '127.0.0.1',
     // redis端口
    'port'       => 6379,
     // 密码
    'password'   => '',
]
```

表示使用 `redis` 作为 `session` 类型。

Session示例

下面举一个例子。

```
namespace app\index\controller;

use think\Controller;
use think\Session;

class Index extends Controller
{
    public function index()
    {
        return $this->fetch();
    }

    public function save($name='')
    {
        Session::set('user_name',$name);
```

```
            $this->success('Session设置成功');
        }
    }
```

定义默认文件（`application/index/view/index.html`）如下：

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>Session示例</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>Session示例</h2>
<FORM method="post" class="form" action="{:url('save')}">
输入用户名：<INPUT type="text" class="text" name="name" value="{$Request.session.user_nam
e}" ><br/>
<INPUT type="submit" class="btn" value=" 保存 ">
</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>
```

访问URL地址

```
http://tp5.com
```

页面显示：

当我们输入 `ThinkPHP` 后点击保存按钮

然后页面后 用户名输入框中会显示已经保存的 `session` 值。

Cookie

Cookie

ThinkPHP使用`think\Cookie`类提供Cookie支持。

初始化

大多数情况下，我们不需要进行`Cookie`的初始化，系统会在调用`Cooie`类方法的时候自动根据`cache`配置参数初始化，如果需要可以手动使用`init`方法进行初始化设置，例如：

```
// cookie初始化
Cookie::init(['prefix'=>'think_', 'expire'=>3600, 'path'=>'/']);
// 单独指定当前前缀
Cookie::prefix('think_');
```

支持的参数及默认值如下：

```
// cookie 名称前缀
'prefix'    => '',
// cookie 保存时间
'expire'    => 0,
// cookie 保存路径
'path'      => '/',
// cookie 有效域名
'domain'    => '',
//  cookie 启用安全传输
'secure'    => false,
// httponly设置
'httponly'  => '',
// 是否使用 setcookie
'setcookie' => true,
```

读取Cookie

建议的读取`Cookie`数据的方法是通过`Request`请求对象的`cookie`方法（原因和Session读取一样），例如：

```
namespace app\index\controller;

use think\Request;

class User
{
    public function index(Request $request)
    {
        echo $request->cookie('user_name');
        // 读取二维数组
        echo $request->cookie('user.name');
    }
}
```

> 通过Request对象读取Cookie数据支持默认值及过滤方法，因此也更加安全，并且支持多维数组的读取。

当然也支持使用`Cookie`类直接读取数据：

```
namespace app\index\controller;

use think\Cookie;

class User
{
    public function index()
    {
        echo Cookie::get('user_name');
    }
}
```

> Cookie::get方法不支持读取二维数组数据。

模板输出

同样，可以使用下面的方法在模板文件中输出Cookie值。

```
{$Request.cookie.user_name}
```

`Cookie` 操作

下面是一些关于Cookie的基础操作方法。

设置

```
// 设置Cookie 有效期为 3600秒
Cookie::set('name','value',3600);
// 设置cookie 前缀为think_
Cookie::set('name','value',['prefix'=>'think_', 'expire'=>3600]);
// 支持数组
Cookie::set('name',[1,2,3]);
```

判断

```
Cookie::has('name');
// 判断指定前缀的cookie值是否存在
Cookie::has('name','think_');
```

获取

```
Cookie::get('name');
// 获取指定前缀的cookie值
Cookie::get('name','think_');
```

删除

//删除cookie

```
Cookie::delete('name');
// 删除指定前缀的cookie
Cookie::delete('name','think_');
```

清空

```
// 清空指定前缀的cookie
Cookie::clear('think_');
```

> 注意，目前不支持清空所有的 `Cookie` 数据，如果必须这样做，请直接操作 `$_COOKIE` 变量。

助手函数

系统提供了 `cookie` 助手函数用于基本的 `Cookie` 操作，可以完成前面的所有功能，例如：

```
// 初始化
cookie(['prefix' => 'think_', 'expire' => 3600]);
// 设置
cookie('name', 'value', 3600);
// 判断
cookie('?name');
// 获取
echo cookie('name');
// 删除
cookie('name', null);
// 清除
cookie(null, 'think_');
```

验证码

验证码

本篇主要介绍 `ThinkPHP5` 的验证码的使用和配置。

安装类库

在使用验证码之前，必须使用 `Composer` 来安装验证码类库，在命令行下面切换到你的应用根目录下面，执行：

```
composer require topthink/think-captcha
```

执行 `composer` 安装后显示如下信息：

```
>composer require topthink/think-captcha
Using version ^1.0 for topthink/think-captcha
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
  - Installing topthink/think-captcha (v1.0.7)
    Downloading: 100%

Writing lock file
Generating autoload files
```

表示验证码类库安装完成。

验证码显示

创建一个 `Captcha` 控制器类，添加验证码显示方法：

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

class Captcha extends \think\Controller
{
```

```
    // 验证码表单
    public function index()
    {
        return $this->fetch();
    }

}
```

创建模板文件(`application/index/view/captcha/index.html`)

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>验证码示例</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>验证码示例</h2>
<FORM method="post" class="form" action="{:url('check')}">
输入验证码：<INPUT type="text" class="text" name="code"><br/>
<div>{:captcha_img()}</div>
<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>
```

访问

```
http://tp5.com/index/captcha
```

后显示页面如下：

模板中我们使用captcha_img函数显示验证码图片，细心的用户不难发现，验证码图片的URL地址其实是：

```
http://tp5.com/captcha.html
```

这是扩展类库给验证码图片自动注册了一个`captcha`的路由地址，因此请确保你已经开启URL路由。

验证码检测

下面来进行验证码检测，在控制器中添加验证方法`check`，代码如下：

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

class Captcha extends \think\Controller
{

    // 验证码表单
    public function index()
    {
        return $this->fetch();
    }

    // 验证码检测
    public function check($code='')
    {
        $captcha = new \think\captcha\Captcha();
        if (!$captcha->check($code)) {
            $this->error('验证码错误');
        } else {
            $this->success('验证码正确');
```

```
        }
    }
}
```

当我们输入一个错误的验证码后，页面提示：

如果输入正确，则显示：

扩展包内置提供了助手函数 `captcha_check` 用来检测验证码，所以 `check` 方法可以简化为：

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

class Captcha extends \think\Controller
```

```
{
    // 验证码表单
    public function index()
    {
        return $this->fetch();
    }

    // 验证码检测
    public function check($code)
    {
        if (!captcha_check($code)) {
            $this->error('验证码错误');
        } else {
            $this->success('验证码正确');
        }
    }
}
```

更简单的用法可以使用控制器的`validate`方法进行验证，例如：

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

use think\Request;

class Captcha extends \think\Controller
{
    // 验证码表单
    public function index()
    {
        return $this->fetch();
    }

    // 验证码检测
    public function check(Request $request)
    {
        if (true !== $this->validate($request->param(),[
            'code|验证码'=>'require|captcha'
        ])) {
            $this->error('验证码错误');
        } else {
            $this->success('验证码正确');
        }
    }
}
```

验证码配置

验证码的配置参数统一在应用配置文件（`application/config.php`）中添加`captcha`配置参数，例如：

```
'captcha'  => [
    // 字体大小
    'fontSize' => 35,
    // 验证码长度（位数）
    'length'   => 4,
],
```

或者配置

```
// 扩展配置文件
'extra_config_list' => ['database', 'route', 'validate', 'captcha'],
```

然后在应用目录下面添加`captcha.php`配置文件，内容如下：

```
return [
    // 字体大小
    'fontSize' => 35,
    // 验证码长度（位数）
    'length'   => 4,
];
```

支持的配置参数包括：	参数	描述
seKey	验证码加密Key	
codeSet	验证码字符集合	
expire	验证码过期时间（秒）	
useZh	使用中文验证码	
useImgBg	使用背景图片	
fontSize	字体大小（px）	
useCurve	是否添加混淆曲线	
useNoise	是否添加杂点	
imageH	验证码图片高度	
imageW	验证码图片宽度	
length	验证码长度	
fontttf	验证码字体	
bg	验证码图片背景色	
reset	验证成功后是否重置	

下面举例进行一些说明：

改变字体大小和验证码长度

```
'captcha'  => [
    // 字体大小
    'fontSize' => 35,
    // 验证码长度（位数）
    'length'   => 4,
],
```

中文验证码

```
'captcha'  => [
    // 使用中文验证码
    'useZh' =>true,
    // 字体大小
    'fontSize' => 35,
    // 验证码长度（位数）
    'length'   => 4,
],
```

添加背景图片

```
'captcha'  => [
    // 使用背景图片
    'useImgBg' => true
    // 验证码长度（位数）
    'length'   => 4,
],
```

添加验证码资源

如果需要添加额外的验证码字体、背景图等，可以在 `vendor/topthink/think-captcha/assets/` 下面自己扩展

目录	作用
bgs	背景图
ttfs	英文字体
zhttfs	中文字体

同时使用多个验证码

如果页面上需要使用多个验证码验证不同的提交，可以采用下面的方式。

首先，修改模板文件显示两个验证码，如下：

```
<h2>多个验证码示例</h2>
<FORM method="post" class="form" action="{:url('check1')}">
验证码1：<INPUT type="text" class="text" name="code"><br/>
<div>{:captcha_img(1)}</div>
<INPUT type="submit" class="btn" value=" 提交 ">
</FORM>

<FORM method="post" class="form" action="{:url('check2')}">
验证码2：<INPUT type="text" class="text" name="code"><br/>
<div>{:captcha_img(2)}</div>
<INPUT type="submit" class="btn" value=" 提交 ">
</FORM>
```

注意，最关键的地方在于调用 `captcha_img` 函数显示验证码的时候，我们传入了额外的参数以标识不同的验证码。

使用多个验证码的时候，验证码图片的资源地址变成：

```
http://tp5.com/captcha/1.html // 验证码1
http://tp5.com/captcha/2.html // 验证码2
```

控制器类的代码进行如下修改：

```
<?php
```

```
namespace app\index\controller;

class Captcha extends \think\Controller
{

    // 验证码表单
    public function index()
    {
        return $this->fetch();
    }

    // 验证码1检测
    public function check1($code = '')
    {
        if (!captcha_check($code, 1)) {
            $this->error('验证码1错误');
        } else {
            $this->success('验证码1正确');
        }

    }

    // 验证码2检测
    public function check2($code = '')
    {
        if (!captcha_check($code, 2)) {
            $this->error('验证码2错误');
        } else {
            $this->success('验证码2正确');
        }
    }
}
```

文件上传

文件上传

本篇主要介绍如何使用 `ThinkPHP5` 进行文件上传及验证。

- 控制器定义
- 上传文件验证
- 文件保存规则
- 多文件上传
- 后续文件操作

控制器定义

> 文件上传使用ThinkPHP5内置的 `think\File` 类库，该类库可以轻松实现文件上传到本地服务器，如果需要上传到其它服务器或者平台，则需要后续调用其它类库或者接口。

首先来创建一个Upload控制器如下：

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

use think\Request;

class Upload extends \think\Controller
{

    // 文件上传表单
    public function index()
    {
        return $this->fetch();
    }

    // 文件上传提交
    public function up(Request $request)
    {
        // 获取表单上传文件
        $file = $request->file('file');
        if (empty($file)) {
            $this->error('请选择上传文件');
        }
        // 移动到框架应用根目录/public/uploads/ 目录下
        $info = $file->move(ROOT_PATH . 'public' . DS . 'uploads');
        if ($info) {
            $this->success('文件上传成功：' . $info->getRealPath());
        } else {
            // 上传失败获取错误信息
            $this->error($file->getError());
        }

    }

}
```

然后创建模板文件（`application/index/view/upload/index.html`）：

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>文件上传示例</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;
}
.form .file{
    margin:6px;
    padding: 6px;
    width: 220px;

    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>文件上传示例</h2>
<FORM method="post" enctype="multipart/form-data" class="form" action="{:url('up')}">
选择文件：<INPUT type="file" class="file" name="file"><br/>
<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>
```

访问下面的URL地址：

```
http://tp5.com/index/upload
```

如果没有选择任何文件点击提交的话，会显示：

选择文件后点击提交：

ThinkPHP V5{ 十年磨一剑-为API开发设计的高性能框架 }

:)

文件上传成功：

D:\www\tp\public\uploads\20160717\dea7867caa4f59caa1fa98e38ef52267.png

页面自动 跳转 等待时间： 3

默认的上传文件会按照当前的日期自动保存

上传文件验证

可以在上传之前调用`validate`方法设置验证规则，例如：

```
    // 文件上传提交
    public function up(Request $request)
    {
        // 获取表单上传文件
        $file = $request->file('file');
        if (empty($file)) {
            $this->error('请选择上传文件');
        }
        // 移动到框架应用根目录/public/uploads/ 目录下
        $info = $file->validate(['ext' => 'jpg,png'])->move(ROOT_PATH . 'public' . DS .
'uploads');
        if ($info) {
            $this->success('文件上传成功：' . $info->getRealPath());
        } else {
            // 上传失败获取错误信息
            $this->error($file->getError());
        }

    }
```

`validate`方法设置了允许上传的文件后缀是jpg和png，当我们选择一个gif格式的文件后，会提示：

`validate`方法支持的验证规则包括：	验证规则	说明	参数类型
size	上传文件最大字节大小	integer	
ext	允许上传的文件后缀	数组或者字符串	
type	允许上传的文件类型	数组或者字符串	

如果上传的文件后缀是图像格式的话，系统会自动检测是否为合法的图像文件，例如，我们创建一个hello.jpg

文件，并且写入下列文本内容

```
Hello, ThinkPHP5！
```

当我们上传该文件并且提交后显示：

> 注意：文件上传验证用到了 exif 扩展，如果没有开启请自行开启。

如果你统一使用验证类 think\Validate 对表单进行验证，也可以直接使用控制器类 think\Controller 的 validate 方法进行上传文件验证，修改控制器up方法的代码为：

```
    // 文件上传提交
    public function up(Request $request)
    {
        // 获取表单上传文件
        $file = $request->file('file');
        // 上传文件验证
        $result = $this->validate(['file' => $file], ['file'=>'require|image'],['file.r
equire' => '请选择上传文件', 'file.image' => '非法图像文件']);
        if(true !== $result){
            $this->error($result);
        }
        // 移动到框架应用根目录/public/uploads/ 目录下
        $info = $file->move(ROOT_PATH . 'public' . DS . 'uploads');
        if ($info) {
            $this->success('文件上传成功：' . $info->getRealPath());
        } else {
            // 上传失败获取错误信息
            $this->error($file->getError());
        }

    }
```

关键的代码是这行：

```
// 上传文件验证
$result = $this->validate(['file' => $file], ['file'=>'require|image'],['file.require'
=> '请选择上传文件', 'file.image' => '非法图像文件']);
```

由于设置了`require`验证规则（表示必须上传文件），所以如果没有选择任何文件直接提交的话，页面会提示：

`image`验证规则表示上传的必须是一个图像文件，如果选择上传了一个文本文件，页面会提示：

如果需要上传一个100*100的`PNG`图像文件，可以这样进行验证：

```
// 上传文件验证
```

```
$result = $this->validate(['file' => $file], ['file'=>'require|image:100,100,png'],['fi
le.require' => '请选择上传文件', 'file.image' => '必须是100*100的PNG格式文件']);
```

更多的上传文件验证规则还包括：	验证规则	说明
file	验证是否为File对象	
image	验证是否为图像File对象	
image:width,height[,type]	验证图像文件的类型和宽高	
fileExt:zip,doc,...	验证文件后缀	
fileMime:image/png,...	验证文件类型	
fileSize:1024	验证文件大小	

文件保存规则

可以在上传之前调用 `rule` 方法设置上传文件的保存规则。

```
    // 文件上传提交
    public function up(Request $request)
    {
        // 获取表单上传文件
        $file = $request->file('file');
        if (empty($file)) {
            $this->error('请选择上传文件');
        }
        // 移动到框架应用根目录/public/uploads/ 目录下
        $info = $file->rule('md5')->move(ROOT_PATH . 'public' . DS . 'uploads');
        if ($info) {
            $this->success('文件上传成功：' . $info->getRealPath());
        } else {
            // 上传失败获取错误信息
            $this->error($file->getError());
        }

    }
```

:)

文件上传成功：

D:\www\tp\public\uploads\23\1766b13f370ee58d33fb0cc961e3d1.jpg

页面自动 跳转 等待时间： **3**

系统默认提供了几种上传命名规则，包括：	规则	描述
date	根据日期和微秒数生成（默认规则）	
md5	对文件使用md5_file散列生成	
sha1	对文件使用sha1_file散列生成	

> 其中md5和sha1规则会自动以散列值的前两个字符作为子目录，后面的散列值作为文件名。

如果需要自定义上传文件的保存规则，可以使用：

```
// 移动到框架应用根目录/public/uploads/ 目录下
$info = $file->rule('uniqid')->move(ROOT_PATH . 'public' . DS . 'uploads');
```

这里使用了`uniqid`函数来生成唯一的ID命名上传文件，如果必要，还可以支持使用闭包定义规则，例如：

```
// 移动到框架应用根目录/public/uploads/ 目录下
$info = $file->rule(function ($file) {
    // 使用自定义的文件保存规则
    return $file->getInfo('type') . uniqid();
})->move(ROOT_PATH . 'public' . DS . 'uploads');
```

文件上传成功：D:\www\tp\public\uploads\image\jpeg578b843d4baa0.jpg

页面自动 跳转 等待时间： **3**

如果希望指定某个文件的保存文件名，还可以直接使用：

```
// 移动到框架应用根目录/public/uploads/ 目录下
$info = $file->move(ROOT_PATH . 'public' . DS . 'uploads','test');
```

文件上传成功：D:\www\tp\public\uploads\test.png

页面自动 跳转 等待时间： **3**

如果希望保持上传文件的原文件名保存，则可以使用

```
// 移动到框架应用根目录/public/uploads/ 目录下
$info = $file->move(ROOT_PATH . 'public' . DS . 'uploads','');
```

默认情况下，会覆盖服务器上传目录下的同名文件，如果不希望覆盖，可以使用：

```
// 移动到服务器的上传目录 并且设置不覆盖
```

```
$file->move(ROOT_PATH . 'public' . DS . 'uploads',true,false);
```

如果要获取上传的保存文件名，可以调用返回对象的`getSaveName`方法：

```
$info = $file->rule('md5')->move(ROOT_PATH . 'public' . DS . 'uploads');
if ($info) {
    echo $info->getSaveName();
} else {
    $this->error($file->getError());
}
```

多文件上传

如果你使用的是多文件上传表单，例如：

```
<FORM method="post" enctype="multipart/form-data" class="form" action="{:url('up')}">
<input type="file" name="image[]" /> <br>
<input type="file" name="image[]" /> <br>
<input type="file" name="image[]" /> <br>
<INPUT type="submit" class="btn" value=" 提交 ">
</FORM>
```

控制器代码可以改成：

```
public function up(Request $request){
    // 获取表单上传文件
    $files = $request->file('image');
    $item  = [];
    foreach($files as $file){
        // 移动到框架应用根目录/public/uploads/ 目录下
        $info = $file->move(ROOT_PATH . 'public' . DS . 'uploads');
        if($info){
            $item[] = $info->getRealPath();
        }else{
            // 上传失败获取错误信息
            $this->error($file->getError());
        }
    }
    $this->success('文件上传成功'.implode('<br/>',$item));
}
```

后续文件操作

上传成功后返回的是`File`对象，除了可以使用`SplFileObject`的属性和方法之外，还可以使用File类自身提供的下列方法，便于进行后续的文件处理（例如对图像文件进行剪裁处理或者移动到远程服务器）。

方法	描述
getSaveName	获取保存的文件名（包含动态生成的目录）
getInfo	获取上传文件信息
getMime	获取文件的MIME信息
md5	获取文件的md5散列值

sha1	获取文件的sha1散列值

下面是一个简单的FTP文件移动的处理例子。

```
    public function up(Request $request){
        // 获取表单上传文件
        $file = $request->file('file');
        // 上传文件
        $info = $file->move(ROOT_PATH . 'public' . DS . 'uploads');
        if ($info) {
            // 移动文件到FTP服务器
            $link = ftp_connect('212.45.5.78');
            ftp_login($link, 'root', 'password');
            /* 移动文件 */
            $path = ftp_pwd($link) . '/uploads/';
            if (!ftp_put($link, $path.$info->getFilename(), $info->getRealPath(), FTP_B
INARY)) {
                $this->error('文件上传保存错误！');
            }
        } else {
            // 上传失败获取错误信息
            $this->error($file->getError());
        }
    }
```

还可以对上传的文件进行处理，后面我们还会讲到对上传图像文件进行额外的图像处理。

图像处理

图像处理

我们继续前面的文件上传部分，并调用图像扩展类库对上传的图像文件进行相关的处理和保存。

安装扩展

首先使用`Composer`安装`ThinkPHP5`的图像处理类库：

```
composer require topthink/think-image
```

示例代码

本示例使用下面的示例代码进行图片上传和处理。

控制器定义

然后修改之前创建的`Upload`控制器，添加`picture`方法如下：

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

use think\Image;
use think\Request;

class Upload extends \think\Controller
{

    // 文件上传表单
    public function index()
    {
        return $this->fetch();
    }
```

```
    // 图片上传处理
    public function picture(Request $request)
    {
        // 获取表单上传文件
        $file = $request->file('image');
        if (true !== $this->validate(['image' => $file], ['image' => 'require|image']))
 {
            $this->error('请选择图像文件');
        } else {
            // 读取图片
            $image = Image::open($file);
            // 图片处理
            switch ($request->param('type')) {
                case 1: // 图片裁剪
                    $image->crop(300, 300);
                    break;
                case 2: // 缩略图
                    $image->thumb(150, 150, Image::THUMB_CENTER);
                    break;
                case 3: // 垂直翻转
                    $image->flip();
                    break;
                case 4: // 水平翻转
                    $image->flip(Image::FLIP_Y);
                    break;
                case 5: // 图片旋转
                    $image->rotate();
                    break;
                case 6: // 图片水印
                    $image->water('./logo.png', Image::WATER_NORTHWEST, 50);
                    break;
                case 7: // 文字水印
                    $image->text('ThinkPHP', VENDOR_PATH . 'topthink/think-captcha/asse
ts/ttfs/1.ttf', 20, '#ffffff');
                    break;
            }
            // 保存图片（以当前时间戳）
            $saveName = $request->time() . '.png';
            $image->save(ROOT_PATH . 'public/uploads/' . $saveName);
            $this->success('图片处理完毕...', '/uploads/' . $saveName, 1);
        }
    }

}
```

模板定义

配合控制器的上传页面模板修改为：

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>图像上传和处理示例</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 select {
    padding: 5px;
    margin:2px 10px;
    width: 150px;
    height: 30px;
    line-height: 30px;
    border: 1px solid #D4D4D4;
}

.form .btn{
    margin:6px;
    padding: 6px;
    width: 120px;

    font-size: 16px;
    border: 1px solid #D4D4D4;
    cursor: pointer;
    background:#eee;
}
.form .file{
    margin:6px;
    padding: 6px;
    width: 220px;

    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>图像上传和处理示例</h2>
<form method="post" enctype="multipart/form-data" class="form" action="{:url('picture')
}">
选择图像文件：<input type="file" class="file" name="image"><br/>
选择处理类型：<select name="type">
    <option value="1" selected>图片裁剪
    <option value="2">生成缩略图
    <option value="3">垂直翻转
    <option value="4">水平翻转
    <option value="5">图片旋转
    <option value="6">添加图片水印
    <option value="7">添加文字水印
</select><br/>
<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>
```

示例演示

访问下面的URL地址：

```
http://tp5.com/index/upload
```

页面显示：

这里仅能上传`png`和`jpg`类型的图像文件，没有选择或者选择其他类型的文件都会提示：

本节示例选择上传的图片文件为：

点击上传并提交后，页面显示：

之后会显示处理之后的图片：

下面来具体讲下各种图片处理操作的用法。

读取图片

`think\Image`类提供了`open`方法打开图片文件

```
// 获取上传文件
$file = $request->file('image');
// 读取图片文件
$image = Image::open($file);
```

这里的`$file`是一个`\think\File`对象（其实可以支持任何`SplFileInfo`或者子对象），也可以直接传入一个图片文件地址：

```
// 读取图片文件
$image = Image::open('./logo.png');
```

> `open`方法不支持读取远程图片

使用open方法读取图片文件后，可以读取图片相关信息：

```
// 返回图片的宽度
$width = $image->width();
// 返回图片的高度
$height = $image->height();
// 返回图片的类型
$type = $image->type();
// 返回图片的mime类型
$mime = $image->mime();
// 返回图片的尺寸数组 [ 图片宽度 , 图片高度 ]
$size = $image->size();
```

图片裁剪

图片剪裁使用`crop`方法，用法：

> **crop(剪裁宽度,剪裁高度,X坐标（默认0）,Y坐标（默认0））**

示例中图片剪裁的主要代码为

```
$image->crop(300, 300);
```

表示从左上角开始剪裁宽高都是300的图片，如果需要改变剪裁的坐标位置，可以用：

```
$image->crop(300, 300, 100, 50);
```

最终的效果变成：

生成缩略图

生成图片缩略图使用`thumb`方法，用法：

> **thumb(最大宽度,最大高度,裁剪类型)**

缩略图剪裁类型包括如下：	剪裁类型（常量=值）	描述
Image::THUMB_SCALING = 1	等比例缩放（默认类型）	
Image::THUMB_FILLED = 2	缩放后填充	
Image::THUMB_CENTER = 3	居中裁剪	
Image::THUMB_NORTHWEST = 4	左上角裁剪	
Image::THUMB_SOUTHEAST = 5	右下角裁剪	
Image::THUMB_FIXED = 6	固定尺寸缩放	

示例代码中的缩略图效果如图：

图像翻转

图片翻转使用`flip`方法，用法如下：

flip(翻转方式)

翻转方式	常量值
垂直翻转	Image::FLIP_X=1
水平翻转	Image::FLIP_Y=2

垂直翻转效果：

水平翻转效果：

图片旋转

图片翻转使用 `rotate` 方法，用法如下：

> **rotate(顺时针旋转的度数)**

示例采用90旋转的效果如图：

添加水印

使用`water`方法添加图片水印

> **water(水印图片,水印位置常量（默认右下角）,水印透明度（默认100））**

水印位置常量如下：

水印位置	常量值
左上角	Image::WATER_NORTHWEST=1
上居中	Image::WATER_NORTH=2

右上角	Image::WATER_NORTHEAST=3
左居中	Image::WATER_WEST=4
居中	Image::WATER_CENTER=5
右居中	Image::WATER_EAST=6
左下角	Image::WATER_SOUTHWEST=7
下居中	Image::WATER_SOUTH=8
右下角	Image::WATER_SOUTHEAST=9

示例代码：

```
$image->water('./logo.png', Image::WATER_NORTHWEST, 50);
```

示例水印效果如图：

> 为了示例效果，在操作之前，请把下面的logo图片保存到public目录下面。
>
> 

文字水印

使用`text`方法给图片添加文字（水印）

> **text(水印文字,字体文件路径,文字大小,文字颜色,文字写入位置,偏移量,文字倾斜角度)**

前面四个参数必须，颜色默认为 #000000 文字位置常量和水印位置一样。

示例代码

```
$image->text('ThinkPHP', VENDOR_PATH . 'topthink/think-captcha/assets/ttfs/1.ttf', 20,
'#ffffff');
```

> 为了配合演示，我们使用了验证码类库中的字体素材

图片保存

前面所有的操作都是对图片进行相关的处理，最后一步就是需要把处理过的图片文件保存下来。

这就需要调用`save`方法进行图片的保存操作

> **save(保存文件名,图像类型,图像质量,隔行扫描)**

示例中使用当前时间戳作为文件名保存图片文件，代码：

```
// 保存图片（以当前时间戳）
$saveName = $request->time() . '.png';
$image->save(ROOT_PATH . 'public/uploads/' . $saveName);
```

默认图片保存的质量是`80`，如果希望采用最高的质量保存，可以使用下面的代码：

```
// 保存图片（以当前时间戳）
$saveName = $request->time() . '.png';
// 采用最高质量保存图片
$image->save(ROOT_PATH . 'public/uploads/' . $saveName,'png',100);
```

> save方法的第四个参数仅针对jpg格式的图像类型。

单元测试

单元测试

> 单元测试是对单独的代码对象进行测试的过程，比如对函数、类、方法进行测试。单元测试可以使用任意一段已经写好的测试代码，也可以使用一些已经存在的测试框架，比如`PHPUnit`、`Phake`或者`SimpleTest`，单元测试框架提供了一系列共同、有用的功能来帮助人们编写自动化的检测单元，例如检查一个实际的值是否符合我们期望的值的断言。单元测试框架经常会包含每个测试的报告，以及给出你已经覆盖到的代码覆盖率。

单元测试对于团队开发而言，好处不言而喻，主要作用包括：

- 尽可能减少开发中的BUG；
- 帮助提高应用（或者接口）设计；
- 协助代码文档的编写；
- 减少开发过程的代码修改带来的错误；

`ThinkPHP5.0`目前支持`PHPUnit`进行单元测试，这是最常用的单元测试工具，ThinkPHP单元测试扩展还针对控制器和模型的单元测试做了完善支持，添加了额外的单元测试方法和断言。

`ThinkPHP5.0`核心也是采用`PHPUnit`作为单元测试工具，本文主要讲述如何对应用进行`PHPUnit`单元测试，但不打算详细介绍单元测试的用法，如果你还不了解`PHPUnit`以及什么是单元测试，我们建议首先阅读了解下PHPUnit手册或者相关书籍。

安装扩展

首先安装`ThinkPHP5`的单元测试扩展，进入命令行，切换到应用根目录下面后，执行：

```
composer require topthink/think-testing
```

你不需要单独安装`PHPUnit`包，安装单元测试扩展的时候会自动安装相关的依赖包，其中就包括`PHPUnit`。

> 由于单元测试扩展的依赖较多，因此安装过程会比较久，请耐心等待。

安装完成后，会在应用根目录下面增加`tests`目录和`phpunit.xml`文件。

`tests`目录是应用的单元测试用例目录，该目录下面的`TestCase.php`文件是所有单元测试类必须基础的基础类，**不能删除**，`ExampleTest.php`是测试示例文件，可以删除。

运行测试

由于单元测试扩展默认自带了一个`tests/ExampleTest.php`单元测试文件，内容如下：

```
namespace tests;

class ExampleTest extends TestCase
{

    public function testBasicExample()
    {
        $this->visit('/')->see('ThinkPHP');
    }
}
```

表示测试访问网站首页的响应输出内容中是否包含`ThinkPHP`内容，因为我们的默认。

因此我们可以直接在命令行下面运行单元测试：

```
php think unit
```

如果你是第一次安装ThinkPHP5的话，或者没有改动过Index模块Index控制器的index方法的话，测试是OK的，会显示类似下面的结果：

```
PHPUnit 4.8.27 by Sebastian Bergmann and contributors.

.

Time: 221 ms, Memory: 6.00MB
```

如果改过index方法，并且页面输出内容没有ThinkPHP的话，测试就会失败。

```
PHPUnit 4.8.27 by Sebastian Bergmann and contributors.

F

Time: 230 ms, Memory: 6.00MB

There was 1 failure:

1) tests\ExampleTest::testBasicExample
Failed asserting that 'Hello,world' matches PCRE pattern "/ThinkPHP/i".

D:\www\tp5\vendor\topthink\think-testing\src\InteractsWithPages.php:67
D:\www\tp5\tests\ExampleTest.php:18
D:\www\tp5\vendor\topthink\think-testing\src\command\Test.php:39
D:\www\tp5\thinkphp\library\think\console\command\Command.php:195
D:\www\tp5\thinkphp\library\think\Console.php:728
D:\www\tp5\thinkphp\library\think\Console.php:188
D:\www\tp5\thinkphp\library\think\Console.php:125
D:\www\tp5\thinkphp\library\think\Console.php:94
D:\www\tp5\thinkphp\console.php:20

FAILURES!
Tests: 1, Assertions: 2, Failures: 1.
```

> 请始终使用以上命令进行单元测试，而不是直接用 `phpunit` 来运行单元测试。

添加单元测试用例

我们来添加一个控制器类的单元测试文件，为了便于管理，我们按照模块名创建单元测试的子目录和应用类库的目录结构对应起来，该单元测试文件为 `tests/index/IndexTest.php` ，内容如下：

```
<?php
namespace tests\index;
use tests\TestCase;

class IndexTest extends TestCase
{

    public function setUp()
    {
        parent::setUp();
    }

    public function testSome()
    {
        $this->assertTrue(true);
    }

    public function tearDown()
    {
    }

}
```

> 每个单元测试类必须继承 `TestCase` 类或其子类，每个单元测试方法必须以 `test` 开头。

除了单元测试方法之外，还可以增加一些特殊的单元测试定义方法（但不是必须的）。

`setUp` 方法会在每一个单元测试方法之前自动调用，所以该方法可以用于一些测试的初始化，也就是所说的基境（ `fixture` ），对应还有一个 `tearDown` 方法会在每个测试方法完成后自动调用，通常用于重置一些测试的数据。

每个单元测试方法必须以 `test` 开头，后面的测试方法名称的命名可以随意，例如上面的 `testSome` 方法改成 `testSomethingToTrue` 并不会影响测试结果。

如果你需要同时对控制器类和模型类做单元测试的话，也可以在模块目录下面创建 `controller` 和 `model` 子目录，然后分别创建单元测试文件。

控制器单元测试文件 `tests/index/controller/IndexTest.php`

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

use tests\TestCase;
```

```
class IndexTest extends TestCase
{

    public function testIndex()
    {
        $this->call('GET', '/');
        $this->assertResponseOk();
    }

}
```

测试用例对网站首页进行了一次GET请求，并判断是否返回正确的响应。

模型单元测试文件`tests/index/model/IndexTest.php`

```
<?php
namespace tests\index\model;

use tests\TestCase;

class IndexTest extends TestCase
{

    public function testSearch()
    {
        $this->seeInDatabase('user', ['id' => 2]);
    }

}
```

> 在测试模型用例之前，请首先确保正确配置了数据库配置文件。

该测试用例测试数据表`think_user`（假设你的数据表前缀设置为`think_`）是否存在`id`为2的数据。

其实模型的测试用例和控制器的测试用例没有明确的界限，大多数情况，我们只需要针对控制器类做单元测试即可，同样，你仍然可以对不同的应用层类库做单元测试。

除了支持`PHPUnit`自带的测试方法外，`ThinkPHP`单元测试扩展还封装了一些额外的方法，我们以后会给你详细描述。

定义单元测试

单元测试最大的工作是使用断言（`assertion`），所以你会发现PHPUnit自身带了很多的断言方法，常用的方法包括：

断言方法	描述
assertTrue($var)	断言变量为true
assertFalse($var)	断言变量为false
assertEquals($value,$var)	断言变量为$value
assertNull($var)	断言变量为null

assertContains($needle,$var)	断言变量中包含$needle

附录

A、常见问题集

ThinkPHP5.0 常见问题集

整理了下ThinkPHP `5.0` 的常见问题（不断更新中~），希望给新手和习惯了 `3.2` 版本的用户带来帮助吧，类似的问题在V5交流群里面一天要问个百八次的 ^_^

为什么5.0取消了很多常量？

核心使用太多的常量不利于单元测试以及今后的组件化设计，目前 `5.0` 版本仅仅保留了核心的路径常量，其它常量均改为通过 `Request` 请求对象的方法获取（有个别常量则废除）。

如果觉得不习惯或者项目迁移需要，也可以在应用里面自己重新定义这些常量。

为啥单字母函数去掉了？

单字母函数是TP历史上争议较大的问题，应该说单字母函数的诞生是有历史原因的，但已经到了退出历史舞台的时候了，可能很多人习惯了单字母函数带来的便捷（虽然同时我们也饱受着某些学院用户的抱怨）。

基于几个原因废除了单字母函数：
（1）5.0 核心已经不依赖任何函数 只是对常用的操作封装提供了助手函数
（2）助手函数是可以完全自己定义和修改，并不影响其他的使用
（3）现在的IDE提示和自动完成功能已经非常强大了，所以用不用助手函数，或者是否需要改成原来的单字母函数，全凭开发者个人意愿。

核心框架不依赖任何助手函数，系统只是默认加载了助手函数，配置如下：

```
// 扩展函数文件定义
'extra_file_list'   => [THINK_PATH . 'helper' . EXT],
```

因此，你可以随意修改助手函数的名称或者添加自己的助手函数，然后修改配置为：

```
// 扩展函数文件定义
'extra_file_list'   =>  [ APP_PATH . 'helper' . EXT],
```

数据库查询和模型怎么用？

官方开发手册的所有示例代码为了简洁明了，省略了所有 think 命名空间的类调用，也就是说 如果示例里面的代码是

```
Db::table('think_user')->select();
```

那么其实正确的姿势应该是

```
\think\Db::table('think_user')->select();
```

或者首先

```
use think\Db;
```

然后

```
Db::table('think_user')->select();
```

这里，我首先要说声抱歉，3.2的函数封装方式某种意义上掩盖了命名空间的调用问题，让大家感觉不到命名空间的存在意义和调用机制，傻瓜式地在使用，如果5.0还不能让你们清醒认识并掌握到命名空间应该是怎么调用的话，那么我奉劝你可以退出开发者的行列了。

下面来说说关于数据查询的问题，原来在模型里面的数据查询，现在Db类都能完成，新的模型类的查询返回的都是模型对象，例如：

```
$user = User::get(1);
$user = User::find();
```

模型的查询返回的都是模型对象实例，每个模型拥有自己的属性和方法。

关于配置参数的问题

很多的配置参数或者配置层次都和之前不同了，有疑问的时候可以直接看框架目录下面的`convention.php`文件，或者完全开发手册中的附录：配置参数。都列出了完整的配置参数。

怎么才能在控制器中正确的输出模板

5.0在控制器中输出模板，使用方法如下：

如果你的控制器继承了`think\Controller`的话，使用：

```
return $this->fetch('index/hello');
```

如果你没有继承`think\Controller`的话，可以使用：

```
return view('index/hello');
```

没有创建模型类怎么查询？

原先版本的M方法可以支持不用创建类进行查询，新版如何处理呢？

首先，我们要搞明白一个，查询是`Db`类的事情，如果只是查询数据表的数据到一个数组，那么直接使用

```
Db::table('think_user')->select();
// 或者
Db::name('user')->select();
```

就可以了，如果你要返回一个模型对象（为啥要对象 这个可能目前你还不一定能理解），则使用下面的方式：

创建一个`User`模型类 然后：

```
$user = User::get(1);
```

或者使用：

```
$user = new User;
$data = $user->where('id',1)->find();
```

原来3.2版本的模型的 getField 方法应该如何使用？

原先模型类的 `getField` 方法，在5.0的数据库 `Query` 类中拆分成两个方法，一个 `value` 方法用于查询某个行的某个值和一个 `column` 方法用于查询某个列的值，用法如下：

查询某个字段的值可以用

```
// 返回某个字段的值
Db::table('think_user')->where('id',1)->value('name');
```

原先的聚合查询方法依然有效，例如：

```
// 查询用户数
Db::table('think_user')->count();
// 查询用户的最高分
Db::table('think_user')->max('score');
```

查询某一列的值可以用

```
// 返回数组
Db::table('think_user')->where('status',1)->column('name');
// 指定id字段作为索引
Db::table('think_user')->where('status',1)->column('name','id');
```

5.0怎么没有编译缓存了？

> 编译缓存的基础原理是第一次运行的时候把核心需要加载的文件去掉空白和注释后合并到一个文件中，第二次运行的时候就直接载入编译缓存而无需载入众多的核心文件。

编译缓存是 `ThinkPHP` 从诞生开始就存在的特色功能，但同时也是困扰很多新手的问题，经常由于不记得清空编译缓存而导致一些莫名其妙的问题。 `5.0` 开始，由于架构的重新设计框架性能已经获得了很大的提升，加上 `PHP` 本身的 `opcode` 的优化，尤其是即将迎来 `PHP7` 的时代，所以决定废除编译缓存机制。

为什么写入数据的时候不会自动去除数据库没有的字段？

ThinkPHP一直以来都支持自动去除数据库没有的字段， `5.0` 版本出于更加严谨的考虑，默认情况下当写入数据库不存在的字段的时候会抛出字段不存在的异常，有两种方法可以避免异常。

第一种，在数据库配置文件或者传入connect方法的连接参数中设置关闭严格检查：

```
// 关闭严格检查字段是否存在
'fields_strict'  => false,
```

这种方式全局有效。

第二种，在查询的时候使用strict方法指定是否进行严格检测字段

```
// 使用strict方法关闭字段严格检测
Db::name('user')->strict(false)->insert($data);
```

该方法仅针对当前查询有效。

为什么不能识别驼峰法命名的控制器

5.0版本默认情况下不区分URL的大小写，也就是说URL里面的控制器和操作都会强制转小写然后去定位控制器类，因此

```
http://serverName/index/UserType/addType
// 和下面的访问是等效的
http://serverName/index/usertype/addtype
```

最终访问的其实都是`Usertype`控制器类的`addtype`方法。

如果需要访问驼峰法命名的`UserType`控制器，有两种方式：
一、使用下面的URL地址访问

```
http://serverName/index/user_type/addtype
```

二、配置`url_convert`参数，关闭URL强制转换

```
// 关闭自动转换
'url_convert'   =>  false,
```

一旦关闭URL自动转换后，URL访问就区分大小写了
要访问`UserType`控制器类的操作，必须使用：

```
http://serverName/index/UserType/addType
```

下面的访问地址都是无效的：

```
http://serverName/index/Usertype/addtype
http://serverName/index/usertype/addtype
```

但下面的访问地址仍然有效：

```
http://serverName/index/user_type/addtype
```

`5.0`的类库定义可以带上后缀么？

因为有了命名空间，默认配置下，5.0的类库都不再和3.2版本一样加上`Controller`、`Model`之类的后缀了，如果仍然希望采用这种方式命名类库的话，两种方式可以配置：

一、单独配置控制器类的后缀

```
// 控制器类后缀
'controller_suffix'      => true,
```

这样所有的控制器类定义必须加上`Controller`后缀。

二、配置所有类库的后缀

```
// 开启应用类库后缀
'class_suffix'           => true,
```

开启以后，所有的应用类库都需要加上对应的后缀，包括控制器类、模型类及验证器类等等，以`User`控制器/模型/验证器类为例就是：

```
UserController
UserModel
UserValidate
```

> 如果使用助手函数或者Loader类的方法实例化的话，不需要加上后缀，会自动识别当前的配置参数决定是否需要添加后缀。

`5.0`怎么实现模板输出的特殊字符串替换？

`5.0`版本仍然支持模板及输出的特殊字符串替换功能，只是默认不再内置任何的替换规则，需要替换的时候，可以使用`View`类的`replace`方法或者全局配置`view_replace_str`参数。

```
// 视图输出字符串内容替换
'view_replace_str'       => [
    '__ROOT__'      =>  '',
    '__PUBLIC__'    =>  '/public',
],
```

5.0的fetch和display方法有什么区别？

5.0模板渲染提供了`fetch`和`display`两个方法，最常用的是`fetch`方法用于渲染模板文件输出，而`display`方法则是渲染内容输出。

```
// 渲染模板输出
$this->fetch($template);
// 渲染内容输出
$this->display($content);
```

两个方法的共同点是都支持模板或者内容的标签解析。

为什么Session无法获取？

`5.0` 默认并没有初始化和启动 `Session`，而是按需调用，只有在调用 `Session` 类的时候才会进行初始化，但如果你直接操作 `$_SESSION` 数组的话，则需要自己手动初始化或者启动 `session`，方法如下：

```
// 进行session初始化
Session::init();
// 或者直接调用
session_start();
```

为什么无法接收表单数组类型数据？

如果你使用框架的 `Request` 类或者 `input` 助手函数获取表单数据的话，需要使用变量修饰符 `/a` 才能正确获取数组类型的数据，例如：

```
input('get.data/a');
Request::instance()->get('data/a');
```

否则，默认会当作字符串接收而报错。

但是如果获取完整数据则不需要加上 `/a` 修饰符，例如：

```
input('get./a');
Request::instance()->get();
```

为什么模型的查询返回的都是对象而不是数组

5.0的设计就是 `Db` 类查询返回数组，模型类查询则返回当前模型的对象实例。模型采用的是对象化设计，查询的结果是一个对象，你可以进行一些额外的操作，比如调用模型的属性（字段）、业务方法，同时该模型对象仍然可以当成数组一样操作，例如：

```
$user = User::get(10);
echo $user->email;
echo $user->name;
$user->email = 'thinkphp@qq.com';
$user->save();
```

或者使用数组方式操作

```
$user = User::get(10);
echo $user['email'];
echo $user['name'];
$user['email'] = 'thinkphp@qq.com';
$user->save();
```

为什么 `Input` 类没有了？

为了保持统一的入口，5.0的 `Input` 类的相关方法已经并入 `Request` 类，`Request` 类作为当前的请求对象统一接收和处理，并且需要注意，不要在程序里面手动操作 `$_GET` 、 `$_POST` 和 `$_REQUEST` 等全局变

量，这样会导致和Request请求对象的获取不一致。
具体使用请参考第七章 请求和响应，或者查看官方的开发手册。

原来的模板标签怎么不能用了？

5.0版本的模板标签界定符由原来的 `<>` 更改为 `{}` ，主要是为了解决很多IDE无法解析导致提示错误的问题，例如：

原来的写法

```
<volist name="list" id="vo">
{$vo.id}:{$vo.name}
</volist>
```

需要改为

```
{volist name="list" id="vo"}
{$vo.id}:{$vo.name}
{/volist}
```

或者也可以直接修改配置更改标签界定符：

```
'template'               => [
    // 模板引擎类型 支持 php think 支持扩展
    'type'           => 'Think',
    // 模板引擎普通标签开始标记
    'tpl_begin'      => '{',
    // 模板引擎普通标签结束标记
    'tpl_end'        => '}',
    // 标签库标签开始标记
    'taglib_begin' => '<',
    // 标签库标签结束标记
    'taglib_end'     => '>',
],
```

5.0模型的 `save` 方法如何过滤非数据表字段值？

3.2模型类的 `create` 是一个很神奇的方法，实现了数据字段自动过滤，自动验证和自动完成，5.0里面应该如何实现同样的功能？
5.0使用模型的 `allowField` 方法可以在使用 `save` 方法的时候过滤非数据表字段，例如：

```
$user = new User;
$user->allowField(true)->save($_POST);
```

或者指定字段写入：

```
$user = new User;
$user->allowField(['email','name'])->save($_POST);
```

为什么路由变量用 $_GET 获取不到？

5.0版本的变量获取界限有调整，路由变量以及 pathinfo 地址参数是不能使用$_GET方式获取的，例如：注册了下面的路由地址：

```
Route::Rule('hello/:name','index/hello');
```

然后，访问下面的URL地址：

```
http://serverName/hello/thinkphp/id/8
```

使用 $_GET['name']、$_GET['id'] 或者 input('get.name') 、input('get.id')是不能正确获取到变量的，正确的方式是：

```
input('name');
input('id');
// 或者
request()->param('name');
request()->param('id');
```

> Request类的param方法用于获取**当前请求**的变量，而get方法是用于获取$_GET变量，而pathinfo地址中的参数是不属于GET变量范畴的。

如何关闭未定义变量的错误提示？

本着严谨的原则，5.0版本默认情况下会对任何错误（包括警告错误）抛出异常，如果不希望如此严谨的抛出异常，可以在**应用公共函数文件**中或者**应用配置文件**中使用 error_reporting 方法设置错误报错级别（请注意，在入口文件中设置是无效的），例如：

```
// 设置异常错误报错级别 关闭notice错误
error_reporting(E_ALL ^ E_NOTICE );
```

如何获取当前的模块、控制器和操作名

5.0版本取消了原来的代表当前模块、控制器和操作名的常量，如果需要获取这些，可以改成：

```
// 当前模块名
request()->module();
// 当前控制器名
request()->controller();
// 当前操作名
request()->action();
```

如果需要在模板里面输出，则可以使用：

```
{$Request.module}
{$Request.controller}
{$Request.action}
```

如果觉得不方便，也可以自己在模块的公共文件里面重新定义常量后操作。

如何让生成的URL地址带上index.php

5.0使用url方法生成的URL地址不带index.php，如果你的服务器环境不支持URL重写，那么可以使用下面的方式：

\think\Url::root('/index.php');

\think\Url::build('index/hello');

就会生成带`index.php`的URL地址了。

模型类的`field`属性和`type`属性有什么区别？

5.0的模型类`field`属性用于定义模型对应数据表的字段信息（包括字段类型），这里定义的字段类型是数据表的实际字段类型，例如：

```
protected $field = [
    'id'          => 'int',
    'birthday'    => 'int',
    'status'      => 'int',
    'create_time' => 'int',
    'update_time' => 'int',
    'nickname', 'email',
];
```

`field`属性必须明确设置数据表所有的字段，字符串类型可以不明确定义类型（例如上面的`nickname`和`email`）。

`type`属性用于定义数据的自动转换类型，代表的不一定是数据表的字段类型，目前支持的自动转换类型包括integer、float、boolean、timestamp、datetime、object、array、json和serialize，例如：

```
protected $type = [
    'birthday'    => 'timestamp',
];
```

`type`属性只需要定义需要系统自动转换的字段。

模型的save方法调用后怎么获取自增主键的值？

模型的save方法用于模型对象的保存（包括新增和更新）操作，当新增数据后，返回值不再返回主键，而是统一返回影响的记录数（一般为1或者0），如果需要获取自增主键的值，可以使用下面的方法：

```
$user = new User;
$user->name = 'thinkphp';
$user->email = 'thinkphp@qq.com';
```

```
if($user->save()){
    // 获取自增主键的值
    echo $user->id;
}
```

B、3.2和5.0区别

`5.0`版本和之前版本的差异较大，本篇对熟悉`3.2`版本的用户给出了一些`5.0`的主要区别。

URL和路由

`5.0`的URL访问不再支持普通`URL`模式，路由也不支持正则路由定义，而是全部改为规则路由配合变量规则（正则定义）的方式：

主要改进如下;

- 增加路由变量规则；
- 增加组合变量支持；
- 增加资源路由；
- 增加路由分组；
- 增加闭包定义支持；
- 增加MISS路由定义；
- 支持URL路由规则反解析；

请求对象和响应对象

`5.0`新增了请求对象`Request`和响应对象`Response`，`Request`统一处理请求和获取请求信息，`Response`对象负责输出客户端或者浏览器响应。

模块和控制器

控制器的命名空间有所调整，并且可以无需继承任何的控制器类。

- 应用命名空间统一为`app`（可定义）而不是模块名；
- 控制器的类名默认不带`Controller`后缀，可以配置开启`use_controller_suffix`参数启用控制器类后缀；
- 控制器操作方法采用`return`方式返回数据 而非直接输出；
- 废除原来的操作前后置方法；
- 增加`beforeActionList`属性定义前置操作；
- 支持任意层次的控制器定义和访问；
- URL访问支持自动定位控制器；

数据库

5.0的数据库查询功能增强，原先需要通过模型才能使用的链式查询可以直接通过`Db`类调用，原来的`M`函数调用可以改用`db`函数，例如：

3.2版本

```
M('User')->where(['name'=>'thinkphp'])->find();
```

5.0版本

```
db('User')->where('name','thinkphp')->find();
```

主要改进如下：

- 支持链式查询操作；
- 数据查询支持返回对象、数组和 `PDOStatement` 对象；
- 数据集查询支持返回数组和 `Collection` 对象；
- 增加查询构造器，查询语法改变；
- 支持闭包查询；
- 支持分块查询；
- 支持视图查询；
- 增加SQL监听事件；

模型

`5.0` 的模型变化是最大的，基本上模型是完全面向对象的概念，包括关联模型，模型类的后缀不再带 `Model` ，直接由命名空间区分，原来的 `D` 函数调用改为 `model` 函数，并且必须创建对应的模型类，例如：

3.2版本

```
D('User')->where(['name'=>'thinkphp'])->find();
```

5.0版本

```
model('User')->where('name','thinkphp')->find();
```

主要改进包括：

- 重构关联模型；
- 支持聚合模型；
- 废除视图模型（改为数据库的视图查询方法）；
- 模型的扩展采用 `Trait` 机制；
- 增加获取器和修改器；
- 增加时间戳自动写入；
- 增加类型字段转换；
- 数组访问支持；
- JSON序列化支持；

自动验证和自动完成

5.0的数据自动验证和自动完成和3.2版本区别较大，5.0的数据验证采用验证器定义并且通过 `think\Validate` 类进行统一的验证。自动完成则通过在模型里面定义修改器来完成。

异常

`5.0` 对错误零容忍，默认情况下会对任何级别的错误抛出异常（但可以在应用公共文件中设置错误级别），并且重新设计了异常页面，展示了详尽的错误信息，便于调试。

调试和日志

`5.0` 的页面 `Trace` 强化，支持浏览器控制台查看Trace信息。

`5.0` 的日志驱动增加 `Socket` 方式，采用 `SocketLog` 支持远程调试。

常量

`5.0` 版本废弃了原来的大部分常量定义，仅仅保留了框架的路径常量定义，其余的常量可以使用 `App` 类或者 `Request` 类的相关属性或者方法来完成，或者自己重新定义需要的常量。

废除的常量包括：

```
REQUEST_METHOD IS_GET IS_POST IS_PUT IS_DELETE IS_AJAX __EXT__ COMMON_MODULE MODULE_NAM
E CONTROLLER_NAME ACTION_NAME APP_NAMESPACE APP_DEBUG MODULE_PATH
```

函数

`5.0` 版本核心框架不依赖任何自定义函数，但仍然封装了一些常用功能到助手函数，你可以随意重新定义或者增加助手函数。

C、助手函数

本篇汇总了系统提供的助手函数的基础用法，更详细的请参考相关章节。

系统的助手函数大致分为下面几个类型：

- 加载和实例化
- 数据操作
- 日志和调试
- 响应输出
- 其它

加载和实例化

import：导入所需的类库 同java的Import 本函数有缓存功能

参数：

名称	描述	默认值
class	类库命名空间字符串	必须
baseUrl	起始路径，留空为自动识别	空
ext	导入的文件扩展名	EXT常量

返回值：
导入成功返回true，否则返回false。

示例：
注意该方法只是导入文件，和是否有命名空间和如何实例化无关。

```
$result = import('org/util/Array');
```

vendor：导入vendor目录下的第三方类库（非命名空间）

参数：

名称	描述	默认值
class	要导入的类库	必须
ext	导入的文件扩展名	EXT常量

返回值：
导入成功返回true，否则返回false。

示例：
注意该方法只是导入文件，和是否有命名空间和如何实例化无关。

```
$result = vendor('org/util/Array');
```

load_trait：快速导入Traits PHP5.5以上无需调用

参数：

名称	描述	默认值
class	要导入的trait库	必须
ext	类库后缀	EXT常量

返回值：

导入成功返回true，否则返回false。

示例：

```
load_trait('controller/Jump');
```

model：实例化模型类（单例）

参数：

名称	描述	默认值
name	Model名称	必须
layer	模型层名称	model
appendSuffix	是否添加类名后缀	false

返回值：

模型实例

示例：

```
$user = model('User');
```

validate：实例化验证器类（单例）

参数：

名称	描述	默认值
name	验证器名称	必须
layer	验证层名称	validate
appendSuffix	是否添加类名后缀	false

返回值：

验证器实例

示例：

```
$validate = validate('User');
```

db：实例化数据库类

参数：

名称	描述	默认值
name	操作的数据表名称（不含前缀）	必须
config	数据库配置参数，数组或者字符串，留空获取配置文件中的配置	空
force	是否强制重新连接	true

返回值：

数据库查询对象实例

示例：

```
$db = db('user');
```

controller：实例化控制器类

参数：

名称	描述	默认值
name	控制器资源地址 [模块/]控制器	必须
layer	控制器层名称	controller
appendSuffix	是否添加类名后缀	false

返回值：

控制器对象实例

示例：

```
$user = controller('Admin/User');
```

action：调用模块的操作方法 参数格式 [模块/控制器/]操作

参数：

名称	描述	默认值
url	调用地址	必须
vars	调用参数 支持字符串和数组	空
layer	要调用的控制层名称	controller
appendSuffix	是否添加类名后缀	false

返回值：

根据方法的返回值

示例：

```
$result = action('Admin/User/getInfo');
```

request：获取当前的请求对象实例（单例）

参数：

无

返回值：

`\think\Request` 对象实例

示例：

```
request()->url();
request()->param();
```

response：创建响应对象实例

参数：

名称	描述	默认值
data	响应输出数据	空
code	输出状态码	200
header	输出HEADER信息	空
type	响应输出类型	html

返回值：

`\think\response` 对象实例

示例：

```
response($data,[],[],'json');
```

数据操作

session：Session操作助手函数

参数：

名称	描述	默认值
name	session名称，如果为数组表示进行session初始化	必须
value	session值	空
prefix	session变量前缀	null

返回值：
多种情况

示例：
```
// session初始化
session(['id'=>'think','prefix'=>'think']);
// session设置
session('name','value');
// session判断
session('?name');
// session获取
session('name');
// session删除
session('name',null);
// session清空
session(null);
```

cookie：Cookie操作助手函数

参数：
名称	描述	默认值
name	cookie名称，如果为数组表示进行cookie初始化	必须
value	cookie值	空
option	参数	null

返回值：
多种情况

示例：
```
// cookie初始化
cookie(['expire'=>3600,'prefix'=>'think']);
// cookie设置
cookie('name','value');
// cookie判断
cookie('?name');
// cookie获取
cookie('name');
// cookie删除
cookie('name',null);
// cookie清空
cookie(null);
```

cache：缓存操作助手函数

参数：
名称	描述	默认值
name	缓存名称，如果为数组表示进行缓存初始化	必须
value	缓存值	空

options	缓存参数 或者传入数字的时候为有效期	null
tag	缓存标签	null

返回值：
多种情况

示例：

```
// 缓存初始化
cache(['expire'=>3600,'prefix'=>'think']);
// 缓存设置
cache('name','value',3600);
// 缓存判断
cache('?name');
// 缓存获取
cache('name');
// 缓存删除
cache('name',null);
```

config：设置或者获取配置参数

参数：

名称	描述	默认值
name	参数名 如果是数组表示批量赋值	必须
value	参数值	空
range	作用域，留空表示当前作用域	空

返回值：
根据情况

示例：

```
config('name','value');
config('?name');
config('name');
config(['name'=>'value']);
```

input：获取输入数据 支持默认值和过滤

参数：

名称	描述	默认值
key	获取的变量名	必须
default	默认值	空
filter	过滤方法	空

返回值：
根据情况

示例：

```
input('name');
input('?name');
input('get.id',0);
```

日志和调试

dump：浏览器友好的变量输出

参数：

名称	描述	默认值
var	调试变量或者信息	必须
echo	是否输出 默认为true 如果为false 则返回输出字符串	true
label	标签	空

返回值：

无

示例：

```
dump($data);
dump($data,false);
dump('测试输出');
```

halt：调试变量并中止

和dump函数的区别在于调试输出后中止执行后面的程序。

参数：

名称	描述	默认值
var	调试变量或者信息	必须

返回值：

无

示例：

```
halt($data);
halt('测试输出');
```

trace：记录日志信息

参数：

名称	描述	默认值
log	log信息 支持字符串和数组	
level	日志级别	log

返回值：
没有传任何参数的时候 获取日志信息，否则为记录日志

示例：

```
trace('日志错误信息','error');
```

exception：抛出异常

参数：

名称	描述	默认值
msg	异常消息	必须
code	异常代码	0
exception	异常类名 留空抛出 `\think\Exception` 异常	空

返回值：
字符串

示例：

```
exception('错误');
exception('权限错误',10002,'\app\common\AuthException');
```

debug：调试时间（微秒）和内存使用情况

参数：

名称	描述	默认值
start	开始标签	必须
end	结束标签	空字符串
dec	小数位 如果是m 表示统计内存占用	6

返回值：
根据情况

示例：

```
debug('begin');
// ...
echo debug('begin','end',6);
echo debug('begin','end','m');
```

响应输出

view：模板渲染输出

参数：

名称	描述	默认值
template	模板文件	空
vars	模板变量	无
replace	模板替换规则	空
code	状态码	200

返回值：

`\think\response\View` 对象实例

示例：

```
view('index/hello',['name'=>'thinkphp']);
```

json：创建JSON响应对象实例

参数：

名称	描述	默认值
data	响应输出数据	空
code	输出状态码	200
header	输出头信息	空
options	输出参数	空

返回值：

`\think\response\Json` 对象实例

示例：

```
json(['id'=>10, 'name'=>'thinkphp']);
```

jsonp：创建JSONP响应对象实例

参数：

名称	描述	默认值
data	响应输出数据	空
code	输出状态码	200
header	输出头信息	空
options	输出参数	空

返回值：

`\think\response\Jsonp` 对象实例

示例：

```
jsonp(['id'=>10, 'name'=>'thinkphp']);
```

xml：创建XML响应对象实例

参数：

名称	描述	默认值
data	响应输出数据	空
code	输出状态码	200
header	输出头信息	空
options	输出参数	空

返回值：

`\think\response\Xml` 对象实例

示例：

```
xml(['id'=>10, 'name'=>'thinkphp']);
```

redirect：创建重定向响应对象实例

参数：

名称	描述	默认值
url	重定向地址	空
params	参数，当传入数字的时候表示状态码	空
code	状态码	302

返回值：

`\think\response\Redirect` 对象实例

示例：

```
redirect('http://thinkphp.cn', ['id'=>10, 'name'=>'thinkphp'],301);
redirect('http://thinkphp.cn/blog', 301);
```

abort：抛出HTTP异常

参数：

名称	描述	默认值
code	状态码 或者 Response对象实例	必须
message	错误信息	空
header	参数	空

返回值：

无

示例：

```
abort(401, '页面不存在');
abort(500, '服务器错误');
```

其它

token：生成表单令牌（form表单项）

参数：

名称	描述	默认值
name	令牌名称	**token**
type	令牌生成方法	md5

返回值：

字符串

示例：

```
<form action="" >
<input type="text" name="name" />
{:token('__hash__')}
<input type="submit" />
</form>
```

url：生成URL地址（支持路由反解）

参数：

名称	描述	默认值
url	路由地址	空
vars	变量	空
suffix	生成的URL后缀	true 表示自动识别配置
domain	是否带域名生成	false

返回值：

字符串

示例：

```
url('index/index/hello',['name'=>'thinkphp']);
url('index/index/hello','name&thinkphp');
```

lang：获取语言变量值

参数：

名称	描述	默认值
name	语言变量名	必须
vars	动态变量值	空
lang	语言 留空自动识别当前语言	空

返回值：
字符串

示例：

```
lang('lang_str');
```

widget：渲染输出Widget

参数：

名称	描述	默认值
name	Widget名称	必须
data	传入的参数	空

返回值：
根据情况

示例：
通常在模板文件中调用

```
{:widget('name')}
```