ThinkPHP V5.0

——

**为API开发而设计的高性能框架**

作者：https://www.kancloud.cn/manual/thinkphp5/118003

第一章 基础

【安装ThinkPHP】

1. ThinkPHP5环境要求：
2. PHP >= 5.4.0
3. PDO PHP Extension
4. MBstring PHP Extension
5. CURL PHP Extension
6. Composer安装ThinkPHP5
7. Linux下安装Composer命令：

curl -sS <https://getcomposer.org/installer> | php mv composer.phar /usr/local/bin/composer

1. Composer更新命令：composer self -update
2. 在命令行下面，切换到web根目录执行下面的命令：

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

1. Git安装
2. ThinkPHP拆分为多个仓库，主要包括
3. 应用项目：<https://github.com/top-think/think>
4. 核心框架：https://github.com/top-think/framework
5. 首先克隆下载应用项目仓库：

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

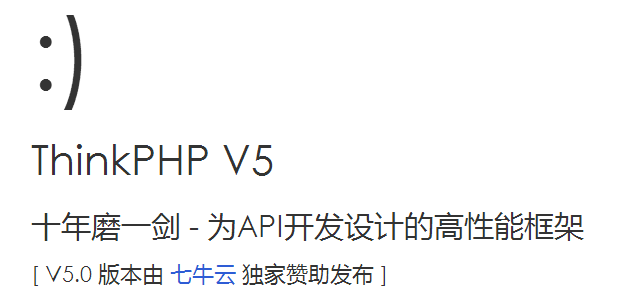
1. 然后切换到tp5目录下面，再克隆核心框架仓库：

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

1. 两个仓库克隆完成后，就完成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

1. 在浏览器输入：<http://localhost/tp5/public>，显示下图即为安装成功。



【开发规范】

ThinkPHP5遵循PSR-2命名规范和PSR-4自动加载规范，并且注意如下规范：

* 1. 目录和文件：
     1. 目录使用小写+下划线；
     2. 类库、函数文件统一以.php为后缀；
     3. 类的文件名均以命名空间定义，并且命名空间的路径和类库文件所在路径一致；
     4. 类文件采用驼峰法明敏（首字母大写），其他文件采用小写+下划线命名；
     5. 类名和类文件名保持一致，统一采用驼峰法命名（首字母大写）；
  2. 函数和类、属性命名：
     1. 类的命名采用驼峰法（首字母大写），例如User、UserType,默认不需要添加后缀，例如UserController应该直接命名为User;
     2. 函数的命名使用小写字母和下划线（小写字母开头）的方式，例如get\_client\_ip；
     3. 方法的命名使用驼峰法（首字母小写），例如getUserName;
     4. 属性的命名使用驼峰法（首字母小写），例如tableName、instance
     5. 以双下划綫“\_\_”打头的函数或方法作为魔术方法，例如\_\_call 和 \_\_autoload;
  3. 常量和配置：
     1. 常量以大写字母和下划线命名，例如APP\_PATH 和 THINK\_PATH；
     2. 配置参数以小写字母和下划线命名，例如url\_route\_on 和 url\_convert;
  4. 数据表和字段：

数据表和字段采用小写加下划线方式命名，并注意字段名不要以下划线开头，例如think\_user 表和 user\_name字段，不建议使用驼峰和中文作为数据表字段命名。

* 1. 应用类库命名空间规范

应用类库的根命名空间统一为app（不建议更改，可以设置app\_namespace配置参数更改，v5.0.8版本开始使用APP\_NAMESPACE常量定义）

例如：app\index\controller\Index 和 app\index\model\User

请尽量避免使用PHP保留字（保留字列表参加http://php.net/manual/zh/reserved.keywords.php）作为常量、类名和方法名，以及命名空间的命名，否则会造成系统错误。

【目录结构】

Project 应用部署目录

|--application 应用目录（可设置）

| |--common 公共模块目录（可更改）

| |--index 模块目录（可更改）

| | |--config.php 模块配置文件

| | |--common.php 模块函数文件

| | |--controller 控制器目录

| | |--model 模型目录

| | |--view 视图目录

| | |--。。。 更多类库目录

| |--command.php 命令行工具配置文件

| |--common.php 应用公共（函数）文件

| |--config.php 应用（公共）配置文件

| |--database.php 数据库配置文件

| |--tags.php 应用行为扩展定义文件

| |--route.php 路由配置文件

|--extend 扩展类库目录（可定义）

|--public WEB部署目录（对外访问目录）

| |--static 静态资源存放目录(css，js，image)

| |--index.php 应用入口文件

| |--router.php 快速测试文件

| |--.htaccess 用于apache的重写

|--runtime 应用的运行时目录（可写，可设置）

|--vendor 第三方类库目录(Composer)

|--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 基础定义文件

| |--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

|--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 命令行入口文件

注：如果是mac或者linux环境，请确保rutime目录有可写权限；

5.0的部署建议是public目录作为web目录访问内容，其他都是web目录之外，当然，你必须修改public/index.php中的相关路径。如果没法做到这点，请记得设置目录的访问权限或者添加目录列表的保护明文件。

router.php 用于php自带webserver支持，可用于快速测试

启动命令：php -S localhost:8888 router.php

第二章 架构

【架构总览】

ThinkPHP5.0应用基于MVC（模型-视图-控制器）的方式来组织。

MVC是一个设计模式，它强制性的使应用程序的输入、处理和输出分开。使用MVC应用程序被分成三个核心部件：模型（M）、视图（V）、控制器（C），他们各自处理自己的任务。

5.0的URL访问受路由决定，如果关闭路由或者没有匹配路由的情况下，则是基于：

<http://serverName/index.php> （或者其他应用入口文件）/模块/控制器/操作/参数/值…

1. 入口文件

用户请求的PHP文件，负责处理一个请求(注意,不一定是URL请求)的生命周期，最常见的入口文件就是index.php，有时候也会为了某些特殊的需求而增加新的入口文件，例如给后台模块单独设置的一个入口文件admin.php或者一个控制器程序入口think都属于入口文件。

1. 应用

应用在ThinkPHP中是一个管理系统架构及生命周期的对象，由系统的\think\App类完成，应用通常在入口文件中被调用和执行，具有相同的应用目录（APP\_PATH）的应用我们认为是同一个应用，但一个应用可能存在多个入口文件。

1. 模块

一个典型的应用有多个模块组成的，这些模块通常都是应用目录下面的一个子目录，每个模块都有自己独立的配置文件、公共文件和类库文件。

5.0支持单一模块架构设计，如果你的应用下面只有一个模块，那么这个模块的子目录可以省略，并且在应用配置文件中修改：

‘app\_multi\_module’ => false

1. 控制器

每个模块拥有独立的MVC类库及配置文件，一个模块下面有多个控制器负责响应请求，而每个控制器其实就是一个独立的控制器类。控制器主要负责请求的接受，并调用相关的模型处理，并最终通过视图输出。严格来说，控制器不应该过多的介入业务逻辑处理。

事实上，5.0中控制器是可以被跳过的，通过路由我们可以直接把请求调度到某个模型或者其他的类进行处理。

5.0的控制器类比较领过，可以无需继承任何基础类库。

一个典型的Index控制器类如下：

namespace app\index\controller;

class Index

{

public function index()()

{

Return ‘hello world!’;

}

}

1. 操作

一个控制器包含多个操作（方法），操作方法是一个URL访问的最小单元。下面是一个典型的Index控制器的操作方法定义，包含了两个操作方法：

namespace app\index\controller;

class Index

{

public function index()

{

return ‘index’;

}

public function hello($name)

{

return ‘hello,’.$name

}

}

操作方法可以不使用任何参数，如果定义了一个非可选参数，则该参数必须通过用户请求传入，如果是URL请求，则通常是$\_GET或者$\_POST方式传入。

1. 模型

模型类通常完成实际的业务逻辑和数据封装，并返回和格式无关的数据。

模型类并不一定要访问数据库，而且在5.0的架构设计中，只有进行实际的数据库查询操作的时候，才会进行数据库的连接，是真正的惰性连接。

ThinkPHP的模型层支持多层设计，你可以对模型层进行更细化的设计和分工，例如把模型层分为逻辑层/服务层/事件层等等。

1. 视图

控制器调用模型类后返回的数据通过视图组装成不同格式的输出。视图根据不同的需求，来决定调用模板引擎进行内容解析后输出还是直接输出。

视图通常会有一系列的模板文件对应不同的控制器和操作方法，并且支持动态设置模板目录。

1. 驱动

系统很多的组件都采用驱动式设计，从而可以更灵活的扩展，驱动类的位置默认是放入核心类库目录下面，也可以重新定义驱动类库的命名空间而改变驱动的文件位置。

1. 行为

行为（Behavior）是在预先定义好的一个应用位置执行的一些操作。类似于AOP编程中的“切面”的概念，给某一个切面绑定相关行为就成了一中类AOP编程的思想。所以，行为通常是和某个位置相关，行为的执行时间依赖于绑定到了哪个位置上。

要执行行为，首先要在应用程序中进行行为侦听，例如

//在app\_init位置侦听行为

\think\Hook::listen(‘app\_init’);

然后对某个位置进行行为绑定：

\think\Hook::add(‘app\_init’,’\app\index\behavior\Test’);

一个位置上如果绑定了多个行为的，按照绑定的顺序依次执行，除非遇到中断。

1. 命名空间

ThinkPHP5采用了PHP的命名空间进行类库文件的设计和规划，并且符合PSR-4的自动加载规范。

【生命周期】

1. 入口文件
2. 用户发起的请求都会经过应用的入口文件，通常是public/index.php文件。当然，你也可以更改或者增加新的入口文件。

通常入口文件的代码都比较简单，一个普通的入口文件代码如下：

//应用入口文件

//定义项目路径

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

//加载框架引导文件

require \_DIR\_ .’/…/thinkphp/start.php’;

一般入口文件以定义一些常量为主，支持的常量请参考后续的内容或者附录部分。

1. 引导文件

接下来就是执行框架的引导文件，start.php文件就是系统默认的一个引导文件。在引导文件中，会依次执行下面操作：

加载系统常量定义；

加载环境变量定义文件；

注册自动加载机制；

注册错误和异常处理机制；

加载惯例配置文件；

执行应用；

Start.php引导文件首先会调用base.php基础引导文件，某些特殊需求下面可能直接在入口文件中引入基础引导文件。

如果在你的应用入口文件中更改了默认的引导文件，则上述执行流程可能会跟随发生变化。

1. 注册自动加载

系统会调用Loader::register()方法注册自动加载，在这一步完成后，所有符合规范的类库（包括Composer依赖加载的第三方类库）都将自动加载。

系统的自动加载由下面主要部分组成：

1. 注册系统的自动加载方法 \think\Loader::autoload
2. 注册系统命名空间定义
3. 加载类映射文件（如果存在）
4. 如果存在Composer安装，则注册\*\* Composer \*\* 自动加载
5. 注册extend扩展目录

一个类库的自动加载检测顺序为：

1. 是否定义类库映射；
2. PSR-4自动加载检测；
3. PSR-0自动加载检测；

可以看到，定义类库映射的方式是最高效的。

1. 注册错误和异常机制

执行Error::register()注册错误和异常处理机制。

由三部分组成：

应用关闭方法：think\Error::appShutdown

错误处理方法：think\Error::appError

异常处理方法：think\Error:::appException

注册应用关闭方法是为了便于拦截一些系统错误。

在整个应用请求的生命周期过程中，如果跑出了异常或者严重错误，均会导致应用提前结束，并相应输出异常和错误信息。

1. 应用初始化

执行应用的第一步操作就是对应用进行初始化，包括：

加载应用（公共）配置；

加载扩展配置文件（由extra\_config\_list定义）；

加载应用状态配置；

加载别名定义；

加载行为定义；

加载公共（函数）文件；

注册应用命名空间；

加载扩展函数文件（有extra\_file\_list定义）；

设置默认时区；

加载系统语言包；

1. URL访问检测

应用初始化完成后，就会进行URL的访问检测，包括PATH\_INFO检测和URL后缀检测。

5.0的URL访问必须是PATH\_INFO方式（包括兼容方式）的URL地址，例如：

<http://serverName/index.php/index/index/hello/val/value>

所以，如果你的环境只能支持普通的URL参数访问，那么必须使用

<http://serverName/index.php?s=/index/index/hell0&val=value>

如果是命令行下面访问入口文件的话，则通过

$php index.php index/index/hello/val/value。。。

获取到正常的$\_SERVER[‘PATH\_INFO’]参数后才能继续。

1. 路由检测

如果开起来url\_route\_on参数的话，会首先进行URL的路由检测。如果一旦检测到匹配的路由，根据定义的路由地址会注册到相应的RUL调度。

5.0的路由地址支持如下方式：

路由到模块/控制器/操作；

路由到外部重定向地址；

路由到控制器方法；

路由到闭包函数；

路由到类的方法；

路由地址可能会受域名绑定的影响。

如果关闭路由或者路由检测无效则进行默认的**模块/控制器/操作**的分析识别。

如果在应用初始化的时候制定了应用调度方式，那么路由检测是可选的。

可以使用\think\APP::dispatch()进行应用调度，例如：

App::dispatch([‘type’=>’module’,’module’=>’index/index’]);

1. 分发请求

在完成了URL检测和路由检测之后，路由器会分发请求到对应的路由地址，这也是应用请求的生命周期中最重要的一个环节。

在这一步骤中，完成应用的业务逻辑及数据返回。

建议统一使用return返回数据，而不是echo输出，如非必要，请不要使用exit或者die中断执行。

直接echo输出的数据将无法进行自动转换响应输出的便利。

下面是系统支持的分发请求机制，可以根据情况选择：

**模块/控制器/操作**

这是默认的分发请求机制，系统会根据URL或者路由地址来判断当前请求的模块、控制器和操作明，并自动调用相应的访问控制器类，执行操作对应的方法。

该机制下面，首先会判断当前模块，并进行模块的初始化操作（和应用的初始化操作类似），模块的配置参数会覆盖应用的尚未生效的配置参数。

支持模块映射、URL参数绑定到方法，一级操作绑定到类等一些功能。

**控制器方法**

和前一种方法类似，知识无需判断模块、控制器和操作，直接分发请求到一个指定的控制器类的方法，因此进行模块的初始化操作。

**外部重定向**

可以直接分发请求到一个外部的重定向地址，支持指定重定向代码，默认为301重定向。

**闭包函数**

路由地址定义的时候可以直接采用闭包函数，完成一些相对简单的逻辑操作和输出。

类的方法

除了以上方式外，还支持分发请求到类的方法，包括：

静态方法：’blog/:id’=>’\org\util\Blog::read’

类的方法：’blog/:id’=>’\app\index\controller\Blog@read’

1. 响应输出

控制器的所有操作方法都是return返回而不是直接输出，系统会调用Response::send方法将最终的应用返回的数据输出到页面或者客户端，并自动转换成default\_return\_type参数配置的格式。所以，应用之星的数据输出只需要返回一个正常的PHP数据即可。

1. 应用结束

事实上，在应用的数据相应输出之后，应用并没真正的结束，系统会在应用输出或者中断后进行日志保存写入操作。系统的日志包括用户调试输出的和系统自动生成的日志，统一会在应用结束的时候进行写入操作。

而日志的写入操作受日志初始化的影响。

ThinkPHP采用单一入口模式进行项目部署和访问，无论完成什么功能，一个应用都有一个统一（但不一定是唯一）的入口，应该说，所有应用都是从入口文件开始的，并且不同应用的入口文件是类似的。

【入口文件】

入口文件主要完成：

1. 定义框架路径、项目路径（可选）
2. 定义系统相关常量（可选）
3. 载入框架入口文件（必须）

5.0默认的应用入口文件位于public/index.php，内容如下：

//定义应用目录

difine(‘APP\_PATH’,\_DIR\_ . ‘/../application/’);

//加载框架引导文件

require \_DIR\_ . ‘/../thinkphp/start.php’;

入口文件位置的设计是为了让应用部署更安全，public目录为web可访问目录，其他的文件都可以放到非WEB访问目录下面。

修改入口文件位置请查看章节<部署-虚拟主机环境>

入口文件中还可以定义一些系统变量，用于相关的绑定操作（通常用于多个入口的情况），这个会在后面涉及，暂且不提。

给APP\_PATH定义绝对路径会提高系统的加载效率。

在有些情况下，你可能需要加载框架的基础引导文件base.php，该引导文件和start.php的区别是不会主动执行应用，而是需要自己进行应用执行，下面是一个例子：

//定义应用目录

Define(‘APP\_PATH’,\_DIR\_ . ‘/../application/’);

//加载框架基础引导文件

Require \_DIR\_ . ‘/../thinkphp/base.php’;

//添加额外的代码

// . . .

//执行应用

\think\App::run()->send();

【URL访问】

* 1. URL设计

ThinkPHP5.0在没有启用路由的情况下典型的URL访问规则是：

<http://serverName/index.php>(其他应用入口文件)/模块/控制器/操作/[参数名/参数值…]

支持切换到命令进行访问，如果切换到命令行模式下面的访问规则是：

>php.exe index.php(或者其他应用程序入口文件) 模块/控制器/操作/[参数名/参数值…]

如果不支持PATHINFO的服务器可以使用兼容模式访问如下：

<http://serverName/index.php> (或者其他应用入口文件)?s=/模块/控制器/操作/[参数名/参数值...]

必要的时候，我们可以通过某种方式，省略URL里面的模块和控制器。

* 1. URL大小写

默认情况下，URL是不区分大小写的，也就是说URL里面的模块/控制器/操作名会自动转换为小写，控制器在最后调用的时候会转换为驼峰法处理。

例如：<http://localhost/index.php/Index/Blog/read>

//和下面的访问时等效的

<http://localhost/index.php/index/blog/read>

在这种URL不区分的大小写情况下，如果要访问驼峰法的控制器类，则需要使用：

<http://localhost/index.php/Index/blog_test/read>

模块名和操作名会直接转换为小写处理。

如果希望URL访问严格区分大小写，可以在应用设置文件中设置：

//关闭URL中控制器和操作名的自动转换

‘url\_convert’ = > false,

一旦关闭自动转换，URL地址中的控制器名就编程大小写敏感了，例如前面的访问地址就要写成：<http://localhost/index.php/Index/BlogTest/read>

但是下面的URL访问依然是有效的：

http://localhost/index.php/Index/blog\_test/tead

下面的URL访问则无效：

<http://localhost/index.php/Index/blogtest/read>

需要注意：路由规则中定义的路由地址是按照控制器名的实际名称定义（区分大小写）。

* 1. 隐藏入口文件

在ThinkPHP5.0中，处于优化的URL访问原则，还支持通过URL重写隐藏入口文件，下面以Apache为例说明隐藏应用入口文件Index.php的设置。

下面是Apache的配置过程，可以参考下：

1. Httd.conf配置文件中加载了mod\_rewrite.so模块
2. AllowOverride None将None改为All
3. 在应用入口文件同级目录添加.htaccess文件，内容如下：

<IfModule mod\_rewrite.c>

Options +FollowSymlinks -Multiviews

RewirteEngine on

RewriteCond %{REQUEST\_FILENAME} ! - d

RewriteCond %{REQUEST\_FILENAME} ! – f

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

</IfModule>

更多其他环境的隐藏入口文件参考后面的URL重写

【模块设计】

5.0版本对模块的功能做了灵活设计，默认采用多模块的架构，并且支持单一模块设计，所有模块的命名空间均以app作为根命名空间（可配置更改）

1. 目录结构

标准的应用和模块目录结构如下：

|—application 应用目录（可设置）

| |—common 公共模块目录（可选）

| |—common.php 公共函数文件

| |—route.php 路由配置文件

| |—database.php 数据库配置文件

| |—config.php 应用配置文件

| |—module1 模块1目录

| | |—config.php 模块配置文件

| | |—common.php 模块函数文件

| | |—controller 控制器目录

| | |—mode1 模型目录（可选）

| | |—view 视图目录（可选）

| | |—. . . 更多类库目录

| |

| |—module2 模块2目录

| | |—config.php 模块配置文件

| | |—common.php 模块函数文件

| | |—controller 控制器目录

| | |—mode1 模型目录（可选）

| | |—view 视图目录（可选）

| | |—. . . 更多类库目录

遵循ThinkPHP5.0的命名规范，模块目录全部采用小写和下划线命名

模块名称请避免使用PHP保留关键字，否则会造成系统错误。

其中common模块是一个特殊的模块，默认是禁止直接访问的，一般用于放置一些公共的类库用于其他模块的继承。

1. 模块类库

一个模块下面的类库文件的命名空间统一以app\模块名开头，例如：

//index模块的Index控制器类

app\index\controller\Index

//index模块的User模型类

app\index\model\User

其中app可以通过定义的方式更改，例如我们在应用配置文件中修改：

‘app\_namespace’ => ‘application’,

那么，index模块的类库命名空间则变成：

//index模块的Index控制器类

application\index\controller\Index

//index模块的User模型类

application\index\model\User

更多关于类库和命名空间的关系可以参考下一章节：命名空间。

1. 模块和控制器隐藏

由于默认是采用模块的支持，所以多个模块的情况下必须在URL地址中标识当前模块，如果只有一个模块的话，可以进行模块绑定，方法是应用的入口文件中添加如下代码：

//绑定当前访问到Index模块

define(‘BIND\_MODULE’,’index’);

绑定后，我们的URL访问地址则变成：

<http://serverName/index.php/控制器/操作/>[参数名/参数值..]

访问的模块是index模块。

如果你的应用比较简单，模块和控制器都只有一个，那么可以在应用公共文件中绑定模块和控制器，如下：

//绑定当前访问到index模块的index控制器

define(‘BIND\_MODULE’,’index/index’);

设置后，我们的URL访问地址则变成：

<http://serverName/index.php/操作/>[参数名/参数值...]

访问的模块是index模块，控制器是Index控制器

1. 单一模块

如果你的应用比较简单，只有一个模块，那么可以进一步简化成使用单一模块结构，方法如下：

首先在应用配置文件中定义：

//关闭多模块设计

‘app\_multi\_module’ => false,

然后，调整应用目录的结构如下：

|—application 应用目录（可设置）

| |—controller 控制器目录

| |—model 模型目录

| |—view 视图目录

| |—. . . 更多类库目录

| |—common.php 函数文件

| |—route.php 路由配置文件

| |—database.php 数据库配置文件

| |—config.php

URL访问地址变成

<http://serverName/index.php（或者其他应用入口）/>控制器/操作/[参数名/参数值...]

同事，单一模块设计下的应用类库的命名空间也有所调整，例如：

原来的：

app\index\controller\Index

app\index\model\User

变成：

app\controller\Index

app\model\User

更多的URL简化和定制还可以通过URL路由功能实现。

【命名空间】

ThinkPHP5采用命名空间方式定义和自动加载类库文件，有效的解决了多模块和Composer类库之间的命名空间冲突问题，并且实现了更加高效的类库自动加载机制。

特别注意的是，如果你需要调用PHP内置的类库，或者第三方没有使用命名空间的类库，记得在实例化类库的时候加上\，例如：

//错误的用法

$class = new stdClass();

$xml = new SimpleXmlElement($xmlstr);

//正确的用法

$class = new \stdClass();

$xml = new \SimplieXmlElement($xmlstr);

在ThinkPHP5.0中，只需要给类库正确定义所在的命名空间，并且命名空间的路径与类库文件的目录一致，那么就可以实现类的自动加载，从而实现真正的惰性加载。

例如： \think\cache\driver\File类的定义为：

Namespace think\cache\driver;

Class File{

}

如果我们实例化该类的话，应该是：

$class = new \think\cache\driver\File();

系统会自动加载该类对应路径的类文件，其所在的路径是thinkphp/library/think/cache/driver/File.php。

5.0默认的目录规范是小写，类文件命名是驼峰法，并且首字母大写。

原则上，可以支持驼峰法命名的目录，只要命名空间定义和目录一致即可，例如：我们实例化：

$class = new \Think\Cache\Driver\File();

系统则会自动加载 thinkphp/library/Think/Cache/Driver/File.php文件。

1. 根命名空间（类库宝）

根命名空间是一个关键的概念，以上面的\think\cache\driver\File类为例，think就是一个根命名空间，其对应的初始命名空间目录就是系统的类库目录（thinkphp/library/think），我们可以简单的理解一个根命名空间对应了一个类库包。

系统内置的几个根命名空间（类库包）如下：

|  |  |  |
| --- | --- | --- |
| 名称 | 描述 | 类库目录 |
| think | 系统核心类库 | thinkphp/library/think |
| traits | 系统Trait类库 | thinkphp/library/traits |
| app | 应用类库 | application |

如果需要增加新的根命名空间，有两种方式：注册新的根命名空间或者放入EXTEND\_PATH目录（自动注册）。

请注意本手册中的示例代码为了简介，如无指定类库的命名空间的话，都表示指的是think命名空间，例如下面的代码：

Route::get(‘hello’,’index/hello’);

请自行添加use think\Route或者使用

\think\Route::get(‘hello’,’index/hello’);

1. 自动注册

我们只需要把自己的类库包目录放入EXTEND\_PATH目录（默认为extend，可配置），就可以自动注册对应的命名空间，例如：我们在extend目录下面新增一个my目录，然后定义一个\my\Test类（类文件位于extend/my/Test.php）如下：

Namespace my;

Class Test

{

Public function sayHello()

{

Echo ‘hello’;

}

}

我们就可以直接实例化和调用：

$Test = new \my\Test();

$Test -> sayHello();

如果我们在应用入口文件中重新定义了EXTEND\_PATH常量的话，还可以改变\my\Test类文件的位置，例如：define(‘EXTEND\_PATH’,’../vendor/’);

那么\my\Test类文件的位置就变成了/vendor/my/File.php。

1. 手动注册

也可以通过手动注册的方式注册新的根命名空间，例如：

在应用公共文件中添加下面的代码：

\think\Loader::addNamespace(‘my’,’../application/extend/my/’);

如果要同时注册多个根命名空间，可以使用：

\think\Loader::addNamespace([

‘my’ => ‘../application/extend/my/’,

‘org’ => ‘../application/extend/org/’,

]);

也可以直接在应用的配置文件中添加配置，系统会在应用执行的时候自动注册。

‘root\_namespace’ => [

‘my’ => ‘../application/extend/my/’,

‘org’ => ‘../application/extend/org/’,

]

1. 应用类库包

为了避免和Composer自动加载的类库存在冲突，应用类库的命名空间的根都统一以app命名，例如：

namespace app\index\mode1;

class User extends \think\Mode1

{

}

其类文件位于application/index/model/User.php。

namespace app\admin\evect;

class User

{

}

其类文件位于application/admin/event/User.php.

如果觉app根命名空间不适合或者有冲突，可以在应用配置文件中修改：

‘app\_namespace’ => ‘application’,

V5.0.8+版本的话，app\_namespace配置参数改为APP\_NAMESPACE常量入口文件中定义。

定义后，应用类库的命名空间改为：

Namespace application\index\model;

Class User extends \think\Mode1

{

}

1. 命名空间别名

框架允许给命名空间定义别名，例如：

namespace app\index\mode1

use think\Model1;

class User extends Model

{

}

原来在控制器里面调用的方式为：

aamespace app\index\controller;

use app\index\mode1\User;

class Index

{

Public function index()

{

$user = new User();

}

}

如果我们在应用公共文件中注册命名空间别名如下：

\think\Loader::addNamespaceAlias(‘model’,’app\index\model’);

那么，上面的控制器代码就可以更改为：

namespace app\index\controller;

use model\User;

class Index

{

Public function index()

{

$user = new User();

}

}

本手册后续的章节，均建立在你已经了解PHP命名空间的基础之上，如果不掌握请自行补充PHP基础，否则你在后续的文档和ThinkPHP5.0的学习过程中，对命名空间的缺乏理解会成为你最大的学习障碍。

【自动加载】

1. 概述

ThinkPHP5.0真正实现了按需加载，所有类库采用自动加载机制，并且支持类库映射和composer类库的自动加载。自动加载的实现由think\Loader类库完成，自动加载规范符合PHP的PSR-4。

1. 自动加载

由于新版ThinkPHP完全采用了命名空间的特性，因此只需要给类库正确定义所在的命名空间，而命名空间的路径与类库文件的目录一致，那么就可以实现类的自动加载。

类库的自动加载检测顺序如下：

1. 类库映射检测；
2. PSR-4自动加载检测；
3. PSR-0自动加载检测；

系统会按顺序检测，一旦检测生效的话，就会自动载入对应的类库文件。

1. 类库映射

遵循我们上面的命名空间定义规范的话，基本上可以完成类库的自动加载了，但是如果定义了较多的命名空间的话，效率会有所下降，所以，我们可以给常用的类库定义类库映射。命名类库映射相当于给类文件定义了一个别名，效率会比命名空间定位更高效，例如：

Loader::addClassMap(‘think\Log’,LIB\_PATH.’think\Log.php’);

Loader::addClassMap(‘org\util\Array’,LIB\_PATH.’org\util\Array.php’);

也可以利用addClassMap方法批量导入类库映射定义，例如：

$map = [

‘think\Log’ = > LIB\_PATH.’think\Log.php’,

‘org\util\arrary’ => LIB\_PATH.’org\util\Array.php’

];

Loader::addClassMap($map);

虽然通过类库映射的方式注册的类可以不强制要求对应命名空间目录，但是仍然建议遵循PSR-4规范定义类库和目录。

1. 类库导入

如果你不需要系统的自动加载功能，有或者没有使用命名空间的话，那么也可以使用think\Loader类的import方法手动加载类库文件，例如：

Loader::import(‘org.util.arrary’);

Loader::import(‘@.util.upload’);

示例

//引入extend/qrcode.php

Loader::import(‘qrcode’,EXTEND\_PATH);

//助手函数

import(‘qrcode’,EXTEND\_PATH);

//引入extend/wechat-sdk/wechat.class.php

Loader::import(‘wechat-sdk.wechat’,EXTEND\_PATH,’.class.php’);

//助手函数

import(‘wechat-sdk.wechat’,EXTEND\_PATH,’.class.php’);

类库导入也采用类似命名空间的概念（但不需要实际的命名空间支持），支持的”根命名空间”包括：

|  |  |
| --- | --- |
| 目录 | 说明 |
| behavior | 系统行为类库 |
| think | 核心基类库 |
| traits | 系统Traits类库 |
| app | 应用类库 |
| @ | 表示当前模块类库包 |

如果完全遵从系统的命名空间定义的话，一般来说无需手动加载类库文件，直接实例化即可。Thinkphp5.0不推荐使用import方法。

1. Composer自动加载

5.0版本支持Composer安装的类库的自动加载，你可以直接按照Composer依赖库中的命名空间直接调用。

【Traits引入】

ThinkPHP5.0开始采用trait功能（PHP5.4+）来作为一种扩展机制，可以方便的实现一个类库的多继承问题。

trait是一种类似PHP的单继承语言而准备的代码复用机制。trait为了减少单继承语言的限制，使开发人员能够自由地在不同层次结构内独立的类中复用方法集。trait和类组合的语义是定义了一种方式来减少复杂性，避免传统多继承和混入类(Mixin)相关的典型问题。

但由于PHP5.4版本不支持trait的自动加载，因此如果是PHP5.4版本，必须手动导入trait类库，系统提供了一个助手函数load\_trait,用于自动加载trait类库,例如，可以这样正确引入trait类库。

namespace app\index\controller;

load\_trait(‘controller/Jump’); //引入traits\controller\Jump

class index

{

use \traits\controller\Jump;

public function index()

{

$this->assign(‘name’,’value’);

$this->show(‘index’);

}

}

如果你的PHP版本大于5.5的话，则可以省略load\_trait函数引入trait。

namespace app\index\controller;

class index

{

use \traits\controller\Jump;

public function index()

{

}

}

可以支持同时引入多个trait类库，例如：

namespace app\index\controller;

load\_trait(‘controller/Other’);

load\_trait(‘controller/Jump’);

class index

{

use \traits\controller\Other;

use \traits\controller\Jump;

public function index()

{

}

}

或者使用

namespace app\index\controller;

load\_trait(‘controller/Other’);

load\_trait(‘controller/Jump’);

class index

{

use \traits\controller\Other,\traits\controller\Jump;

public function index()

{

}

}

系统提供了一些封装好的trait类库，主要是用于控制器和模型类的扩展。这些系统内置的trait类库的根命名空间采用traits而不是trait，是为了避免和系统的关键字冲突。

trait方式引入的类库需要注意优先级，从基类继承的成员将被trait插入的成员所覆盖。优先顺序是来自当前类的成员覆盖了trait的方法，而trait则覆盖了被继承的方法。

trait类中不支持定义类的常量，在trait中定义的属性将不能在当前类中或者继承的类中重新定义。

1. 冲突的解决

我们可以在一个类库中引入多个trait类库，如果两个trait都定义了一个同名的方法，如果没有明确解决冲突将会产生一个致命错误。

为了解决多个trait在同一个类中的命名冲突，需要使用insteadof操作符来明确指定使用冲突方法中的哪一个。

以上方式仅允许排除掉其他方法，as操作法可以将其中一个冲突的方法以另一个名称来引入。

更多的关于trait内容可以参考PHP官方手册。

【API友好】

1. 数据输出

新版的控制器输出采用Response类统一处理，而不是直接在控制器中进行输出，通过设置default\_return\_type或者动态设置不同类型Response输出就可以自动进行数据处理转换处理，一般来说，你只需要在控制器中返回字符串或者数组即可，例如如果我们配置：

‘default\_return\_type’=>’json’

那么下面的控制器方法返回值会自动转换为json格式并返回。

namespace app\index\controller;

class Index

{

public function index()

{

$data = [‘name’=>’thinkphp’,’url’=>’thinkphp.cn’];

return [‘data’=>$data,’code’=>1,’message’=>’操作完成’];

}

}

访问该请求URL地址后，最终可以在浏览器中看到输出结果如下：

{“data”:{“name”:”thinkphp”,”url”:”thinkphp.cn”},”code”:1,”message”:”\u64cd\u4f5c\u5b8c\u6210”}

如果你需要返回其他的数据格式的话，控制器本身的代码无需做任何改变。

支持明确指出输出类型的方式输出，例如下面制定JSON数据输出：

namespace app\index\controller;

class Index

{

public function index()

{

$data = [‘name’ => ‘thinkphp’,’url’ => ‘thinkphp.cn’];

//指定json数据输出和

return json([‘data’=>$data,’code’=>1,’message’=>’操作完成’]);

}

}

或者指定输出XML类型数据：

namespace app\index\controller;

class Index

{

public function index()

{

$data = [‘name’=>’thinkphp’,’url’ =>’thinkphp.cn’];

//指定xml数据输出

return xml([‘data’=>$data,’code’=>1,’message’=>’操作完成’]);

}

}

核心支持的数据类型包括view、xml、json和jsonp，其他类型的需要自己扩展。

1. 错误调试

由于API开发不方便在客户端进行开发调试，但ThinkPHP5的Trace调试功能支持Socket在内的方式，可以实现远程的开发调试。

设置方式：

‘app\_strace’ => true,

‘trace’ => [

‘type’ => ‘socket’,

//socket服务器

‘host’ => ‘slog.thinkphp.cn’,

],

然后安装chrome浏览器插件后即可进行远程调试，详细参考调试部分。

第三章 配置

ThinkPHP提供了灵活的全局配置功能，采用最有效率的PHP返回数组方式定义，支持惯例配置、公共配置、模块配置、扩展配置、场景配置、环境变量配置、和动态配置。

对于有些简单的应用，你无须配置任何配置文件，而对于复杂的要求，你还可以扩展自己的独立配置文件。

系统的配置参数是通过静态变量全局存取的，存取方式简单高效。

配置功能由\think\Config类完成。

【配置目录】

系统默认的配置文件目录就是应用目录（APP\_PATH），也就是默认的application下面，并分为应用配置（整个应用都有效）和模块配置（仅针对该模块有效）。

|—application 应用目录

| |—config.php 应用配置文件

| |—database.php 数据库配置文件

| |—route.php 路由配置文件

| |—index index模块配置文件目录

| | |—config.php index模块配置文件

| | |—database.php index模块数据库配置文件

如果不希望配置文件放到应用目录下面，可以在入口文件中定义独立的配置目录，添加CONF\_PATH常量定义即可，例如

//定义配置文件目录和应用目录同级

define(‘CONF\_PATH’,\_DIR\_.’/../config/’);

配置目录下面的结构类似如下：

|—application 应用目录

|—config 配置目录

| |—config.php 应用配置文件

| |—database.php 数据库配置文件

| |—route.php 路由配置文件

| |—index index模块配置文件目录

| | |—config.php index模块配置文件

| | |—database.php index模块数据库配置文件

1. 扩展配置目录（V5.0.1）

5.0.1开始增加了扩展配置目录的概念，在应用配置目录或者模块配置目录下面增加extra子目录，下面的配置文件都会自动加载，无需任何配置。

如果你定义了CONF\_PATH常量为config目录为例，扩展配置目录如下：

|—application 应用目录

|—config 配置目录

| |—config.php 应用配置文件

| |—database.php 数据库配置文件

| |—route.php 路由配置文件

| |—extra 应用扩展配置目录

| |—index index模块配置文件目录

| | |—config.php index模块配置文件

| | |—database.php index模块数据库配置文件

扩展配置文件的文件名（不含后缀）就是配置参数名，并且会和应用配置文件中的参数进行合并。

【配置格式】

ThinkPHP支持多种格式的配置格式，但最终都是解析为PHP数组的方式。

1. PHP数组定义

返回PHP数组的方式是默认的配置定义格式，例如：

//项目配置文件

return [

//默认模块名

‘default\_module’ => ‘index’,

//默认控制器名

‘default\_controller’ => ‘Index’,

//默认操作名

‘default\_action’ => ‘index’,

配置参数名不区分大小写（因为无论大小写定义都会转成小写），新版的建议是使用小写定义配置参数的规范。例如：

//项目配置文件

return [

‘cache’ => [

‘type’ => ‘File’,

‘path’ => CACHE\_PATH,

‘prefix’ => ‘’,

‘expire’ => 0,

],

];

1. 其他配置格式支持

默认方式为PHP数组方式定义配置文件，你可以在入口文件定义CONF\_EXT常量来更改为其他的配置类型：

//更改配置格式为ini格式

define(‘CONFG\_EXT’,’.ini’);

配置后，会自动解析支持的配置类型，包括.ini、.xml、.json和.php在内的格式支持。

5.0.2版本开始支持yaml配置格式支持

ini格式配置示例：

default\_module=Index ;默认模块

default\_controller=index ;默认控制器

default\_action=index ;默认操作

xml格式配置示例：

<config>

<default\_module>Index</default\_module>

<default\_controller>index</default\_controller>

<default\_action>index</default\_action>

</config>

json格式配置示例：

{

“default\_module”:”Index”,

“default\_controller”:”index”,

“default\_action”:”index”

}

1. 二级配置

配置参数支持二级，例如，下面是一个二级配置的设置和读取示例：

$config = [

‘user’ => [

‘type’ => 1,

‘name’ => ‘thinkphp’,

],

‘db’ => [

‘type’ => ‘mysql’,

‘user’ => ‘root’,

‘password’ => ‘’,

],

];

//设置参数

Config::set($config);

//读取二级配置参数

echo Config::get(‘user.type’);

//或者使用助手函数

echo config(‘user.type’);

系统布置二级以上的配置参数读取，需要手动分步骤读取。

有作用域的情况下，仍然支持二级配置的操作。

如果采用其他格式的配置文件的话，二级配置定义方式如下（以ini和xml为例）

[user]

type=1

name=thinkphp

[db]

type=mysql

user=rot

password=’’

标准的xml格式文件定义：

<config>

<user>

<type>1</type>

<name>thinkphp</name>

</user>

<db>

<type>mysql</type>

<user>root</user>

<password>/password>

</db>

</config>

set方法也支持二级配置，例如：

Config::set([

‘type’ => ‘file’,

‘prefix’ => ‘think’

],’cache’);

【配置加载】

在ThinkPHP中，一般来说应用的配置文件是自动加载的，加载的顺序是：

惯例配置->应用配置->扩展配置->场景配置->模块配置->动态配置

以上是配置文件的加载顺序，因为后面的配置会覆盖之前的同名配置（在没有生效的前提下），所以配置的优先顺序从右往左。

下面说明下不同的配置文件的区别和位置：

1. 惯例配置

惯例重于配置是系统遵循的一个重要思想，框架内置有一个惯例配置文件（位于thinkphp/convention.php），按照大多数的使用对常用参数进行了默认配置。所以，对于应用的配置文件，往往只需要配置和惯例配置不同的或者新增的配置参数，如果你完全采用默认配置，甚至可以不需要定义任何配置文件。

建议仔细阅读下系统的惯例配置文件中的相关配置参数，了解下系统默认的配置参数。

1. 应用配置

应用配置文件是应用初始化的时候首先加载的公共配置文件，默认位于application/config.php。

1. 扩展配置

扩展配置文件是由extra\_config\_list配置参数定义的额外的配置文件，默认会加载database和validate两个扩展配置文件。

V5.0.1开始，取消了该配置参数，扩展配置文件直接放入application/extra目录会自动加载。

1. 场景配置

每个应用都可以在不同的情况下设置自己的状态（或者称之为应用场景），并且加载不同的配置文件。

举个例子，你需要在公司和家里分别设置不同的数据库测试环境。那么可以这样处理，在公司环境中，我们在应用配置文件中配置：

‘app\_status’ => ‘office’

那么就会自动加载该状态对应的配置文件（默认位于application/office.php）。

场景配置文件和应用配置文件config.php是一样的定义。

如果我们回家后，我们修改定义为：

‘app\_status’ => ‘home’

那么就会自动加载该状态对应的配置文件（位于application/home.php）。

场景配置文件是可选的。

1. 模块配置

每个模块会自动加载自己的配置文件（位于application/当前模块/config.php）。

模块还可以支持独立的状态配置文件，命名规范为：application/当前模块名/应用状态.php。

模块配置文件是可选的。

如果你的应用的配置文件比较大，想分成几个单独的配置文件或者需要加载额外的配置文件的话，可以考虑采用扩展配置或者动态配置（参考后面的描述）。

加载配置文件

Config::load(‘配置文件名’);

配置文件一般位于APP\_PATH目录下面，如果需要加载其他位置的配置文件，需要使用完整路径，例如：

Config::load(APP\_PATH.’config/config.php’);

系统默认的配置定义格式是PHP返回数组的方式，例如

return [

‘配置参数1’ => ‘配置值’,

‘配置参数2’ => ‘配置值2’,

//. . . 更多配置

];

如果你定义格式是其他格式的话，可以使用parse方法来导入，例如：

Config::parse(APP\_PATH.’my\_config.ini’,’ini’);

Config::parse(APP\_PATH.’myconfig.xml’,’xml’);

parse方法的第一个参数需要传入完整的文件名或者而配置内容。

如果不传入第二个参数的话，系统会根据配置文件名自动识别配置类型，所以下面的写法仍然是支持的：

Config::parse(‘my\_config.php’);

parse方法除了支持读取配置文件外，也支持直接传入配置内容，例如：

$config= ’var1’ = val

var2=val’;

Config::parse($config,’ini’);

支持传入配置文件内容的时候，第二个参数必须显式指定。

标准的ini格式文件定义：

配置参数1=配置值

配置参数2=配置值

标准的xml格式文件定义：

<config>

<var1>val1</varl>

<var2>val2</var2>

</config>

配置类采用驱动方式支持各种不同的配置文件类型，因此可以根据需要随意扩展。

【读取配置】

1. 读取配置参数

设置完配置参数后，就可以使用get方法读取配置了，例如：

echo Config::get(‘配置参数1’);

系统定义了一个助手函数config，以上可以简化为：

echo config(‘配置参数1’);

读取所有的配置参数：

dump(Config:get());

//或者 dump(config());

或者你需要判断是否存在某个设置参数：

Config::has(‘配置参数2’);

//或者config(‘?配置参数2’);

如果需要读取二级配置，可以使用：

echo Config::get(‘配置参数.二级参数’);

echo config(‘配置参数.二级参数’);

【动态配置】

1. 设置配置参数

使用set方法动态设置参数，例如：

Config::set(‘配置参数’,’配置值’);

//或者使用助手函数

config(‘配置参数’,’配置值’);

也可以批量设置，例如：

Config::set([

‘配置参数1’ => ‘配置值’,

‘配置参数2’ => ‘配置值’

]);

//或者使用助手函数

config([

‘配置参数1’ => ‘配置值’,

‘配置参数2’ => ‘配置值’

]);

【独立配置】

1. 独立配置文件

配置文件支持分离（也称为扩展配置），只需要在公共配置文件配置extra\_config\_list参数（V5.0.1版本已经废除该写法）。

例如，不使用独立配置文件的话，数据库配置信息应该是在config.php中配置如下：

/\*数据库设置\*/

‘database’ =>[

//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’ => ‘’,

//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’ => ‘urt8’,

//数据库表前缀

‘prefix’ => ‘’,

//数据库调试模式

‘debug’ => false,

],

如果需要使用独立配置问价的话，则首先在config.php中添加配置：

‘extra\_config\_list’ => [‘database’],

定义之后，数据库配置就可以独立使用database.php，配置内容如下：

/\*数据库设置\*/

return [

//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’ => ‘’,

//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’ => ‘urt8’,

//数据库表前缀

‘prefix’ => ‘’,

//数据库调试模式

‘debug’ => false,

],

如果配置了extra\_config\_list参数，并同时在config.php和database.php文件中都配置的话，则database.php文件的配置会覆盖config.php中的设置。

独立配置文件的参数获取都是二维配置方式，例如，要获取database独立配置文件的type参数，应该是：

Config::get(‘database.type’);

要获取完整的独立配置文件的参数，则使用：

Config::get(‘database’);

1. 自动读取扩展配置

版本要求V5.0.1

5.0.1以上版本支持自动读取扩展配置文件(extra\_config\_list配置参数废弃)，只需要将扩展配置文件放入application/extra目录，即可自动读取。

自动读取的配置文件都是二级配置参数，一级配置名称就是扩展配置的文件名。

模块也可以支持自己的扩展配置文件，只需要放入application/模块名/extra下面就可以自动加载。

系统默认加载的独立配置文件不在此列，包括：

|  |  |
| --- | --- |
| 文件名 | 描述 |
| config | 应用或者模块配置文件 |
| database | 数据库配置文件 |
| tags | 行为定义文件 |
| 场景名 | 应用场景配置文件 |

如果你更改了CONFG\_PATH，那么扩展配置文件目录应该是CONF\_PATH/extra，模块配置目录则变成CONF\_PATH/module/，模块的扩展配置目录则变成CONF\_PATH/module/extra。

【配置作用域】

1. 作用域

配置支持作用域的概念，默认情况下，所用参数都在同一个系统默认作用域下面。如果你的配置参数需要用于不同的项目或者相互隔离；那么就可以使用作用域功能，作用域的作用好比是配置参数的命名空间一样。

//导入my\_config.php中的配置参数，并纳入user作用域

Config::load(‘my\_config.php’,’’,’user’);

//解析并导入my\_config.ini中的配置参数，读入test作用域

Config::parse(‘my\_config.ini’,’ini’,’test’);

//设置user\_type参数，并纳入user作用域

Config::set(‘user\_type’,1,’user’);

//批量设置配置参数，并纳入test作用域

Config::set($config,’test’);

//读取user作用域的user\_type配置参数

echo Config::get(‘user\_type’,’user’);

//读取user作用域下面的所有配置参数

dump(Config::get(‘’,’user’));

dump(config(‘’,null,’user’));//同上

//判断在test作用域下面是否存在user\_type参数

Config::has(‘user\_type’,’test’);

可以使用range方法切换当前配置文件的作用域，例如：

Config::range(‘test’);

【环境变量配置】

ThinkPHP5.0支持环境变量配置。

在开发过程中，可以在应用根目录下面的.evn来模拟环境变量配置，.env文件中的配置参数定义格式采用ini方式，例如：

app\_debug = true

app\_trace = true

如果你的部署环境单独配置了环境变量，那么请删除.env配置文件，避免冲突。

环境变量配置的参数会全部转换为答谢，值为null，no和false等效于””，值为yes和true等效于”1”。

ThinkPHP5.0默认的环境变量前缀是PHP\_，也可以通过改变ENV\_PREFIX常量来重新设置。

注意，环境变量不支持数组参数，如果需要使用数组参数可以，使用下划线分割定义配置参数名：

database\_username = root

database\_password = 123456

或者使用

[database]

username = root

password = 123456

获取环境变量的值可以使用下面的两种方式获取：

Env::get(‘database.username’);

Env::get(‘database.password’);

//同时下面的方式也可以获取

Env::get(‘database\_username’);

Env::get(‘database\_password’);

可以支持默认值，例如：

//获取环境变量，如果不存在则使用默认值root

Env::get(‘database.username’,’root’);

可以直接在应用配置中使用环境变量，例如：

return [

‘hostname’ => Evn::get(‘hostname’,’127.0.0.1’),

];

环境变量中设置的app\_debug和app\_trace参数会自动生效（优先于应用的配置文件），其他参数则必须通过Env::get方法才能读取。

第四章 路由

路由功能由\think\Route类完成

【概述】

由于ThinkPHP5.0默认采用的URL规则是：

<http://server/module/controller/action/param/value/>. . .

路由的作用是简化URL访问地址，并根据定义的路由类型做出正确的解析。

新版的路由功能做了大量的增强，包括：

1. 支持路由到模块的控制器/操作、控制器类的方法、闭包函数和重定向地址，甚至是任何类库的方法；
2. 闭包路由的增强；
3. 规则路由支持全局和局部变量规则定义（正则）；
4. 支持路由到任意层次的控制器；
5. 子域名路由功能改进；
6. 支持路由分组并支持分组参数定义；
7. 增加资源路由和嵌套支持；
8. 支持使用行为或者自定义函数检测路由规则；

ThinkPHP5.0的路由支持三种方式的URL解析规则。

5.0的路由是针对应用而不是针对模块，因此路由的设置也是针对应用下面的所有模块，如果希望不同的模块区分不同的设置（例如某些模块需要关闭路由，某些模块需要强制路由等），需要给该模块增加单独的入口文件，并作如下修改：

//定义项目路径

difine(‘APP\_PATH’,\_DIR\_.’/../application/’);

//加载框架基础文件

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

//绑定当前入口文件到admin模块

\think\Route::bind(‘admin’);

//关闭admin模块的路由

\think\App::route(false);

//执行应用

\think\App::run()->send();

V5.0.21+版本开始，支持了路由解析缓存。

在配置文件中设置开启

//开启路由解析缓存

‘route\_check\_cache’ => true,

【路由模式】

1. 普通模式

关闭路由，完全使用默认的PATH\_INFO方式URL：

‘url\_route\_on’ => false,

路由关闭后，不会解析任何路由规则，采用默认的PATH\_INFO模式访问URL：

<http://serverName/index.php/module/controller/action/param/value/>. . .

但仍然可以通过操作方法的参数绑定、空控制器和空操作等特实现URL地址的简化。

可以设置url\_param\_type配置参数来改变pathinfo模式下面的参数获取方式，默认是按名称成对解析，支持按照顺序解析变量，只需要更改为：

//按照顺序解析变量

‘url\_param\_type’ => 1,

1. 混合模式

开启路由，并使用路由定义+默认PATH\_INFO方式的混合：

‘url\_route\_on’ => true,

‘url\_route\_must’ => false,

该方式下面，只需要对需要定义路由规则的访问地址定义路由规则，其他的仍然按照第一种普通模式的PATH\_INFO模式访问URL。

1. 强制模式

开启路由，并且必须定义路由才能访问：

‘url\_route\_on’ => true,

‘url\_route\_must’ => ture,

这种方式下面必须严格给每一个访问地址定义路由规则（包括首页），否则将跑出异常。

首页的路由规则采用/定义即可，例如下面把网站首页路由输出Hello,world!

Route::get(‘/’,function(){

return ‘Hello,world!’;

});

【路由定义】

1. 注册路由规则

路由注册可以采用方法动态单个和批量注册，也可以直接定义路由定义文件的方式进行集中注册。

1. 动态注册

路由定义采用\think\Route类的rule方法注册，通常是在应用的路由配置文件application/route.php进行注册，格式是：

Route::rule(‘路由表达式’,’路由地址’,’请求类型’,’路由参数（数组）’,’变量规则（数组）’);

例如注册如下路由规则：

use think\Route;

//注册路由到index模块的News控制器的read操作

Route::rule(‘new/id’,’index/News/read’);

我们访问：

<http://serverName/new/5>

ThinkPHP5.0的路由规则定义是从根目录开始，而不是基于模块名的。

会自动路由到：

<http://serverName/index/news/read/id/5>

并且原来的访问地址会自动失效。

路由表达式（第一个参数）支持定义命名标识，例如：

//定义new路由命名标识

Route::rule([‘new’,’new/:id’],’index/News/read’);

注意，路由命名标识必须唯一，定义后可以用于URL的快速生成。

可以在rule方法中指定请求类型，不指定的话默认为任何请求类型，例如：

Route::rule(‘new/:id’,’News/update’,’POST’);

表示定义的路由规则在POST请求下才有效。

请求类型包括：

|  |  |
| --- | --- |
| 类型 | 描述 |
| GET | GET请求 |
| POST | POST请求 |
| PUT | PUT请求 |
| DELETE | DELETE请求 |
| \* | 任何请求类型 |

注意：请求类型参数必须大写。

系统提供了为不同的请求类型定义路由规则的简化方法，例如：

Route::get(‘new/:id’,’News/read’); //定义GET请求路由规则

Route::post(‘new/:id’,’News/read’); //定义POST请求路由规则

Route::put(‘new/:id’,’News/read’); //定义PUT请求路由规则

Route::delete(‘new/:id’,’News/read’); //定义DELETE请求路由规则

Route::any(‘new/:id’,’News/read’); //所有请求都支持的路由规则

如果要定义get和post请求支持的路由规则，也可以用

Route::rule(‘new/:id’,’News/read’,’GET|POST’);

我们也可以批量注册路由规则，例如：

Route::rule([‘new/:id’=>’News/read’,’blog/:name’=>’Blog/detail’]);

Route::get([‘new/:id’=>’News/read’,’blog/:name’=>Blog/detail’]);

Route::post([‘new/:id’=>’News/read’,’blog/:name’=>Blog/detail’]);

多个路由规则后，系统会一次遍历注册过的满足请求类型的路哟规则，一旦匹配到正确的路由规则后则开始调用控制器的操作方法，后续规则就不在检测。

1. 路由表达式

路由表达式统一使用字符串定义，采用规则定义的方式。

正则路由定义功能已经废除，改由变量规则定义完成。

1. 规则表达式

规则表达式通常包含静态地址和动态地址，或者两种地址的结合，例如下面都属于有效的规则表达式：

‘/’ => ‘index’ , //首页访问路由

‘my’ => ‘Member/myinfo’, //静态地址路由

‘blog/:id => ‘Blog/read’, //静态地址和动态地址结合

‘new/:year/:month/:day’ => ‘News/read’ , //静态地址和动态地址结合

‘:user/:blog\_id’ => ‘Blog/read’, //全动态地址

规则表达式的定义以/为参数分隔符（无论你的PATH\_INFO分隔符设置是什么，请确保在路由规则表达式的时候统一使用/进行URL参数分割）。

每个参数中以“:”开头的参数都表示动态变量，并且会自动绑定到操作方法的对应参数。

可选定义

支持对路由参数的可选定义，例如：

‘blog/:year/[:month]’ => ‘Blog/archive’,

[:month]变量用[]包含起来后就表示该变量是由匹配的可选变量。

以上定义路由规则后，下面的URL访问地址和都可以被正确的路由匹配：

<http://serverName/index.php/blog/2015>

<http://serverName/index.php/blog/2015/12>

采用可选变量定义后，之前需要定义两个或者多个路由规则才能处理的情况可以合并为一个路由规则。

可选参数只能放到路由规则的最后，如果在中间使用了可选参数的话，后面的变量都会变成可选参数。

1. 完全匹配

规则匹配检测的时候只是对URL从头开始匹配，只要URL地址包含了定义的路由规则就会匹配成功，如果希望完全匹配，可以在路由表达式最后使用$符号，例如：

‘new/:cate$’ => ‘News/category’,

<http://serverName/index.php/new/info>

会匹配成功，而

<http://serverName/index.php/new/info/2>

则不会匹配成功。

如果是采用

‘new/:cate’ => ‘News/category’,

方式定义的话，则两种方式的URL访问都可以匹配成功。

如果你希望所有的路由定义都是完全匹配的话，可以直接匹配

//开启路由定义的全局完全匹配

‘route\_complete\_match’ => true,

当开启全局完全匹配的时候，如果个别路由不需要使用完全匹配，可以添加路由参数覆盖定义：

Route::rule(‘new/:id’,’News/read’,’GET|POST’,[‘complete\_match’ => false]);

1. 额外参数

在路由跳转的时候支持额外传入参数对（额外参数指的是不在URL里面的参数，隐式传入需要的操作中，有时候能够起到一定的安全防护作用，后面我们会提到）。例如：

‘blog/:id’ => ‘blog/read?status=1&app\_id=5,

上面的路由规则定义中额外参数的传值方式都是等效的。status和app\_id参数都是URL里面不存在的，属于隐式传值，当然并不一定需要用到，只是在需要的时候可以使用。

【批量注册】

如果不希望一个个注册，可以使用批量注册，规则如下：

Route::rule([

‘路由规则1’ => ‘路由地址和参数’,

‘路由规则2’ => ‘[‘路由地址和参数’,’匹配参数（数组）’,’变量规则（数组）’]

. . .

],

如果在外面和规则里面同时传入了匹配参数和变量规则的话，路由规则定义里面的最终生效，但请求类型参数以最外层决定，例如

Route::rule([

‘new/:id’ => ‘News/read’,

‘blog/:id’ => [‘Blog/update’,[‘ext’=>shtml’],[‘id’=>’\d{4}’]],

. . .

],

以上的路由注册，最终blog/:id只会在匹配shtml后缀的访问请求，id变量的规则是\d{4}。

如果不同的请求类型的路由规则是一样的，为了避免数组索引冲突的问题，请使用单独的请求方法定义路由。

同样，我们也可以使用其他几个注册方法进行批量注册。

//批量注册GET路由

Route::get([

‘new/:id’ => ‘News/read’,

‘blog/:id’ => [‘Blog/edit’,[],[‘id’ => ‘\d+’]]

. . .

]);

//效果等同于

Route::rule([

‘new/:id’ => ‘News/read’,

‘blog/:id’ => [‘Blog/edit’,[],[‘id’=>’\d+’]]

. . .

],’’,’GET’);

1. 定义路由配置文件

除了支持动态注册，也可以直接在应用目录下面的route.php的最后通过返回数组的方式直接定义路由规则，内容示例如下：

return [

‘new/:id’ => ‘News/read’

‘blog/:id’ => [‘Blog/update’,[‘method’ => ‘post|put’],[‘id’= ‘\d+’]],

];

路由配置文件定义的路由规则效果和使用any注册路由规则一样。

路由动态注册和配置定义的方式可以共存，例如：

use think\Route;

Route::rule(‘hello/:name’,’index/index/hell0’);

return [

‘new/:id’ => ‘News/read’,

‘blog/:id’ => [‘Blog/update’,[‘method’ => ‘post|put’],[‘id’=’\d+’]],

];

默认情况下，智慧加载一个路由配置文件route.php，如果你需要定义多个路由文件，可以修改route\_config\_file配置参数，例如：

//定义路由配置文件（数组）

‘route\_config\_file’ => [‘route’,’route1’,’route2’],

如果存在相同的路由规则，一样可以参考前面的批量注册方式进行定义。

由于检测机制问题，动态注册的性能比路由配置要高一些，尤其是多种请求类型混合定义的时候。

【变量规则】

ThinkPHP5.0支持在规则路由中为变量用正则的方式指定变量规则，弥补了动态变量无法限制具体的类型问题，并且支持全部规则设置。使用方式如下：

1. 全局变量规则

设置全局变量规则，全部路由有效：

//设置name变量规则（采用正则定义）

Route::pattern(‘name’,’\w+’);

//支持批量添加

Route::pattern([

‘name’ => ‘\w+’,

‘id’ => ‘\d+’,

]);

1. 局部变量规则

局部变量规则，仅在当前路由有效：

//定义GET请求路由规则，并设置name变量规则

Route::get(‘new/:name’,’News/read’,[],[‘name’ => ‘\w+’]);

如果一个变量同时定义了全局规则和局部规则，局部规则会覆盖全局变量的定义。

1. 完整URL规则

如果要对整个URL进行规则检查，可以进行\_url\_变量规则，例如：

//定义GET请求路由规则，并设置完整URL变量规则

Route::get(‘new/:id’,’News/read’,[],[‘\_url\_’ => ‘new\/\w+$’]);

【组合变量】

如果你的路由规则比较特殊，可以在路由定义的时候使用组合变量。

例如：

Route::get(‘item-<name>-<id>’,’product/detail’,[],[‘name’=>’\w+’,’id’=>’\d+’]);

组合变量的优势是路由规则中没有固定的分隔符，可以随意组合需要的变量规则，例如路由规则改成如下一样可以支持：

Route::get(‘item<name><id>’,’product/detail’,[],[‘name’=>’[a-zA-Z]+’,’id’=>’\d+’]);

Route::get(‘item@<name>-<id>’,’product/detail’,[],[‘name’=>’\w+’,’id’=>’\d+’]);

如果需要使用可选变量，则可以使用：

Route::get(‘item-<name><id?>’,’product/detail’,[],[‘name’ => ‘[a-zA-Z]+’,’id’=>’\d+’]);

【路由参数】

路由参数是指可以设置一些路由匹配的条件参数，主要用于验证当前的路由规则是否有效，主要包括：

|  |  |
| --- | --- |
| 参数 | 说明 |
| method | 请求类型检测，支持多个请求类型 |
| ext | URL后缀检测，支持匹配多个后缀 |
| deny\_ext | URL禁止后缀检测，支持匹配多个后缀 |
| https | 检测是否https请求 |
| domain | 域名检测 |
| before\_behavior | 前置行为（检测） |
| after\_behavior | 后置行为（执行） |
| callback | 自定义检测方法 |
| merge\_extra\_vars | 合并额外参数化 |
| bind\_model | 绑定模型（V5.0.1+） |
| cache | 请求缓存（V5.0.1+） |
| param\_depr | 路由参数分隔符（V5.0.2+） |
| ajax | Ajax检测（V5.0.2+） |
| pjax | Pjax检测（V5.0.2+） |

v5.0.3+版本对ajax/pjax/https条件支持false。

这些路由参数可以混合使用，只要有任何一条参数检查不通过，当前路由就不会生效，继续检测后面的路由规则。

1. 请求类型

如果指定请求类型注册路由的话，无需设置method请求类型参数。如果使用了rule或者any方法注册路由，或者使用路由配置定义文件的话，可以单独使用method参数进行请求类型检测。

使用方法：

//检测路由规则仅GET请求有效

Route::any(‘new/:id’,’News/read’,[‘method’=>’get’]);

//检测路由规则仅GET和POST请求有效

Route::any(‘new/:id’,’News/read’,[‘method’=>’get|post’]);

1. URL后缀

//定义GET请求路由规则，并设置URL后缀为html的时候有效

Route::get(‘new/:id’,’News/read’,[‘ext’=>’html’]);

支持匹配多个后缀，例如：

Route::get(‘new/:id’,’News/read’,[‘ext’=>’shtml|html’]);

可以设置禁止访问的URL后缀，例如：

//定义GET请求路由规则，并设置禁止URL后缀为png、jpg、和gif的访问

Route::get(‘new/:id’,’News/read’,[‘deny\_ext’=>’jpg|png|gif’]);

V5.0.7版本以上，ext和deny\_ext参数允许设置为空，跟别表示不允许任何后缀以及必须使用后缀访问。

1. 域名检测

支持使用完整域名或者子域名进行检测，例如：

//Route::get(‘new/:id’,’News/read’,[‘domain’ => ‘news.thinkphp.cn’]);

//子域名检测

Route::get(‘new/:id’,’News/read’,[‘domain’=>’news’]);

1. HTTPS检测

支持检测当前是否HTTPS访问

Route::get(‘new/:id’,’News/read’,[‘https’=>true]);

1. 前置行为检测

支持使用行为对路由进行检测是否匹配，如果行为方法返回false表示当前路由规则无效。

Route::get(‘user/:id’,’index/User/read’,[‘before\_behavior’=>’\app\index\behavior\UserCheck’]);

行为类定义如下：

namespace app\index\behavior;

class UserCheck

{

public function run()

{

if(‘user/0’==request()->url()){

return false;

}

}

}

1. 后置行为执行

可以为某个路由或者分组路由定义后置行为执行，表示当路由匹配成功后，执行的行为，例如：

Route::get(‘user/:id’,’User/read’,[after\_behavior’=>’\app\index\behavior\ReadInfo’]);

其中\app\index\behavior\ReadInit行为类定义如下：

namespace app\index\behavior;

use app\index\model\User;

class ReadInfo{

public function run(){

$id = request()->route(‘id’);

request()->user = User::get($id);

}

}

如果成功匹配到new/:id路由后，就会执行行为类的run方法，参数是路由地址，可以动态改变。

1. Callback检测

也可以支持使用函数检测路由，如果函数返回false则表示当前路由规则无效，例如：

Route::get(‘new/:id’,’News/read’,[‘callback’=>mycheck\_fun’]);

1. 合并额外参数

通常用于完整匹配的情况，如果有额外的参数则合并作为变量值，例如：

Route::get(‘new/:name$’,’News/read’,[‘merge\_extra\_vars’=>true]);

<http://serverName/new/thinkphp/hello>

会被匹配到，并且name变量的值为thinkphp/hello。

1. 配置文件中添加路由参数

如果使用配置文件的话，可以使用：

return [

‘blog/:id’ => [‘Blog/update’,[method’ => ‘post’,’ext’=>’html|shtml’]],

];

1. 路由绑定模型（V5.0.1）

可以在当前路由匹配后绑定模型，后面则同过方法的对象自动注入来获取。

Route::get(‘new/:name$’,’News/read’,[‘bind\_mode’]=>[‘User’,’name’]]);

更多细节可以参考请求->依赖注入

1. 缓存路由请求

可以对当前的路由请求进行缓存处理，例如：

Route::get(‘new/:name$’,’News/read’,[‘cache’=>3600]);

表示对当前路由请求缓存3600秒。

【路由地址】

1. 路由地址定义

|  |  |
| --- | --- |
| 版本 | 新增功能 |
| 5.0.4 | 路由到类和控制器的方法支持传入额外参数 |

路由地址表示定义的路由表达式最终需要路由到的地址以及一些需要的额外参数，支持下面5种方式定义：

|  |  |
| --- | --- |
| 定义方式 | 定义格式 |
| 方式1：路由到模块/控制器 | ‘[模块/控制器/操作]？额外参数1=值1&额外参数2=值2. . . ‘ |
| 方式2：路由到重定向地址 | ‘外部地址’（默认301重定向）或者[‘外部地址’,’重定向代码’] |
| 方式3：路由到控制器的方法 | ‘@[模块/控制器/]操作’ |
| 方式4：路由到类的方法 | ‘\完整的命名空间类::静态方法’或者’\完整的命名空间类@动态方法’ |
| 方式5：路由到闭包函数 | 闭包函数定义（支持参数传入） |

其中方式5我们将会在下一个章节闭包支持中详细描述。

1. 路由到模块/控制器/操作

这是最常用的一种路由方式，把满足条件的路由规则路由到相关的模块、控制器和操作，然后由App类调度相关的操作。

同时会进行模块的初始化操作（包括配置读取、公共文件载入、行为定义载入、语言包载入等等）。

路由地址的格式为：

[模块/控制器/]操作？参数1=值1&参数2=值2. . .

解析规则是从操作开始解析，然后解析控制器，最后解析模块，例如：

//路由到默认或者绑定模块

‘blog/:id’ => ‘blog/read’,

//路由到index模块

‘blog/:id’ => ‘index/blog/read’,

Blog类定义如下：

namespace app\index\controller;

class Blog {

public function read($id){

return ‘read:’.$id;

}

}

路由地址中支持多级控制器，使用下面的方式进行设置：

‘blog/:id’=>’index/group.blog/read’

表示路由到下面的控制器类，

index/controller/group/Blog

Blog类定义如下：

namespace app\index\controller\group;

class Blog {

public function read($id){

return ‘read:’.$id;

}

}

还可以支持路由到动态的模块、控制器或者操作，例如：

//action变量的值作为操作方法传入

‘:action/blog/:id’ => ‘index/blog/:action’

//变量传入index模块的控制器和操作方法

‘:c/:a’ => ‘index/:c/:a’

如果关闭路由功能的话，默认也会按照该规则对URL进行解析调度。

1. 额外参数

在这种方式路由跳转的时候支持额外传入参数对（额外参数指的是不在URL里面的参数，隐式传入需要的操作中，有时候能够起到一定的安全防护作用，后面我们会提到）。例如：

‘blog/:id’ => ‘blog/read?status=1&app\_id=5’,

上面的路由规则定义中额外参数status和app\_id参数都是URL里面不存在的，属于隐式传值，当然并不一定需要用到，只是在需要的时候可以使用。

1. 路由到操作方法

路由地址的格式为：

@[模块/控制器/]操作

这种方式看起来似乎和第一种是一样的，本质的区别是直接执行某个控制器类的方法，而不需要去解析**模块/控制器/操作**这些，同时也不会去初始化模块。

例如，定义如下路由后：

‘blog/:id’=>’@index/blog/read’,

系统会直接执行

Loader::action(‘index/blog/read’);

相当于直接调用\app\index\controller\blog类的read方法。

Blog类定义如下：

namespace app\index\controller;

class Blog {

public function read($id){

return ‘read:’.$id;

}

}

通常这种方式下面，由于没有定义当前模块名、当前控制器名和当前方法名，从而导致视图的默认模板规则失效，所以这种情况下面，如果使用了视图模板渲染，则必须传入明确的参数。

1. 路由到类的方法

路由地址的格式为（动态方法）：

\类的命名空间\类名@方法名

或者（静态方法）：

\类的命名空间\类名::方法名

这种方式更进一步，可以支持执行任何类的方法，而不仅仅是执行控制器的操作方法，例如：

‘blog/:id’ => ‘\app\index\service\Blog@read’,

执行的是\app\index\service\Blog类的read方法。

也支持执行某个静态方法，例如：

‘blog/:id’=>’\app\index\service\Blog::read’,

V5.0.4+版本开始，支持传入额外的参数作为方法的参数调用（用于参数绑定），例如：

‘blog/:id’ => ‘\app\index\service\Blog::read?status=1’,

1. 路由到重定向地址

重定向的外部地址必须以“/”或者http开头的地址。

如果路由地址以“/”或者“http”开头则会认为是一个重定向地址或者外部地址，例如：

‘blog/:id’=>’/blog/read/id/:id’

和

‘blog/:id’ => ‘blog/read’

虽然都是路由到同一个地址，但是前者采用的是301重定向的方式路由跳转，这种方式的好处是URL可以比较随意（包括可以在URL里面传入更多非标准格式的参数），而后者只是支持模块和操作地址。举个例子，如果我们希望avatar/123重定向到/member/avatar/id/123\_small的话，只能使用：

‘avatar/:id’ => ‘/member/avatar/id/:id\_small’

路由地址采用重定向地址的话，如果要引用动态变量，直接使用动态变量即可。

采用重定向到外部地址通常对网站改版后的URL迁移过程非常有用，例如：

‘blog/:id’=>’http://blog.thinkphp.cn/read/:id’

当前网站（可能是<http://thinkphp.cn）的blog/123>地址会直接重定向到<http://blog.thinkphp.cn/read/123>。

【资源路由】