# thinkphp

* 基础

入口文件主要完成：

* 定义框架路径、项目路径（可选）
* 定义调试模式和应用模式（可选）
* 定义系统相关常量（可选）
* 载入框架入口文件（必须）

define('APP\_PATH','./Application/');

require './ThinkPHP/ThinkPHP.php';

如果你改变了项目目录（例如把Application更改为Apps），只需要在入口文件更改**APP\_PATH**常量定义即可：

define('APP\_PATH','./Apps/');

require './ThinkPHP/ThinkPHP.php';

THINK\_PATH 框架目录

APP\_PATH 应用目录

RUNTIME\_PATH 应用运行时目录（可写）

APP\_DEBUG 应用调试模式（默认false）

STORAGE\_TYPE 存储类型（默认为File）

APP\_MODE 应用模式（默认为common）

所有路径常量都必须以“/”结尾

例如，我们可以在入口文件中重新定义相关目录并且开启调试模式：

1. // 定义应用目录
2. define('APP\_PATH','./Apps/');
3. // 定义运行时目录
4. define('RUNTIME\_PATH','./Runtime/');
5. // 开启调试模式
6. define('APP\_DEBUG',True);
7. // 更名框架目录名称，并载入框架入口文件
8. require './Think/ThinkPHP.php';

最终应用目录结构

1. www WEB部署目录（或者子目录）
2. ├─index.php 应用入口文件
3. ├─Apps 应用目录
4. ├─Public 资源文件目录
5. ├─Runtime 运行时目录
6. └─Think 框架目录

为了避免某些服务器开启了目录浏览权限后可以直接在浏览器输入URL地址查看目录，系统默认开启了目录安全文件机制，会在自动生成目录的时候生成空白的index.html文件，当然安全文件的名称可以设置，例如你想给安全文件定义为default.html可以在入口文件中添加：

1. define('DIR\_SECURE\_FILENAME', 'default.html');
2. define('APP\_PATH','./Application/');
3. require './ThinkPHP/ThinkPHP.php';

如果你的环境足够安全，不希望生成目录安全文件，可以在入口文件里面关闭目录安全文件的生成，例如：

1. define('BUILD\_DIR\_SECURE', false);

* 配置

所有配置文件的定义格式均采用返回**PHP数组**的方式

配置参数不区分大小写，建议大写

1. //项目配置文件
2. return array(
3. 'DEFAULT\_MODULE' => 'Index', //默认模块
4. 'URL\_MODEL' => '2', //URL模式
5. 'SESSION\_AUTO\_START' => true, //是否开启session
6. //更多配置参数
7. //...
8. );

可以使用二维数组来配置更多的信息，例如：

1. //项目配置文件
2. return array(
3. 'DEFAULT\_MODULE' => 'Index', //默认模块
4. 'URL\_MODEL' => '2', //URL模式
5. 'SESSION\_AUTO\_START' => true, //是否开启session
6. 'USER\_CONFIG' => array(
7. 'USER\_AUTH' => true,
8. 'USER\_TYPE' => 2,
9. ),
10. //更多配置参数
11. //...
12. );

配置格式：php , yaml , ini , xml , xml , json

通过在入口文件中定义CONF\_EXT常量来指定格式，例如：

1. define('CONF\_EXT','.ini');

定义后，应用的配置文件（包括模块的配置文件）后缀都统一采用.ini。

配置加载顺序：



惯例配置->应用配置->模式配置->调试配置->状态配置->模块配置->扩展配置->动态配置

读取配置：**C('参数名称')**（Config） 获取已经设置的参数值，如：

1. $model = C('URL\_MODEL');

可以读取到系统的调试模式的设置值，同样，由于配置参数不区分大小写，因此

1. $model = C('url\_model');

是等效的，但是建议使用大写方式的规范。

如果url\_model尚未存在设置，则返回NULL。

1. C('my\_config',null,'default\_config');

如果my\_config尚未设置的话，则返回default\_config。

C方法也可以用于读取二维配置：

1. //获取用户配置中的用户类型设置
2. C('USER\_CONFIG.USER\_TYPE');

C(‘’)读取全部配置项

C(‘name’)读取配置项名称为name的配置值

C(‘name’,null,’default’)读取 配置项name的配置值，如未配置返回default

C(‘parent.sub’)读取二维数组配置

配置的名称不区分大小写，但二维数组配置读取的参数，parent是严格区分大小写

动态配置：C('参数名称','新的参数值');

例如，我们需要动态改变数据缓存的有效期的话，可以使用

1. // 动态改变缓存有效期
2. C('DATA\_CACHE\_TIME',60);
3. // 获取已经设置的参数值
4. C('USER\_CONFIG.USER\_TYPE');
5. //设置新的值
6. C('USER\_CONFIG.USER\_TYPE',1);

C(‘name’,$value)设置配置项名称为name的配置值

C($config\_array)批量配置

$config\_array中 Key为配置项名称，对应Value为配置值

扩展配置：

通过设置LOAD\_EXT\_CONFIG配置项的值来加载当前模块的自定义配置文件

1. // 加载扩展配置文件
2. 'LOAD\_EXT\_CONFIG' => 'user,db',

假设扩展配置文件user.php 和db.php分别用于用户配置和数据库配置，这样做的好处是哪怕以后关闭调试模式，你修改db配置文件后依然会自动生效。

如果在应用公共设置文件中配置的话，那么会自动加载应用公共配置目录下面的配置文件Application/Common/Conf/user.php和Application/Common/Conf/db.php。

如果在模块（假设是Home模块）的配置文件中配置的话，则会自动加载模块目录下面的配置文件 Application/Home/Conf/user.php 和 Application/Home/Conf/db.php。

默认情况下，扩展配置文件中的设置参数会并入项目配置文件中。也就是默认都是一级配置参数，例如user.php中的配置参数如下：

1. <?php
2. //用户配置文件
3. return array(
4. 'USER\_TYPE' => 2, //用户类型
5. 'USER\_AUTH\_ID' => 10, //用户认证ID
6. 'USER\_AUTH\_TYPE' => 2, //用户认证模式
7. );

那么，最终获取用户参数的方式是：

1. C('USER\_AUTH\_ID');

如果配置文件改成：

1. // 加载扩展配置文件
2. 'LOAD\_EXT\_CONFIG' => array('USER'=>'user','DB'=>'db'),

则最终获取用户参数的方式改成：

1. C('USER.USER\_AUTH\_ID');

批量配置

C配置方法支持批量配置，例如：

1. $config = array('WEB\_SITE\_TITLE'=>'ThinkPHP','WEB\_SITE\_DESCRIPTION'=>'开源PHP框架');
2. C($config);

$config数组中的配置参数会合并到现有的全局配置中。

我们可以通过这种方式读取数据库中的配置参数，例如：

1. // 读取数据库中的配置（假设有一个config表用于保存配置参数）
2. $config = M('Config')->getField('name,value');
3. // config是一个关联数组 键值就是配置参数 值就是配置值
4. // 例如： array('config1'=>'val1','config2'=>'val2',...)
5. C($config); // 合并配置参数到全局配置

合并之后，我们就可以和前面读取普通配置参数一样，读取数据库中的配置参数了，当然也可以动态改变。

1. // 读取合并到全局配置中的数据库中的配置参数
2. C('CONFIG1');
3. // 动态改变配置参数（当前请求有效，不会自动保存到数据库）
4. C('CONFIG2','VALUE\_NEW');

* 架构

模块化设计

一个典型的URL访问规则是（我们以默认的PATHINFO模式为例说明，当然也可以支持普通的URL模式）：

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

模块化设计的思想下面模块是最重要的部分，模块其实是一个包含配置文件、函数文件和MVC文件（目录）的集合。

公共模块

Common模块是一个特殊的模块，是应用的公共模块，访问所有的模块之前都会首先加载公共模块下面的配置文件（Conf/config.php）和公共函数文件（Common/function.php）。但Common模块本身不能通过URL直接访问，公共模块的其他文件则可以被其他模块继承或者调用。

公共模块的位置可以通过**COMMON\_PATH**常量改变，我们可以在入口文件中重新定义**COMMON\_PATH**如下：

1. define('COMMON\_PATH','./Common/');
2. define('APP\_PATH','./Application/');
3. require './ThinkPHP/ThinkPHP.php';

其应用目录结构变成：

1. www WEB部署目录（或者子目录）
2. ├─index.php 入口文件
3. ├─README.md README文件
4. ├─Common 应用公共模块目录
5. ├─Application 应用模块目录
6. ├─Public 应用资源文件目录
7. └─ThinkPHP 框架目录

定义之后，Application目录下面就不再需要Common目录了。

自动生成模块目录

需要生成一个Admin模块用于后台应用，在应用入口文件中定义如下：

1. // 绑定Admin模块到当前入口文件
2. define('BIND\_MODULE','Admin');
3. define('APP\_PATH','./Application/');
4. require './ThinkPHP/ThinkPHP.php';

然后访问URL地址

1. http://serverName/index.php

就会生成Admin模块的目录，并生成一个默认的控制器类Admin\Controller\IndexController。 如果需要生成更多的控制器类，可以定义BUILD\_CONTROLLER\_LIST常量，例如：

1. // 绑定Admin模块到当前入口文件
2. define('BIND\_MODULE','Admin');
3. define('BUILD\_CONTROLLER\_LIST','Index,User,Menu');
4. define('APP\_PATH','./Application/');
5. require './ThinkPHP/ThinkPHP.php';

访问后会自动生成三个指定的控制器类：

1. Admin\Controller\IndexController
2. Admin\Controller\UserController
3. Admin\Controller\MenuController

同样，也可以定义BUILD\_MODEL\_LIST支持生成多个模型类：

1. // 绑定Admin模块到当前入口文件
2. define('BIND\_MODULE','Admin');
3. define('BUILD\_CONTROLLER\_LIST','Index,User,Menu');
4. define('BUILD\_MODEL\_LIST','User,Menu');
5. define('APP\_PATH','./Application/');
6. require './ThinkPHP/ThinkPHP.php';

访问会自动生成模型类：

1. Admin\Model\UserModel
2. Admin\Model\MenuModel

禁止访问模块

通常情况下无需配置模块列表即可访问，但可以配置禁止访问的模块列表（用于被其他模块调用或者不开放访问），默认配置中是禁止访问Common模块和Runtime模块（Runtime目录是默认的运行时目录），我们可以增加其他的禁止访问模块列表：

1. // 设置禁止访问的模块列表
2. 'MODULE\_DENY\_LIST' => array('Common','Runtime','Api'),

设置后，Api模块不能通过URL直接访问，事实上，可能我们只是在该模块下面放置一些公共的接口文件，因此都是内部调用即可。

## 设置访问列表

如果你的应用下面模块比较少，还可以设置允许访问列表和默认模块，这样可以简化默认模块的URL访问。

1. 'MODULE\_ALLOW\_LIST' => array('Home','Admin','User'),
2. 'DEFAULT\_MODULE' => 'Home',

设置之后，除了Home、Admin和User模块之外的模块都不能被直接访问，并且Home模块是默认访问模块（可以不出现在URL地址）。

## 单模块设计

如果你的应用够简单，那么也许仅仅用一个模块就可以完成，那么可以直接设置：

1. // 关闭多模块访问
2. 'MULTI\_MODULE' => false,
3. 'DEFAULT\_MODULE' => 'Home',

一旦关闭多模块访问后，就只能访问默认模块（这里设置的是Home）。

## 多入口设计

可以给相同的应用及模块设置多个入口，不同的入口文件可以设置不同的应用模式或者绑定模块。

例如，我们在index.php文件的同级目录新增一个home.php入口文件，并绑定Home模块：

1. // 绑定Home模块到当前入口文件
2. define('BIND\_MODULE','Home');
3. define('APP\_PATH','./Application/');
4. require './ThinkPHP/ThinkPHP.php';

绑定模块后，原来的访问地址

1. http://serverName/index.php/Home/Index/index

就变成

1. http://serverName/home.php/Index/index

同样的方式，我们也可以在入口文件中绑定控制器，例如：

1. define('BIND\_MODULE', 'Home'); // 绑定Home模块到当前入口文件
2. define('BIND\_CONTROLLER','Index'); // 绑定Index控制器到当前入口文件
3. define('APP\_PATH','./Application/');
4. require './ThinkPHP/ThinkPHP.php';

绑定模块和控制器后，原来的访问地址：

1. http://serverName/index.php/Home/Index/index

就变成：

1. http://serverName/home.php/index

URL模式

入口文件是应用的单一入口，对应用的所有请求都定向到应用入口文件，系统会从URL参数中解析当前请求的模块、控制器和操作：

1. http://serverName/index.php/模块/控制器/操作

## URL大小写

ThinkPHP框架的URL是区分大小写（主要是针对模块、控制器和操作名，不包括应用参数）的

框架内置了一个配置参数用于解决URL大小写的问题，如下：

1. 'URL\_CASE\_INSENSITIVE' => true,

当URL\_CASE\_INSENSITIVE设置为true的时候表示URL地址不区分大小写，这个也是框架在部署模式下面的默认设置。

这种URL模式就是系统默认的PATHINFO模式，不同的URL模式获取模块和操作的方法不同，ThinkPHP支持的URL模式有四种：**普通模式、PATHINFO、REWRITE和兼容模式**，可以设置**URL\_MODEL**参数改变URL模式。

| **URL模式** | **URL\_MODEL设置** |
| --- | --- |
| 普通模式 | 0 |
| PATHINFO模式 | 1 |
| REWRITE模式 | 2 |
| 兼容模式 | 3 |

普通模式

传统的GET传参方式来指定当前访问的模块和操作，如：http://localhost/?m=home&c=user&a=login&var=value

m参数表示模块，c参数表示控制器，a参数表示操作（当然这些参数都是可以配置的），后面的表示其他GET参数。

如果默认的变量设置和你的应用变量有冲突的话，你需要重新设置系统配置，例如改成下面的：

1. 'VAR\_MODULE' => 'module', // 默认模块获取变量
2. 'VAR\_CONTROLLER' => 'controller', // 默认控制器获取变量
3. 'VAR\_ACTION' => 'action', // 默认操作获取变量

上面的访问地址则变成： http://localhost/?module=home&controller=user&action=login&var=value

注意，VAR\_MODULE只能在应用配置文件中设置，其他参数可以则也可以在模块配置中设置

PATHINFO模式

是系统的默认URL模式，提供了最好的SEO支持，系统内部已经做了环境的兼容处理，所以能够支持大多数的主机环境。对应上面的URL模式，PATHINFO模式下面的URL访问地址是：http://localhost/index.php/home/user/login/var/value/

PATHINFO地址的前三个参数分别表示模块/控制器/操作。

不过，PATHINFO模式下面，依然可以采用普通URL模式的参数方式，例如：http://localhost/index.php/home/user/login?var=value 依然是有效的

PATHINFO模式下面，URL是可定制的，例如，通过下面的配置：

1. // 更改PATHINFO参数分隔符
2. 'URL\_PATHINFO\_DEPR'=>'-',

我们还可以支持下面的URL访问： http://localhost/index.php/home-user-login-var-value

REWRITE模式

是在PATHINFO模式的基础上添加了重写规则的支持，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。

如果是Apache则需要在入口文件的同级添加.htaccess文件，内容如下：

1. <IfModule mod\_rewrite.c>
2. RewriteEngine on
3. RewriteCond %{REQUEST\_FILENAME} !-d
4. RewriteCond %{REQUEST\_FILENAME} !-f
5. RewriteRule ^(.\*)$ index.php/$1 [QSA,PT,L]
6. </IfModule>

接下来，就可以用下面的URL地址访问了： http://localhost/home/user/login/var/value

兼容模式

**兼容模式**是用于不支持PATHINFO的特殊环境，URL地址是： http://localhost/?s=/home/user/login/var/value

可以更改兼容模式变量的名称定义，例如：

1. 'VAR\_PATHINFO' => 'path'

PATHINFO参数分隔符对兼容模式依然有效，例如：

1. // 更改PATHINFO参数分隔符
2. 'URL\_PATHINFO\_DEPR'=>'-',

使用以上配置的话，URL访问地址可以变成： http://localhost/?path=/home-user-login-var-value

兼容模式配合Web服务器重写规则的定义，可以达到和REWRITE模式一样的URL效果。

例如，我们在Apache下面的话，.htaccess文件改成如下内容：

1. <IfModule mod\_rewrite.c>
2. RewriteEngine on
3. RewriteCond %{REQUEST\_FILENAME} !-d
4. RewriteCond %{REQUEST\_FILENAME} !-f
5. RewriteRule ^(.\*)$ index.php?s=/$1 [QSA,PT,L]
6. </IfModule>

就可以和REWRITE模式一样访问下面的URL地址访问了：

 http://localhost/home/user/login/var/value

多层MVC

## 模型（Model）层

例如在某个项目设计中需要区分数据层、逻辑层、服务层等不同的模型层，我们可以在模块目录下面创建Model、Logic和Service目录，把对用户表的所有模型操作分成三层：

1. 数据层：Model/UserModel 用于定义数据相关的自动验证和自动完成和数据存取接口
2. 逻辑层：Logic/UserLogic 用于定义用户相关的业务逻辑
3. 服务层：Service/UserService 用于定义用户相关的服务接口等

而这三个模型操作类统一都继承Model类即可，例如： 数据层：Home/Model/UserModel.class.php

1. namespace Home\Model;
2. use Think\Model;
3. class UserModel extends Model{
4. }

逻辑层：Home/Logic/UserLogic.class.php

1. namespace Home\Logic;
2. use Think\Model;
3. class UserLogic extends Model{
4. }

服务层：Home/Service/UserService.class.php

1. namespace Home\Service;
2. use Think\Model;
3. class UserService extends Model{
4. }

在调用的时候，我们也可以用内置的D方法很方便的调用：

1. D('User') //实例化UserModel
2. D('User','Logic') //实例化UserLogic
3. D('User','Service') //实例化UserService

默认的模型层是Model，我们也可以更改设置，例如：

1. 'DEFAULT\_M\_LAYER' => 'Logic', // 更改默认的模型层名称为Logic

更改之后，实例化的时候需要改成：

1. D('User') //实例化UserLogic
2. D('User','Model') //实例化UserModel
3. D('User','Service') //实例化UserService

对模型层的分层划分是很灵活的，开发人员可以根据项目的需要自由定义和增加模型分层，你也完全可以只使用Model层。

## 视图（View）层

视图层由模板和模板引擎组成，在模板中可以直接使用PHP代码，模板引擎的设计会在后面讲述，通过驱动也可以支持其他第三方的模板引擎。视图的多层可以简单的通过目录（也就是模板主题）区分，例如：

1. View/default/User/add.html
2. View/blue/User/add.html

复杂一点的多层视图还可以更进一步，采用不同的视图目录来完成，例如：

1. view 普通视图层目录
2. mobile 手机端访问视图层目录

这样做的好处是每个不同的视图层都可以支持不同的模板主题功能。

默认的视图层是View目录，我们可以调整设置如下：

1. 'DEFAULT\_V\_LAYER' => 'Mobile', // 默认的视图层名称更改为Mobile

非默认视图层目录的模板获取需要使用T函数，后面会讲到。

## 控制器（Controller）层

ThinkPHP的控制器层由核心控制器和业务控制器组成，核心控制器由系统内部的App类完成，负责应用（包括模块、控制器和操作）的调度控制，包括HTTP请求拦截和转发、加载配置等。业务控制器则由用户定义的控制器类完成。多层业务控制器的实现原理和模型的分层类似，例如业务控制器和事件控制器：

1. Controller/UserController //用于用户的业务逻辑控制和调度
2. Event/UserEvent //用于用户的事件响应操作

访问控制器 Home/Controller/UserController.class.php 定义如下：

1. namespace Home\Controller;
2. use Think\Controller;
3. class UserController extends Controller{
4. }

事件控制器 Home/Event/UserEvent.class.php 定义如下：

1. namespace Home\Event;
2. use Think\Controller;
3. class UserEvent extends Controller{
4. }

UserController负责外部交互响应，通过URL请求响应，例如 http://serverName/User/index,而 UserEvent 负责内部的事件响应，并且只能在内部调用： A('User','Event');

默认的访问控制器层是Controller，我们可以调整设置如下：

1. 'DEFAULT\_C\_LAYER' => 'Event', // 默认的控制器层名称改为Event

所以是和外部隔离的。多层控制器的划分也不是强制的，可以根据应用的需要自由分层。控制器分层里面可以根据需要调用分层模型，也可以调用不同的分层视图（主题）。

在MVC三层中，ThinkPHP并不依赖M或者V，甚至可以只有C或者只有V，这个在ThinkPHP的设计里面是一个很重要的用户体验设计，用户只需要定义视图，在没有C的情况下也能自动识别。

CBD模式

**CBD（核心Core+行为Behavior+驱动Driver）**架构模式

命名空间

自动加载

应用模式

默认情况下的应用模式是普通模式（common），如果要采用其他的应用模式（当然，前提是已经有定义），必须在入口文件中定义，设置APP\_MODE常量即可，例如：

1. // 定义存储类型和应用模式为SAE（用于支持SAE平台）
2. define('STORAGE\_TYPE','sae');
3. define('APP\_MODE','sae');
4. define('APP\_PATH','./Application/');
5. require './ThinkPHP/ThinkPHP.php';

应用模式的一个典型应用是对分布式平台的支持，对不同的平台定义不同的应用模式就可以支持。

每个入口文件仅能定义一个应用模式，所以，如果需要对相同的应用模块设置不同的应用模式访问，就可以通过增加入口文件的方式来解决。

每个应用模式可以定义单独的配置文件，一般是config\_模式名称，例如，sae模式下面可以定义：

1. // 应用配置文件
2. Application/Common/Conf/config\_sae.php

或者

1. // 模块配置文件
2. Application/Home/Conf/config\_sae.php

config\_sae配置文件只会sae模式下面加载，如果不是sae模式则不会加载。

项目编译

系统流程

* 路由

路由定义

要使用路由功能，前提是你的URL支持PATH\_INFO（或者兼容URL模式也可以，采用普通URL模式的情况下不支持路由功能），并且在应用（或者模块）配置文件中开启路由：

1. // 开启路由
2. 'URL\_ROUTER\_ON' => true,

然后就是配置路由规则了，在模块的配置文件中使用**URL\_ROUTE\_RULES**参数进行配置，配置格式是一个数组，每个元素都代表一个路由规则，例如：

1. 'URL\_ROUTE\_RULES'=>array(
2. 'news/:year/:month/:day' => array('News/archive', 'status=1'),
3. 'news/:id' => 'News/read',
4. 'news/read/:id' => '/news/:1',
5. ),

系统会按定义的顺序依次匹配路由规则，一旦匹配到的话，就会定位到路由定义中的控制器和操作方法去执行（可以传入其他的参数），并且后面的规则不会继续匹配。

## 路由定义

路由规则的定义格式为： **'路由表达式'=>'路由地址和传入参数'**

或者：**array('路由表达式','路由地址','传入参数')**

##### 路由表达式

路由表达式包括规则路由和正则路由的定义表达式，只能使用字符串。

| **表达式** | **示例** |
| --- | --- |
| 正则表达式 | /^blog\/(\d+)$/ |
| 规则表达式 | blog/:id |

## 路由参数

当路由地址采用数组方式定义的时候，还可以传入额外的路由参数。

这些参数的作用是限制前面定义的路由规则的生效条件。

##### 限制URL后缀

例如：

1. 'blog/:id'=>array('blog/read','status=1&app\_id=5',array('ext'=>'html')),

就可以限制html后缀访问该路由规则才能生效。

##### 限制请求类型

例如：

1. 'blog/:id'=>array('blog/read','status=1&app\_id=5',array('method'=>'get')),

就限制了只有GET请求该路由规则才能生效。

##### 自定义检测

支持自定义检测，例如： 例如：

1. 'blog/:id'=>array('blog/read','status=1&app\_id=5',array('callback'=>'checkFun')),

就可以自定义checkFun函数来检测是否生效，如果函数返回false则表示不生效。

规则路由

## 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：

1. 'my' => 'Member/myinfo', // 静态地址路由
2. 'blog/:id' => 'Blog/read', // 静态地址和动态地址结合
3. 'new/:year/:month/:day'=>'News/read', // 静态地址和动态地址结合
4. ':user/:blog\_id' =>'Blog/read',// 全动态地址

规则表达式的定义始终以“/”为参数分割符，不受URL\_PATHINFO\_DEPR设置的影响

每个参数中以“:”开头的参数都表示动态参数，并且会自动对应一个GET参数，例如:id表示该处匹配到的参数可以使用$\_GET['id']方式获取，:year、 :month 、:day 则分别对应$\_GET['year']、 $\_GET['month'] 和 $\_GET['day']。

##### 数字约束

支持对变量的类型检测，但仅仅支持数字类型的约束定义，例如

1. 'blog/:id\d'=>'Blog/read',

表示只会匹配数字参数，如果你需要更加多的变量类型检测，请使用正则表达式定义来解决。

##### 函数支持

可以支持对路由变量的函数过滤，例如：

1. 'blog/:id\d|md5'=>'Blog/read',

表示对匹配到的id变量进行md5处理，也就是说，实际传入read操作方法的$\_GET['id'] 其实是 md5($\_GET['id'])。

##### 可选定义

支持对路由参数的可选定义，例如：

1. 'blog/:year\d/[:month\d]'=>'Blog/archive',

[:month\d]变量用[ ]包含起来后就表示该变量是路由匹配的可选变量。

以上定义路由规则后，下面的URL访问地址都可以被正确的路由匹配：

1. http://serverName/index.php/Home/blog/2013
2. http://serverName/index.php/Home/blog/2013/12

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。

正则路由

静态路由

闭包支持

实例说明

* 控制器

ThinkPHP的控制器是一个类，而操作则是控制器类的一个**公共方法**。

1. 'ACTION\_SUFFIX' => 'Action', // 操作方法后缀

设置操作方法的后缀为Action，这样，控制器的操作方法定义调整为：

1. <?php
2. namespace Home\Controller;
3. use Think\Controller;
4. class IndexController extends Controller {
5. public function listAction(){
6. echo 'list';
7. }
8. public function helloAction(){
9. echo 'hello';
10. }
11. public function testAction(){
12. echo 'test';
13. }
14. }

操作方法的后缀设置只是影响控制器类的定义，对URL访问没有影响。

## 多层控制器

ThinkPHP的控制器支持多层和多级，多层指的是控制器可以分层，例如除了默认的Controller控制器层（我们可以称之为访问控制器），还可以添加事件控制器（层），例如：

1. ├─Controller 访问控制器
2. │ ├─UserController.class.php
3. │ ├─BlogController.class.php
4. │ ...
5. ├─Event 事件控制器
6. │ ├─UserEvent.class.php
7. │ ├─BlogEvent.class.php
8. │ ...

访问控制器的名称是通过**DEFAULT\_C\_LAYER**设置的，默认是Controller。

访问控制器负责外部交互响应，通过URL请求响应，例如 http://serverName/Home/User/index,而事件控制器负责内部的事件响应，并且只能在内部调用，所以是和外部隔离的。

多层控制器的划分可以根据项目的需要自由分层。

如果是定义其他的控制器层，则不一定必须要继承系统的Controller类或其子类，通常需要输出模版的时候才需要继承Controller类。例如：

1. <?php
2. namespace Home\Event;
3. class UserEvent {
4. public function login(){
5. echo 'login event';
6. }
7. public function logout(){
8. echo 'logout event';
9. }
10. }

UserEvent事件控制器位于 Home/Event/UserEvent.class.php 。

## 多级控制器

多级控制器是指控制器可以通过子目录把某个控制器层分组存放，首先需要设置控制器的分级层次，例如，我们设置2级目录的控制器层：

1. 'CONTROLLER\_LEVEL' => 2,

控制器文件的位置放置如下：

1. ├─Controller 访问控制器
2. │ ├─User User分级（组）
3. │ │ ├─UserTypeController.class.php
4. │ │ ├─UserAuthController.class.php
5. │ ...
6. │ ├─Admin Admin分级（组）
7. │ │ ├─UserController.class.php
8. │ │ ├─ConfigController.class.php
9. │ ...

多级控制器中的命名空间需要这样定义：

1. <?php
2. namespace Home\Controller\Admin;
3. use Think\Controller;
4. class IndexController extends Controller {
5. public function hello(){
6. echo 'hello';
7. }
8. public function test(){
9. echo 'test';
10. }
11. }

然后就可以通过URL地址访问：

1. http://serverName/Home/User/UserType
2. http://serverName/Home/Admin/User

如果希望简化URL地址中的模块地址，可以参考 [模块部署](http://document.thinkphp.cn/manual_3_2.html#module_deploy)

## 实例化控制器

访问控制器的实例化通常是自动完成的，系统会根据URL地址解析出访问的控制器名称自动实例化，并且调用相关的操作方法。

如果你需要跨控制器调用的话，则可以单独实例化：

1. // 实例化Home模块的User控制器
2. $User = new \Home\Controller\UserController();
3. // 实例化Admin模块的Blog控制器
4. $Blog = new \Admin\Controller\BlogController();

系统为上面的控制器实例化提供了一个快捷调用方法A，上面的代码可以简化为：

1. // 假设当前模块是Home模块
2. $User = A('User');
3. $Blog = A('Admin/Blog');

默认情况下，A方法实例化的是默认控制器层（Controller），如果你要实例化其他的分层控制器的话，可以使用：

1. // 假设当前模块是Home模块
2. // 实例化Event控制器
3. $User = A('User','Event');
4. $Blog = A('Admin/Blog','Event');

上面的代码等效于：

1. // 实例化Home模块的User事件控制器
2. $User = new \Home\Event\UserEvent();
3. // 实例化Admin模块的Blog事件控制器
4. $Blog = new \Admin\Event\BlogEvent();

前置操作和后置操作

Action参数绑定

参数绑定是通过直接绑定URL地址中的变量作为操作方法的参数，可以简化方法的定义甚至路由的解析。

要启用参数绑定功能，首先确保你开启了URL\_PARAMS\_BIND设置：

1. 'URL\_PARAMS\_BIND' => true, // URL变量绑定到操作方法作为参数

参数绑定有两种方式：**按照变量名绑定和按照变量顺序绑定**。

伪静态

通过设置URL\_HTML\_SUFFIX参数随意在URL的最后增加你想要的静态后缀，而不会影响当前操作的正常执行。例如，我们设置

1. 'URL\_HTML\_SUFFIX'=>'shtml'

的话，我们可以把下面的URL http://serverName/Home/Blog/read/id/1 变成http://serverName/Home/Blog/read/id/1.shtml

设置伪静态后缀为空，

1. 'URL\_HTML\_SUFFIX'=>''

则可以支持所有的静态后缀，并且会记录当前的伪静态后缀到常量 **\_\_EXT\_\_** ，但不会影响正常的页面访问。

例如：

1. http://serverName/Home/blog/3.html
2. http://serverName/Home/blog/3.shtml
3. http://serverName/Home/blog/3.xml
4. http://serverName/Home/blog/3.pdf

都可以正常访问，如果要获取当前的伪静态后缀，通过常量 **\_\_EXT\_\_** 获取即可。

如果希望支持多个伪静态后缀，可以直接设置如下：

1. // 多个伪静态后缀设置 用|分割
2. 'URL\_HTML\_SUFFIX' => 'html|shtml|xml'

那么，当访问 http://serverName/Home/blog/3.pdf 的时候会报系统错误。

可以设置禁止访问的URL后缀，例如：

1. 'URL\_DENY\_SUFFIX' => 'pdf|ico|png|gif|jpg', // URL禁止访问的后缀设置

如果访问 http://serverName/Home/blog/3.pdf 就会直接返回404错误。

**注意：**URL\_DENY\_SUFFIX的优先级比URL\_HTML\_SUFFIX要高。

URL大小写

提供了一个不区分URL大小写的解决方案，可以通过配置简单实现。

只要在项目配置中，增加：

1. 'URL\_CASE\_INSENSITIVE' =>true

配置好后，即使是在Linux环境下面，也可以实现URL访问不再区分大小写了。

1. http://serverName/index.php/Home/Index/index
2. // 将等效于
3. http://serverName/index.php/home/index/index

这里需要注意一个地方，一旦开启了不区分URL大小写后，如果我们要访问类似UserTypeController的控制器，那么正确的URL访问应该是：

1. // 正确的访问地址
2. http://serverName/index.php/home/user\_type/index
3. // 错误的访问地址（linux环境下）
4. http://serverName/index.php/home/usertype/index

利用系统提供的U方法可以为你自动生成相关的URL地址。

如果设置

1. 'URL\_CASE\_INSENSITIVE' =>false

的话，URL就又变成： http://serverName/index.php/Home/UserType/add

注意：URL不区分大小写并不会改变系统的命名规范，并且只有按照系统的命名规范后才能正确的实现URL不区分大小写。

URL生成

## 定义规则

U方法的定义规则如下（方括号内参数根据实际应用决定）：

**U('地址表达式',['参数'],['伪静态后缀'],['显示域名'])**

##### 地址表达式

地址表达式的格式定义如下：

1. [模块/控制器/操作#锚点@域名]?参数1=值1&参数2=值2...

如果不定义模块的话 就表示当前模块名称，下面是一些简单的例子：

1. U('User/add') // 生成User控制器的add操作的URL地址
2. U('Blog/read?id=1') // 生成Blog控制器的read操作 并且id为1的URL地址
3. U('Admin/User/select') // 生成Admin模块的User控制器的select操作的URL地址

##### 参数

U方法的第二个参数支持数组和字符串两种定义方式，如果只是字符串方式的参数可以在第一个参数中定义，例如：

1. U('Blog/cate',array('cate\_id'=>1,'status'=>1))
2. U('Blog/cate','cate\_id=1&status=1')
3. U('Blog/cate?cate\_id=1&status=1')

三种方式是等效的，都是生成Blog控制器的cate操作 并且cate\_id为1 status为1的URL地址。

但是不允许使用下面的定义方式来传参数

1. U('Blog/cate/cate\_id/1/status/1');

##### 伪静态后缀

U函数会自动识别当前配置的伪静态后缀，如果你需要指定后缀生成URL地址的话，可以显式传入，例如：

1. U('Blog/cate','cate\_id=1&status=1','xml');

## 自动识别

根据项目的不同URL设置，同样的U方法调用可以智能地对应产生不同的URL地址效果，例如针对：

1. U（'Blog/read?id=1'）;

这个定义为例。

如果当前URL设置为普通模式的话，最后生成的URL地址是：

1. http://serverName/index.php?m=Blog&a=read&id=1

如果当前URL设置为PATHINFO模式的话，同样的方法最后生成的URL地址是：

1. http://serverName/index.php/Home/Blog/read/id/1

如果当前URL设置为REWRITE模式的话，同样的方法最后生成的URL地址是：

1. http://serverName/Home/Blog/read/id/1

如果当前URL设置为REWRITE模式，并且设置了伪静态后缀为.html的话，同样的方法最后生成的URL地址是：

1. http://serverName/Home/Blog/read/id/1.html

如果开启了URL\_CASE\_INSENSITIVE，则会统一生成小写的URL地址。

## 域名支持

如果你的应用涉及到多个子域名的操作地址，那么也可以在U方法里面指定需要生成地址的域名，例如：

1. U('Blog/read@blog.thinkphp.cn','id=1');

@后面传入需要指定的域名即可。

系统会自动判断当前是否SSL协议，生成https://。

此外，U方法的第4个参数如果设置为true，表示自动识别当前的域名，并且会自动根据子域名部署设置APP\_SUB\_DOMAIN\_DEPLOY和APP\_SUB\_DOMAIN\_RULES自动匹配生成当前地址的子域名。

## 锚点支持

U函数可以直接生成URL地址中的锚点，例如：

1. U('Blog/read#comment?id=1');

生成的URL地址可能是：

1. http://serverName/index.php/Home/Blog/read/id/1#comment

AJAX返回

ThinkPHP可以很好的支持AJAX请求，系统的\Think\Controller类提供了ajaxReturn方法用于AJAX调用后返回数据给客户端。并且支持JSON、JSONP、XML和EVAL四种方式给客户端接受数据，并且支持配置其他方式的数据格式返回。

ajaxReturn方法调用示例：

1. $data = 'ok';
2. $this->ajaxReturn($data);

支持返回数组数据：

1. $data['status'] = 1;
2. $data['content'] = 'content';
3. $this->ajaxReturn($data);

默认配置采用JSON格式返回数据（通过配置DEFAULT\_AJAX\_RETURN进行设置），我们可以指定格式返回，例如：

1. // 指定XML格式返回数据
2. $data['status'] = 1;
3. $data['content'] = 'content';
4. $this->ajaxReturn($data,'xml');

返回数据data可以支持字符串、数字和数组、对象，返回客户端的时候根据不同的返回格式进行编码后传输。如果是JSON/JSONP格式，会自动编码成JSON字符串，如果是XML方式，会自动编码成XML字符串，如果是EVAL方式的话，只会输出字符串data数据。

JSON和JSONP虽然只有一个字母的差别，但其实他们根本不是一回事儿：JSON是一种数据交换格式，而JSONP是一种非官方跨域数据交互协议。一个是描述信息的格式，一个是信息传递的约定方法。

默认的JSONP格式返回的处理方法是jsonpReturn，如果你采用不同的方法，可以设置：

1. 'DEFAULT\_JSONP\_HANDLER' => 'myJsonpReturn', // 默认JSONP格式返回的处理方法

或者直接在页面中用callback参数来指定。

除了上面四种返回类型外，我们还可以通过行为扩展来增加其他类型的支持，只需要对ajax\_return标签位进行行为绑定即可。

跳转和重定向

控制器的redirect方法和redirect函数的区别在于前者是用URL规则定义跳转地址，后者是一个纯粹的URL地址。

输入变量

## 获取变量

虽然你仍然可以在开发过程中使用传统方式获取各种系统变量，例如：

1. $id = $\_GET['id']; // 获取get变量
2. $name = $\_POST['name']; // 获取post变量
3. $value = $\_SESSION['var']; // 获取session变量
4. $name = $\_COOKIE['name']; // 获取cookie变量
5. $file = $\_SERVER['PHP\_SELF']; // 获取server变量

但是我们不建议直接使用传统方式获取，因为没有统一的安全处理机制，后期如果调整的话，改起来会比较麻烦。所以，更好的方式是在框架中统一使用I函数进行变量获取和过滤。

I方法是ThinkPHP用于更加方便和安全的获取系统输入变量，可以用于任何地方，用法格式如下：

###### I('变量类型.变量名/修饰符',['默认值'],['过滤方法'],['额外数据源'])

变量类型是指请求方式或者输入类型，包括：

| 变量类型 | 含义 |
| --- | --- |
| get | 获取GET参数 |
| post | 获取POST参数 |
| param | 自动判断请求类型获取GET、POST或者PUT参数 |
| request | 获取REQUEST 参数 |
| put | 获取PUT 参数 |
| session | 获取 $\_SESSION 参数 |
| cookie | 获取 $\_COOKIE 参数 |
| server | 获取 $\_SERVER 参数 |
| globals | 获取 $GLOBALS参数 |
| path | 获取 PATHINFO模式的URL参数**（3.2.2新增）** |
| data | 获取 其他类型的参数，需要配合额外数据源参数**（3.2.2新增）** |

我们以GET变量类型为例，说明下I方法的使用：

1. echo I('get.id'); // 相当于 $\_GET['id']
2. echo I('get.name'); // 相当于 $\_GET['name']

支持默认值：

1. echo I('get.id',0); // 如果不存在$\_GET['id'] 则返回0
2. echo I('get.name',''); // 如果不存在$\_GET['name'] 则返回空字符串

采用方法过滤：

1. // 采用htmlspecialchars方法对$\_GET['name'] 进行过滤，如果不存在则返回空字符串
2. echo I('get.name','','htmlspecialchars');

支持直接获取整个变量类型，例如：

1. // 获取整个$\_GET 数组
2. I('get.');

用同样的方式，我们可以获取post或者其他输入类型的变量，例如：

1. I('post.name','','htmlspecialchars'); // 采用htmlspecialchars方法对$\_POST['name'] 进行过滤，如果不存在则返回空字符串
2. I('session.user\_id',0); // 获取$\_SESSION['user\_id'] 如果不存在则默认为0
3. I('cookie.'); // 获取整个 $\_COOKIE 数组
4. I('server.REQUEST\_METHOD'); // 获取 $\_SERVER['REQUEST\_METHOD']

param变量类型是框架特有的支持自动判断当前请求类型的变量获取方式，例如：

echo I('param.id');

如果当前请求类型是GET，那么等效于 $\_GET['id']，如果当前请求类型是POST或者PUT，那么相当于获取 $\_POST['id'] 或者 PUT参数id。

由于param类型是I函数默认获取的变量类型，因此事实上param变量类型的写法可以简化为：

1. I('id'); // 等同于 I('param.id')
2. I('name'); // 等同于 I('param.name')

3.2.2新增了path和data两个变量类型，用法如下：

path类型变量可以用于获取URL参数（必须是PATHINFO模式参数有效，无论是GET还是POST方式都有效），例如： 当前访问URL地址是 http://serverName/index.php/New/2013/06/01

那么我们可以通过

1. echo I('path.1'); // 输出2013
2. echo I('path.2'); // 输出06
3. echo I('path.3'); // 输出01

data类型变量可以用于获取不支持的变量类型的读取，例如：

1. I('data.file1','','',$\_FILES);

## 变量过滤

如果你没有在调用I函数的时候指定过滤方法的话，系统会采用默认的过滤机制（由DEFAULT\_FILTER配置），事实上，该参数的默认设置是：

1. // 系统默认的变量过滤机制
2. 'DEFAULT\_FILTER' => 'htmlspecialchars'

也就说，I方法的所有获取变量如果没有设置过滤方法的话都会进行htmlspecialchars过滤，那么：

1. // 等同于 htmlspecialchars($\_GET['name'])
2. I('get.name');

同样，该参数也可以设置支持多个过滤，例如：

1. 'DEFAULT\_FILTER' => 'strip\_tags,htmlspecialchars'

设置后，我们在使用：

1. // 等同于 htmlspecialchars(strip\_tags($\_GET['name']))
2. I('get.name');

如果我们在使用I方法的时候 指定了过滤方法，那么就会忽略DEFAULT\_FILTER的设置，例如：

1. // 等同于 strip\_tags($\_GET['name'])
2. echo I('get.name','','strip\_tags');

I方法的第三个参数如果传入函数名，则表示调用该函数对变量进行过滤并返回（在变量是数组的情况下自动使用array\_map进行过滤处理），否则会调用PHP内置的filter\_var方法进行过滤处理，例如：

1. I('post.email','',FILTER\_VALIDATE\_EMAIL);

表示 会对$\_POST['email'] 进行 格式验证，如果不符合要求的话，返回空字符串。 （关于更多的验证格式，可以参考 官方手册的filter\_var用法。） 或者可以用下面的字符标识方式：

1. I('post.email','','email');

可以支持的过滤名称必须是filter\_list方法中的有效值（不同的服务器环境可能有所不同），可能支持的包括：

1. int
2. boolean
3. float
4. validate\_regexp
5. validate\_url
6. validate\_email
7. validate\_ip
8. string
9. stripped
10. encoded
11. special\_chars
12. unsafe\_raw
13. email
14. url
15. number\_int
16. number\_float
17. magic\_quotes
18. callback

3.2.3版本开始支持进行正则匹配过滤，例如：

1. // 采用正则表达式进行变量过滤
2. I('get.name','','/^[A-Za-z]+$/');
3. I('get.id',0,'/^\d+$/');

如果正则匹配不通过的话，则返回默认值。

在有些特殊的情况下，我们不希望进行任何过滤，即使**DEFAULT\_FILTER**已经有所设置，可以使用：

1. // 下面两种方式都不采用任何过滤方法
2. I('get.name','','');
3. I('get.id','',false);

一旦过滤参数设置为空字符串或者false，即表示不再进行任何的过滤。

## 变量修饰符

3.2.3版本开始，I函数支持对变量使用修饰符功能，可以更好的过滤变量。

用法如下： **I('变量类型.变量名/修饰符');**

例如：

1. I('get.id/d');
2. I('post.name/s');
3. I('post.ids/a');

可以使用的修饰符包括：

| **修饰符** | **作用** |
| --- | --- |
| s | 强制转换为字符串类型 |
| d | 强制转换为整形类型 |
| b | 强制转换为布尔类型 |
| a | 强制转换为数组类型 |
| f | 强制转换为浮点类型 |

请求类型

## 判断请求类型

在很多情况下面，我们需要判断当前操作的请求类型是GET 、POST 、PUT或 DELETE，一方面可以针对请求类型作出不同的逻辑处理，另外一方面有些情况下面需要验证安全性，过滤不安全的请求。 系统内置了一些常量用于判断请求类型，包括：

| **常量** | **说明** |
| --- | --- |
| IS\_GET | 判断是否是GET方式提交 |
| IS\_POST | 判断是否是POST方式提交 |
| IS\_PUT | 判断是否是PUT方式提交 |
| IS\_DELETE | 判断是否是DELETE方式提交 |
| IS\_AJAX | 判断是否是AJAX提交 |
| REQUEST\_METHOD | 当前提交类型 |

使用举例如下：

1. class UserController extends Controller{
2. public function update(){
3. if (IS\_POST){
4. $User = M('User');
5. $User->create();
6. $User->save();
7. $this->success('保存完成');
8. }else{
9. $this->error('非法请求');
10. }
11. }
12. }

需要注意的是，如果使用的是ThinkAjax或者自己写的Ajax类库的话，需要在表单里面添加一个隐藏域，告诉后台属于ajax方式提交，默认的隐藏域名称是ajax（可以通过VAR\_AJAX\_SUBMIT配置），如果是JQUERY类库的话，则无需添加任何隐藏域即可自动判断。

空操作

空操作是指系统在找不到请求的操作方法的时候，会定位到空操作（\_empty）方法来执行，利用这个机制，我们可以实现错误页面和一些URL的优化。

例如，下面我们用空操作功能来实现一个城市切换的功能。 我们只需要给CityController类定义一个\_empty（空操作）方法：

1. <?php
2. namespace Home\Controller;
3. use Think\Controller;
4. class CityController extends Controller{
5. public function \_empty($name){
6. //把所有城市的操作解析到city方法
7. $this->city($name);
8. }
9. //注意 city方法 本身是 protected 方法
10. protected function city($name){
11. //和$name这个城市相关的处理
12. echo '当前城市' . $name;
13. }
14. }

接下来，我们就可以在浏览器里面输入

1. http://serverName/index.php/Home/City/beijing/
2. http://serverName/index.php/Home/City/shanghai/
3. http://serverName/index.php/Home/City/shenzhen/

由于City控制器并没有定义beijing、shanghai或者shenzhen操作方法，因此系统会定位到空操作方法 \_empty中去解析，\_empty方法的参数就是当前URL里面的操作名，因此会看到依次输出的结果是：

1. 当前城市:beijing
2. 当前城市:shanghai
3. 当前城市:shenzhen

注意：空操作方法仅在你的控制器类继承系统的Think\Controller类才有效，否则需要自己定义 \_\_call 来实现。

空控制器

空控制器的概念是指当系统找不到请求的控制器名称的时候，系统会尝试定位空控制器(EmptyController)，利用这个机制我们可以用来定制错误页面和进行URL的优化。

现在我们把前面的需求进一步，把URL由原来的

1. http://serverName/index.php/Home/City/shanghai/

变成

1. http://serverName/index.php/Home/shanghai/

这样更加简单的方式，如果按照传统的模式，我们必须给每个城市定义一个控制器类，然后在每个控制器类的index方法里面进行处理。可是如果使用空控制器功能，这个问题就可以迎刃而解了。

我们可以给项目定义一个EmptyController类

1. <?php
2. namespace Home\Controller;
3. use Think\Controller;
4. class EmptyController extends Controller{
5. public function index(){
6. //根据当前控制器名来判断要执行那个城市的操作
7. $cityName = CONTROLLER\_NAME;
8. $this->city($cityName);
9. }
10. //注意 city方法 本身是 protected 方法
11. protected function city($name){
12. //和$name这个城市相关的处理
13. echo '当前城市' . $name;
14. }
15. }

接下来，我们就可以在浏览器里面输入

1. http://serverName/index.php/Home/beijing/
2. http://serverName/index.php/Home/shanghai/
3. http://serverName/index.php/Home/shenzhen/

由于系统并不存在beijing、shanghai或者shenzhen控制器，因此会定位到空控制器（EmptyController）去执行，会看到依次输出的结果是：

1. 当前城市:beijing
2. 当前城市:shanghai
3. 当前城市:shenzhen

如果你的默认控制器层不是Controller的话，空控制器也会随之改变，例如默认的控制器层是Action：

1. 'DEFAULT\_C\_LAYER' => 'Action', // 默认的控制器层名称

那么可能你的空控制器定义就变成了EmptyAction类：

1. <?php
2. namespace Home\Action;
3. use Think\Controller;
4. class EmptyAction extends Controller{
5. public function index(){
6. //根据当前控制器名来判断要执行那个城市的操作
7. $cityName = CONTROLLER\_NAME;
8. $this->city($cityName);
9. }
10. //注意 city方法 本身是 protected 方法
11. protected function city($name){
12. //和$name这个城市相关的处理
13. echo '当前城市' . $name;
14. }
15. }

空控制器和空操作还可以同时使用，用以完成更加复杂的操作。

插件控制器

操作绑定到类

* 控制器

1. 就是一个php类，继承thinkphp的核心类Controller
2. public外部访问
3. ACTION\_SUFFIX操作方法定义修改（操作方法后缀）
4. 空控制器、空操作方法
5. 操作绑定到类
6. 调用四种方式：自动定位、new实例化、A函数、R函数

URL模式

四种：通过URL\_MODEL配置

0：普通模式 ( http://domain/index.php?m=m&c=c&a=a )

1：PathInfo ( http://domain/index.php?m/c/a )

2：ReWrite ( http://domain/m/c/a )

3：兼容模式

PathInfo和Rewrite需要运行环境的支持

URL动态生成

使用U函数可以动态生成模式匹配的URL:

U(‘地址表达式’,[‘参数’],[‘伪静态后缀’],[‘显示域名’])

-地址表达式：[模块/控制器/操作#锚点@域名]?参数1=值1&参数2=值2…

-第二个参数支持数组和字符串两种定义方式

-类unix系统对URL大小写是敏感的，通过配置URL\_CASE\_INSENSITIVE实现不区分大小写的URL访问

伪静态

为了满足更好的SEO效果，使用：

URL\_HTML\_SUFFIX 配置伪静态后缀

URL\_DENY\_SUFFIX 配置禁止访问的URL后缀

本地Apache服务器实现Rewrite模式

Rewrite是通过URL重写隐藏应用的入口文件index.php:

Apache开启mod\_rewrite.so模块

AllowOverride 设置为 All

定义分布式配置文件 htaccess

重启Apache服务后生效

实现更接近生产环境的URL

通过Apache开启mod\_vhost\_alias.so模块

配置httpd-vhosts.conf

添加hosts主机头实现转发

重启Apache服务后生效

跳转与AJAX数据返回

以下几种跳转和响应：

redirect重定向

参数（地址表达式 附加参数 跳转时间s 提示信息）

success成功提示跳转

参数（提示信息 跳转地址 跳转时间s）

error错误提示跳转

参数（提示信息 跳转地址 跳转时间s）

ajaxReturn实现Ajax数据返回

参数（返回数据 数据格式）

控制器定义了六个常量，用来快速判断请求的类型：

IS\_GET、IS\_POST、IS\_PUT、IS\_DELETE、IS\_AJAX、REQUEST\_METHOD（枚举）

不使用jquery的话，AJAX请求要添加识别参数：通过VAR\_AJAX\_SUBMIT配置

获取输入变量

以下几种方式：

$\_GET/$\_POST/$\_SERVER(不安全)

I函数（Input单词的首字母），方便实现输入变量读取，并自动进行数据安全过滤

Input函数：

I(‘变量类型.变量名’,[‘变量不存在默认值’],[‘过滤方法’],[‘额外数据源‘])

API补充

Action参数绑定是通过直接绑定URL地址中的变量作为操作方法的参数，可以简化方法的定义甚至路由的解析。

Action参数绑定功能默认是开启的，原理是把URL中的参数（不包括模块、控制器和操作名）和操作方法中的参数进行绑定。

* 视图
* 视图

1.基本结构

*//入口文件绑定模块文件，新建一个模块  
define*(**'BIND\_MODULE'**,**'Views'**);  
*//关闭目录安全文件生成  
define*(**'BUILD\_DIR\_SECURE'**,**false**);

知识点：

1.1.默认视图层（DEFAULT\_V\_LAYER）

1.2.模板文件后缀（TMPL\_TEMPLATE\_SUFFIX）

1.3.连接符（TMPL\_FILE\_DEPR）

1.4.视图目录（VIEW\_PATH）

**'DEFAULT\_V\_LAYER'** => **'Template'**,  
**'TMPL\_TEMPLATE\_SUFFIX'** => **'.jike'**,  
**'TMPL\_FILE\_DEPR'** => **'\_'**,  
**'VIEW\_PATH'** => **'./Theme/Views/'**,

2.视图变量赋值和渲染(动态网页基础)

知识点：

* 1. 使用控制器的assign方法为视图变量赋值：

assign(‘name’,$value)

* 1. 渲染视图（控制器display、fetch和show方法）
  2. display(‘[模板文件]’[,’字符串编码’][,’输出类型’])
  3. fetch(‘[模板文件]’[,’字符串编码’][,’输出类型’])
  4. show(‘渲染内容’[,’字符串编码’][,’输出类型’])
  5. T函数：T([资源://][模块@][主题/][控制器/]操作,[视图分层])

3.主题

知识点：

3.1 设置默认主题（DEFAULT\_THEME）

3.2 控制器theme方法

3.3 多主题切换

3.4 差异主题（TMPL\_LOAD\_DEFAULTTHEME）

* 模板（上）

1. 变量输出

知识点：

1.1自定义变量输出：{$value}

1.2 默认值输出：{$value|default=’默认值’}

1.3系统变量($Think.server/env/post/get/cookie/session/request)

1.4使用运算符需要注意：数学运算，点语法失效

1.5模板解析定界符(TMPL\_L\_DELIM、TMPL\_R\_DELIM)

1. 使用函数

知识点：

2.1变量是第一个参数：{$value|function[=?,?,?]}，例如{$value|trim}

2.1变量不是第一个参数：{$value|str\_replace=?,?,###}}

2.3嵌套使用多个函数：{$value|func1|func2|func3}，优先级从左到右

2.4回归原生PHP：{:expression}

1. 包含文件和模板替换

知识点：

3.1包含文件：<include file=”模板表达式/文件路径[,…]” key=value/> [key]

如：<**include file="Public/navbar" home="主页"**/>

<**li class="active"**><**a href="#"**>[home]<**span class="src-only"**>(current)</**span**></**a**></**li**>

3.2导入资源文件：import、load、js、css

如<**import file="Js.bootstrap" type="js"** />  
<**load href="\_\_PUBLIC\_\_/js/jquery.js"**/>  
<**css href="\_\_PUBLIC\_\_/css/bootstrap.css"** />  
<**js href="\_\_CDN\_\_/js/bootstrap.js"** />

3.3 模板替换：TMPL\_PARSE\_STRING

如*// 模板替换规则定义* **'TMPL\_PARSE\_STRING'**=>**array**(  
 **'\_\_CDN\_\_'**=>**'./Cdn'**,  
 ),

3.4一些默认的替换规则：

\_\_ROOT\_\_，\_\_APP\_\_，\_\_MODULE\_\_，\_\_CONTROLLER\_\_，\_\_ACTION\_\_，\_\_SELF\_\_，\_\_PUBLIC\_\_

* 模型
* 模型：定义与实例化

1模型的定义

知识点：

1.1模型是以PHP类的方式定义，继承自相应的模型类

1.2约定模型名称是不包括前缀的数据表名称，UserModel对应jike\_user数据表

1.3模型的属性：tablePrefix、tableName、trueTableName、dbName、connection

1.4实例化：new UserModel()、M(‘User’)、D(‘User’)和控模型

如：*//new*$user\_model=**new** \Models\Model\UserModel();  
  
*//M*$user\_m\_model=M(**'User'**);  
  
*//D*$user\_d\_model=D(**'User'**);  
  
*//控模型*$empty\_model=**new** \Think\Model();  
$empty\_m\_model=M();  
$empty\_d\_model=D();

2模型数据库信息配置

知识点：

2.1数据库信息配置

2.2切换数据库：db方法

2.3分布式数据库：DB\_DEPLOY\_TYPE

2.4读写分离：DB\_RW\_SEPARATE、DB\_MASTER\_NUM、DB\_SLAVE\_NO

如：*/\* 数据库设置 \*/***'DB\_TYPE'** => **'mysql'**, *// 数据库类型***'DB\_HOST'** => **'localhost'**, *// 服务器地址***'DB\_NAME'** => **'jike'**, *// 数据库名***'DB\_USER'** => **'root'**, *// 用户名***'DB\_PWD'** => **'root'**, *// 密码***'DB\_PORT'** => **'3306'**, *// 端口***'DB\_PREFIX'** => **'jike\_'**, *// 数据库表前缀***'DB\_CHARSET'** => **'utf8'**, *// 数据库编码默认采用utf8***'DB\_DEPLOY\_TYPE'** => 0, *// 数据库部署方式:0 集中式(单一服务器),1 分布式(主从服务器)***'DB\_RW\_SEPARATE'** => **false**, *// 数据库读写是否分离 主从式有效***'DB\_MASTER\_NUM'** => 1, *// 读写分离后 主服务器数量***'DB\_SLAVE\_NO'** => **''**, *// 指定从服务器序号***'DB\_2'**=>**array**(  
 **'DB\_TYPE'** => **'mysql'**, *// 数据库类型* **'DB\_HOST'** => **'localhost'**, *// 服务器地址* **'DB\_NAME'** => **'mysql'**, *// 数据库名* **'DB\_USER'** => **'root'**, *// 用户名* **'DB\_PWD'** => **'root'**, *// 密码* **'DB\_PORT'** => **'3306'**, *// 端口*),

3模型的字段定义

知识点：

3.1字段缓存

为什么要缓存字段信息：数据库中字段信息存入缓存

保存在哪里：缓存目录下的Data/\_fields/

以什么形式保存：数据库名称[.表前缀]+模型名称（小写）.php

如何关闭：DB\_FIELDS\_CACHE

获取模型的字段信息：getDbFields方法

如：**'DB\_FIELDS\_CACHE'** => **true**, *//启用字段缓存*

3.2字段定义

模型的fields属性

使用\_type定义字段数据类型

通过pk属性定义表的主键

* 模型：数据库操作

1. CURD操作

知识点：

* 1. Create（新增insert），使用add方法，注意并不是create方法
  2. Update（更新），使用save方法
  3. Read（读取），使用find、select
  4. Delete（删除），使用delete方法

1. 连贯操作

实例化的模型类每调用一次连贯操作方法，便会修改本身的属性值，最后以一个终结方法来结束连贯操作，对于模型来说就是进行数据库操作然后返回结果。

如：$topicModel->order(‘view desc’)->limit(10)->select()

order和limit是连贯方法，执行它们会返回模型本身，而select属于终结方法，执行数据库读取操作，返回符合条件的数据

连贯方法：

约束类：where、table、field(查询字段)、order、limit、page(分页)、group、having、index

多表辅助类：alias(指定别名)、join、union(合并查询)、relation(关联查询)

工具辅助类：distinct、lock、token(设置表单令牌)、fetchSql(返回sql语句)、cache、strict(长置检查字段)、comment(添加注释)、data

终结方法：find、select、getField、add、addAll、save、delete

1. 模型中定义命名范围

命名范围作用

如何定义：模型的\_scope属性

通过模型的scope方法调用

如：**protected $\_scope**=**array**(  
 **'latest'**=>**array**(  
 **'order'**=>**'create\_time desc'**,  
 **'limit'**=>10  
 ),  
 **'vip'**=>**array**(  
 **'where'**=>**array**(  
 **'is\_vip'**=>1  
 ),  
 ),  
);  
  
*//获取最近注册的10个用户***public function** getLatestUsers(){  
 *//连贯操作方法  
 //$result=$this->order('create\_time desc')->limit(10)->select();* $result=$this->scope(**'latest'**,**array**(**'limit'**=>20))->select();  
 **return** $result;  
}

*//获取最近注册的10个已经通过VIP认证的用户***public function** getLatestVipUsers(){  
  
 *//$conditon=array(  
 // 'is\_vip'=>1  
 //);  
 //$result=$this->where($conditon)->order('create\_time desc')->limit(10)->select();  
 //$result=$this->scope('vip')->scope('latest')->select();* $result=$this->vip()->scope(**'latest'**)->select();  
 **return** $result;  
}  
  
*//获取10个VIP认证用户，要求2015年1月1日之前注册,并且按照积分score从高到低排序***public function** getSomeUsers(){  
 $condition=**array**(  
 **'is\_vip'**=>1  
 );  
 $result=$this  
 ->where($condition)  
 ->where(**'create\_time <'**.*strtotime*(2015-1-1))  
 ->order(**'score desc'**)  
 ->limit(10)  
 ->select();  
 **return** $result;

1. 使用查询语言

查询条件的三种定义方式：字符串、数组和对象

表达式查询

快捷查询

区间查询

组合查询：\_string、\_query、\_complex

统计查询

SQL查询

动态查询

如：*//查询条件的三种定义方式***public function** sql\_1(){  
 *//字符串* $conditionString=**'status=1 OR is\_vip=1'**;  
  
 *//数组* $conditionArray=**array**(  
 **'status'**=>1,  
 **'is\_vip'**=>1,  
 **'\_logic'**=>**'OR'** );  
  
 *//对象* $conditonClass=**new** \stdClass();  
 $conditonClass->**status**=1;  
 $conditonClass->**is\_vip**=1;  
 $conditonClass->**\_logic**=**'OR'**;  
}  
  
*//表达式查询***public function** sql\_2(){  
 *//eq: =  
 //neq: <>  
 //gt: >  
 //egt: >=  
 //lt: <  
 //elt: <=  
 //between  
 //not between  
 //in  
 //not in* $condition=**array**(  
 **'username'**=>**array**(**'eq'**,**'linda'**),  
 **'email'**=>**array**(**'jike'**,**'%jike%'**),  
 **'score'**=>**array**(**'egt'**,1000),  
 **'age'**=>**array**(**'between'**,**'20,35'**),  
 **'id'**=>**array**(**'not in'**,**'1,6,99'**),  
 *//表达式，支持sql语法函数* **'score'**=>**array**(**'exp'**=>**'score+2'**)  
 );  
  
}  
  
*//快捷查询***public function** sql\_3(){  
 $condition=**array**(  
 **'username|email|mobile'**=>I(**'post.account'**),  
 **'password'**=>*md5*(I(**'post.password'**))  
 );  
}  
  
*//区间查询***public function** sql\_4(){  
 $condition=**array**(  
 **'age'**=>**array**(  
 **array**(**'egt'**,18),  
 **array**(**'elt'**,35)  
 ),  
 **'score'**=>**array**(  
 **array**(**'elt'**,99),  
 **array**(**'egt'**,1000),  
 **'or'** )  
 );  
}  
  
*//组合查询***public function** sql\_5(){  
 $subCondition=**array**(  
 **'status'**=>1,  
 **'age'**=>**array**(**'gt'**,18),  
 **'\_logic'**=>**'and'** );  
 $condition=**array**(  
 **'\_string'**=>**'status=1 AND age>18'**,  
 **'\_query'**=>**'status=1&age>18&\_logic=and'**,  
 **'\_complex'**=>$subCondition  
 );  
}  
  
*//统计查询***public function** sql\_6(){  
 $this->count();  
 $this->max(**'age'**);  
 $this->min(**'age'**);  
 $this->avg(**'age'**);  
 $this->sum(**'age'**);  
}  
  
*//sql查询***public function** sql\_7(){  
 M()->query(**'select \* form jike\_user where status=1'**);  
 M()->excute(**'update jike\_user set status=0 where id=1'**);  
}  
  
*//动态查询***public function** sql\_8(){  
 $this->getFieldByUsername(**'linda'**,**'email'**);  
 $this->getByUsername(**'linda'**);  
}

1. ActiveRecord模式

什么是ActiveRecord

ORM对象关系映射（Object Relational Mapping）用于实现面向对象编程语言里不同类型系统的数据之间的转换

基于ActiveRecord的CURD操作

如：*//新增用户***private function** createUser(){  
  
 $userAttribute=**array**(  
 **'username'**=>**'tom'**,  
 **'password'**=>*md5*(**'111'**),  
 **'email'**=>**'tom@jike.com'**,  
 **'create\_time'**=>*time*(),  
 **'status'**=>1  
 );  
 D(**'User'**)->add($userAttribute);  
}  
*//activeRecord模式的新增用户***private function** createUserAR(){  
 $user=D(**'User'**);  
  
 $user->**username**=**'tom'**;  
 $user->**password**=*md5*(**'111'**);  
 $user->**email**=**'tom@jike.com'**;  
 $user->**create\_time**=*time*();  
 $user->**status**=1;  
  
 $user->add();  
}

*//更新用户***private function** updateUserStatus($userId){  
  
 $userUpdateAttribute=**array**(  
 **'id'**=>$userId,  
 **'status'**=>0  
 );  
 D(**'User'**)->save($userUpdateAttribute);  
}  
*//activeRecord模式的更新用户***private function** updateUserStatusAR($userId){  
 $user=D(**'User'**);  
  
 $user->**id**=$userId;  
 $user->**status**=0;  
  
 $user->save();  
}

1. 定义字段映射

为什么要映射字段（防攻击）

模型的\_map属性（调用create方法时自动映射）

通过READ\_DATA\_MAP配置开启读取数据时的字段映射

模型的parseFieldsMap方法

* 模板（下）

1. 模板标签

内置标签：include、import、volist、foreach、for、switch、compare、present、empty、defined、if/else

输出php代码：PHP标签

原样输出：literal标签

1. 标签库

标签库的加载：taglib标签，标签预加载配置（TAGLIB\_PRE\_LOAD）

配置内置标签库：TAGLIB\_BUILD\_IN

1. 模板布局和继承

模板布局的三种方式：

全局模式：

LAYOUT\_ON

LAYOUT\_NAME

TMPL\_LAYOUT\_ITEM

标签模式：

<layout name=”布局名称“ replace=”替换字符串”/>

layout方法：

layout(bool)开启和关闭

模板的继承

extend标签：

模板继承标签

<extend name=”模板表达式/模板地址”/>

block标签：

<block name=”区块名称”/>

block重载：如果字模板中不进行重载则沿用，如果置空则删除

* 模型数据验证和填充

1. 数据验证和填充的概念和实现方式

B/S架构应用的数据安全风险主要集中在数据交互环节；

不要相信用户提交的任何数据，进行过滤和验证是必须的；

对于按照自定义规则生成的数据，如订单编号、积分变更，只能在服务器端进行填充；

模型作为数据库的业务抽象层，非常适合用于做数据的统一处理；

1. 实现数据自动验证

自动验证的四大要素：

对象：验证那个对象的数据

规则：按照什么规则验证

时机：何时、何种情况下验证

反馈：如何反馈验证结果

静态定义和动态定义

批量验证

1. 实现数据自动填充