PHP-ThinkPHP

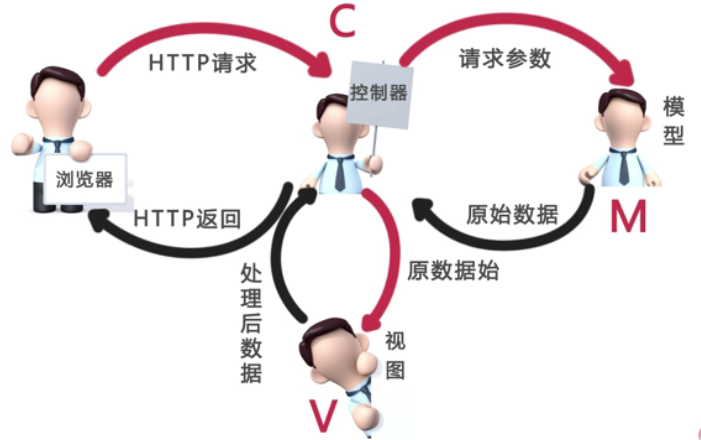
一 MVC

MVC是一种设计模式，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(Model)、视图（View）、控制器（Controller），它们各自处理任务。

模型(Model):处理数据和业务逻辑

视图（View）：处理样式，通过布局向用户展示数据

控制器（Controller）:接收用户请求，并调用相应的模型处理



二 框架

框架是程序结构代码的集合，而不是业务逻辑代码。集合中包含了许多类、函数、功能类包。这个集合是按照一定标准组成的功能体系。体系包含很多设计模式，比如MVC等。

三 ThinkPHP——单一入口框架

在一个网站中，所有请求都指向一个脚本文件。

好处：项目结构规范，控制灵活，更加安全。

四 目录结构

1.ThinkPHP目录

1.Common:核心公共函数目录

2.Conf:核心配置目录

3.Lang:核心语言包目录

4.Library:核心类库目录

1.Think:核心Think类包目录

2.Behavior:行为类库目录

3.Org:org类库包目录

4.Vendor:第三方类库目录

5.Mode:框架应用模式目录

6.Tpl:系统模板目录

7.LICENSE.txt:框架授权协议文件

8.index.php:框架入口文件

2.Application目录

1.Common:应用公共模块

1.Common:应用公共函数目录

2.Conf:应用公共配置目录

2.Home

1.Conf:模块配置文件目录

2.Common:模块函数公共目录

3.Controller:模块控制器目录

4.Model:模块模型目录

5.View:模块视图目录

3.Runtime:运行时目录

1.Cache:模板缓存目录

2.Data:数据目录

3.Logs:日志目录

4.Temp:缓存目录

4.index.html

5.test.php

五 安全机制

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.修改安全文件名:可以生成多个文件

Define(“DIR\_SECURE\_FILENAME”,”default.html,index.html”);

2.在安全文件中添加

Define(“DIR\_SECURE\_CONTENT”,“string”);

3.关闭生成默认文件

Define(“BUILD\_DIR\_SECURE”,“false”);

六 模块

我理解为:项目下的一个个模块:前台(home)、后台(admin).....每个模块都是独立的

七 控制器

1.控制器文件命名方式

控制器名（驼峰字+首字母大写）+Controller+.class.php

1. 控制器类命名方式

控制器名（驼峰字+首字母大写）+Controller

3.命名空间定义 namespace Home\Controller;

统的规范要求，表示当前类是Home模块下的控制器类，命名空间和实际的控制器文件所在的路径是一致的，也就是说： Home\Controller\IndexController类 对应的控制器文件位于应用目录下面的 Home/Controller/IndexController.class.php，如果你改变了当前的模块名(Home)，那么这个控制器类的命名空间也需要随之修改。

注意：命名空间定义必须写在所有的PHP代码之前声明，否则会出错。

4.引入命名空间便于直接使用

|  |
| --- |
| Use Think\Controller  class IndexController extends Controller{  }  等同于  class IndexController extends \Think\Controller{  } |

八 开发规范

1.类文件都是以.class.php为后缀（这里是指的ThinkPHP内部使用的类库文件，不代表外部加载的类库文件），使用驼峰法命名，并且首字母大写，例如 DbMysql.class.php；

类的命名空间地址和所在的路径地址一致，例如 Home\Controller\UserController类所在的路径应该是 Application/Home/Controller/UserController.class.php；

2.确保文件的命名和调用大小写一致，是由于在类Unix系统上面，对大小写是敏感的（而ThinkPHP在调试模式下面，即使在Windows平台也会严格检查大小写）；

3.类名和文件名一致（包括上面说的大小写一致），例如 UserController类的文件命名是UserController.class.php， InfoModel类的文件名是InfoModel.class.php， 并且不同的类库的类命名有一定的规范；

4.函数、配置文件等其他类库文件之外的一般是以.php为后缀（第三方引入的不做要求）；

5.函数的命名使用小写字母和下划线的方式，例如 get\_client\_ip；

6.方法的命名使用驼峰法，并且首字母小写或者使用下划线“\_”，例如 getUserName，\_parseType，通常下划线开头的方法属于私有方法；

7.属性的命名使用驼峰法，并且首字母小写或者使用下划线“\_”，例如 tableName、\_instance，通常下划线开头的属性属于私有属性；

8.以双下划线“\_\_”打头的函数或方法作为魔法方法，例如 \_\_call 和 \_\_autoload；

9.常量以大写字母和下划线命名，例如 HAS\_ONE和 MANY\_TO\_MANY；

10.配置参数以大写字母和下划线命名，例如HTML\_CACHE\_ON；

11.语言变量以大写字母和下划线命名，例如MY\_LANG，以下划线打头的语言变量通常用于系统语言变量，例如 \_CLASS\_NOT\_EXIST\_；

12.对变量的命名没有强制的规范，可以根据团队规范来进行；

13.ThinkPHP的模板文件默认是以.html 为后缀（可以通过配置修改）；

14.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字段是正确写法，类似 \_username 这样的数据表字段可能会被过滤。

九 配置加载优先级

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

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/Conf/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。
2. 应用配置:应用配置文件也就是调用所有模块之前都会首先加载的公共配置文件（默认位于Application/Common/Conf/config.php）。如果更改了公共模块的名称的话，公共配置文件的位置也相应改变
3. 模式配置:如果使用了普通应用模式之外的应用模式的话，还可以为应用模式单独定义配置文件，文件命名规范是： Application/Common/Conf/config\_应用模式名称.php（仅在运行该模式下面才会加载）。
4. 调试配置:如果开启调试模式的话，则会自动加载框架的调试配置文件（位于ThinkPHP/Conf/debug.php）和应用调试配置文件（位于Application/Common/Conf/debug.php）
5. 状态配置：

|  |
| --- |
| 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，我们在入口文件中定义：  define('APP\_STATUS','office');那么就会自动加载该状态对应的配置文件（位于Application/Common/Conf/office.php）。  如果我们回家后，我们修改定义为：  define('APP\_STATUS','home');那么就会自动加载该状态对应的配置文件（位于Application/Common/Conf/home.php）。 |

1. 模块配置:每个模块会自动加载自己的配置文件（位于Application/当前模块名/Conf/config.php）。如果使用了普通模式之外的其他应用模式，你还可以为应用模式单独定义配置文件，命名规范为： Application/当前模块名/Conf/config\_应用模式名称.php（仅在运行该模式下面才会加载）。
2. 动态配置:之前的方式都是通过预先定义配置文件的方式，而在具体的操作方法里面，我们仍然可以对某些参数进行动态配置（或者增加新的配置），主要是指那些还没有被使用的参数。

|  |
| --- |
| 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  // 动态改变缓存有效期C('DATA\_CACHE\_TIME',60);动态配置赋值仅对当前请求有效，不会对以后的请求造成影响。  动态改变配置参数的方法和读取配置的方法在使用上面非常接近，都是使用C方法，只是参数的不同。也可以支持二维数组的读取和设置，使用点语法进行操作，如下：  // 获取已经设置的参数值C('USER\_CONFIG.USER\_TYPE');//设置新的值C('USER\_CONFIG.USER\_TYPE',1); |

8.扩展配置（自定义配置）:'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

十 读取配置项

1.获取已经设置的参数值：C('参数名称')

2.3.2.1版本支持默认值设置

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

十一 模块化

1.概念

| **名称** | **描述** |
| --- | --- |
| 应用 | 基于同一个入口文件访问的项目我们称之为一个应用。 |
| 模块 | 一个应用下面可以包含多个模块，每个模块在应用目录下面都是一个独立的子目录。 |
| 控制器 | 每个模块可以包含多个控制器，一个控制器通常体现为一个控制器类。 |
| 操作 | 每个控制器类可以包含多个操作方法，也可能是绑定的某个操作类，每个操作是URL访问的最小单元 |

2.禁止访问模块

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

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

“MODULE\_DENY\_LIST”=>array(“Common”,”Runtime”,”Api”)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

3.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

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

“MODULE\_ALLOW\_LIST”=>array(“Home”,”Admin”,”User”);

“DEFAULT\_MODULE”=>”Home”;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

4.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

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 关闭模块

\* “MULTI\_MODULE”=> “false”,

\* “DEFAULT\_MODULE”=>”Home”,

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

5.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：

|  |
| --- |
| $\_GET[“m”] = “Home”; // 绑定Home模块到当前入口文件  Define(“APP\_PATH”,”./Application/”);  Require “./ThinkPHP/ThinkPHP.php”;  绑定模块后，原来的访问地址:  http://serverName/index.php/Home/Index/index  就变成  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：

|  |
| --- |
| $\_GET[“m”] = “Home”;  $\_GET[“c”] = “Index”;  Define(“APP\_PATH”,”./Application/”);  Require “./ThinkPHP/ThinkPHP.php”;  绑定模块和控制器后，原来的访问地址：  <http://serverName/index.php/Home/Index/index>  就变成：  http://serverName/home.php/index |

6. 增加模块

在入口文件定义‘BIND\_MODULE’常量，运行入口文件后，注释该语句。例如:

|  |
| --- |
| define(‘BIND\_MODULE’,’Admin’); |

十二 URL模式-’URL\_MODEL’

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.例子：标准模式

<http://serverName/index.php/模块/控制器/操作>

可以通过设置模块绑定或者域名部署等方式简化URL地址中的模块及控制器名称。

2.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，这一点非常关键，因为ThinkPHP的命名规范是采用驼峰法（首字母大写）的规则，而URL中的模块和控制器都是对应的文件，因此在Linux环境下面必然存在区分大小写的问题。

框架内置了一个配置参数用于解决URL大小写的问题，如下：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

“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

当开启调试模式的情况下，这个参数是false，因此你会发现在调试模式下面URL区分 大小写的情况。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

3.详细拆分

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（Home）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://serverName/index.php>

<http://serverName/index.php/Home/Index/index>

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 |

如果你整个应用下面的模块都是采用统一的URL模式，就可以在应用配置文件中设置URL模式，如果不同的模块需要设置不同的URL模式，则可以在模块配置文件中设置。

4.普通模式

普通模式也就是传统的GET传参方式来指定当前访问的模块和操作，例如： 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：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

'VAR\_MODULE' => 'module', // 默认模块获取变量 'VAR\_CONTROLLER' => 'controller', // 默认控制器获取变量 '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

2.PATHINFO模式

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

3.REWRITE模式

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：

|  |
| --- |
| <IfModule mod\_rewrite.c>  RewriteEngine on  RewriteCond %{REQUEST\_FILENAME} !-d  RewriteCond %{REQUEST\_FILENAME} !-f  RewriteRule ^(.\*)$ index.php/$1 [QSA,PT,L]  </IfModule>  接下来，就可以用下面的URL地址访问了： http://localhost/home/user/login/var/value |

4.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

十三 控制器

1.控制器和操作

一般来说ThinkPHP的控制器是一个类，而操作则是类的一个公共方法。

2.定义控制器

控制器需要继承Controller类或者其子类。控制器的名称使用驼峰法命名（首字母为大写），控制器位于:模块名/Controller/控制器名+Controller.class.php。

操作方法的定义必须是公共方法，否则会操作报错

3.多层控制器

ThinkPHP的控制器支持多层和分级

多层:指的是控制器可以分层，例如除了默认的Controller控制器层（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。

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Hmoe/Event/UserEvent.class.php

多级: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：设置2级目录的控制器层

|  |
| --- |
| ‘CONTROLLER\_LEVEL’ => 2;   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. │ ... |

多级控制器的命名空间需要这样定义

|  |
| --- |
| Namespace Home\Controller\Admin  Use ThinkPHP\Controller  Class UserController extends Controller{  public function hello(){  Echo “hello”;  }  Public function test(){  Echo “test”;  }  }  通过  <http://servername/Home/Admin/User/hello访问>  http://servername/Home/Admin/User/test访问 |

4.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。如果需要跨控制器调用的，需要进行单独实例化

|  |
| --- |
| //实例化Hmoe模块的User控制器  $user = new \Home\Controller\UserController();  //实例化Admin模块的Blog控制器  $blog = new \Admin\Controller\BlogCcontroller(); |

系统为上面的控制器实例化提供了一个快捷调用方法A();

|  |
| --- |
| //实例化Hmoe模块的User控制器  $user = A（“User”）;  //实例化Admin模块的Blog控制器  $blog = A（“Admin/Blog”）; //A(“模块名/控制器名”，控制器层名); |

A()方法默认实例化访问控制器层(Controller)，如果实例化其他控制器层如下:

|  |
| --- |
| //假设当前为Home模块  //实例化Event控制器层  $user = A(“User”,”Event”);  $blog = A(“Admin/blog”,”Event”);  上面的代码等效于:  $user = new \Home\Event\UserEvent();  $blog = new \Admin\Event\BlogEvent(); |

十四 前置和后置操作

前置和后置操作指的是在执行某个操作方法之前和之后会自动调用的方法，不过仅对访问控制器有效。

系统会检测当前操作是否具有前置和后置操作，如果存在就会按照顺序执行，前置和后置操作的定义方式如下:

|  |
| --- |
| Namespace Home\Controller  Use Think\Controller  Class IndexController extends Controller{  Public function \_before\_index(){  Echo “before”;  }  Public function index(){  Echo “index”;  }  Public function \_after\_index(){  Echo “after”;  }  Public function \_before\_test(){  Echo “before”;  }  Public function test(){  Echo “index”;  }  Public function \_after\_test(){  Echo “after”;  }  }  输入如下进行测试:  <http://servername/Home/Index/index>  [http://servername/Home/Index/](http://servername/Home/Index/index)test |
| 注意:  1. 如果当前的操作并没有定义操作方法，而是直接渲染模板文件，那么如果定义了前置和后置方法的话，依然会生效。真正有模板输出的可能仅仅是当前的操作，前置和后置操作一般情况是没有任何输出的。   1. 需要注意的是，在有些方法里面使用了exit或者错误输出之类的话 有可能不会再执行后置方法了。例如，如果在当前操作里面调用了系统Action的error方法，那么将不会再执行后置操作，但是不影响success方法的后置方法执行。 2. 注意构造方法与析构方法的差别。 |

十五 Action参数绑定

参数绑定是通过直接绑定到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。

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

要启用参数绑定功能，首先得确定开启了“URL\_PARAMS\_BIND”的设置

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* “URL\_PARAMS\_BIND” => true;

\* 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

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1.按变量名绑定

默认的变量绑定方式是根据变量名进行绑定，例如，我们给Blog控制器定义了两个操作方法read和archive。由于read操作需要指定一个id参数，archive方法需要指定年份（year）和月份（month）两个参数，如下定义:

|  |
| --- |
| Namespace Home\Controller  Use Think\Controller  Class BlogController extends Controller{  Public function read($id){  Echo “id:”.$id;  }  Public function archive($year=”2013”,$month=”01”){  Echo “Year:”.$year.”<br>Month:”.$month;  }  }  URL访问地址:  <http://servername/Home/Blog/read/id/10>  <http://servername/Home/Blog/archive/year/2020/month/12>  变量名绑定不一定由访问URL决定，路由地址也能起到相同的作用  <http://serverName/index.php?s=/Home/Blog/read/id/5>  <http://serverName/index.php?s=/Home/Blog/archive/year/2013/month/11>  <http://serverName/index.php?c=Blog&a=read&id=5>  <http://serverName/index.php?c=Blog&a=archive&year=2013&month=11> |

2.按变量顺序绑定

第二种方式是按照变量的顺序绑定，这种情况下URL地址中的参数顺序非常重要，不能随意调整。要按照变量顺序进行绑定，必须先设置：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* “URL\_PARAMS\_BIND\_TYPE” => 1;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

|  |
| --- |
| Namespace Home\Controller  Use Think\Controller  Class BlogController extends Controller{  Public function read($id){  Echo “id:”.$id;  }  Public function archive($year=”2013”,$month=”01”){  Echo “Year:”.$year.”<br>Month:”.$month;  }  }  URL访问地址:  [http://servername/Home/Blog/read/10](http://servername/Home/Blog/read/id/10)  [http://servername/Home/Blog/archive/2020/12](http://servername/Home/Blog/archive/year/2020/month/12) |

十六 伪静态

URL伪静态通常是为了满足更好的SEO效果，可以通过设置“URL\_HTML\_SUFFIX”参数，随意在URL的最后增加你想要的静态后缀，而不会影响正常执行

默认情况下,伪静态的设置为html，如果我们设置伪静态后缀为空，则可以支持所有的静态后缀，并且会记录伪静态后缀到常量\_\_EXT\_\_,

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* “URL\_HTML\_SUFFIX” => “”;

\* “URL\_HTML\_SUFFIX” => “html|shtml|xml...”

\* 多个后缀用 | 隔开

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

可以设置“URL\_DENY\_SUFFIX”禁止访问后缀，且优先级高于“URL\_HTML\_SUFFIX”

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* “URL\_DENY\_SUFFIX”=> “pdf|ico|png....”

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

十七 URL生成——U方法

1.定义规则

U(“地址表达式”,”参数”,”伪静态后缀”,”显示域名”);

2.地址表达式

[“模块名/控制器/操作#锚点@域名?参数1=值&参数2=值”]

如果不写入模块的话，就表示当前模块

|  |
| --- |
| U(“User/add”) //生成User控制器的add操作的URL地址  U(“Blog/read?Id=1”)// 生成Blog控制器的read操作 并且id为1的URL地址  U(“Admin/User/select”)// 生成Admin模块的User控制器的select操作的URL地址 |

3.参数

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以第一个参数定义

|  |
| --- |
| U(“Blog/cate”,”array(‘cate\_id’=>1,’state’=>1)”);  U(“Blog/cate”,”cate\_id=1&state=1”);  U(“Blog/cate?cate\_id=1&state=1”); |

4.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：

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

5.锚点支持

U函数可以直接生成URL地址中的锚点，例如：

|  |
| --- |
| U('Blog/read#comment?id=1');  生成的URL地址可能是：  http://serverName/index.php/Home/Blog/read/id/1#comment |

十八 Ajax返回

ThinkPHP有效支持ajax请求，在\Thinl\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/EVAL四种方式给客户端接收数据，并且支持配置其他数据格式返回。

1.调用实例-1

|  |
| --- |
| $data=“OK”;  $this->ajaxReturn($data); |

2.调用实例-2

|  |
| --- |
| $data=array(“success”=>true,”msg”=>’请求成功’);  $this->ajaxReturn($data); |

1. 调用实例-3

默认配置采用JSON格式返回数据，可以用过”DEFAULT\_AJAX\_RETURN”进行配置，也可以指定数据格式:

|  |
| --- |
| $data=array(“success”=>true,”msg”=>’请求成功’);  $this->ajaxReturn($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。

十九 跳转和重定向

1.跳转

在应用开发中，经常遇到一些带有提示信息的跳转页面，例如操作成功或者操作错误页面，并且自动跳转到另一个页面。系统的\Think\Controller类中内置了两个跳转方法success和error，用于页面跳转提交，且支持ajax提交。

调用方式:

|  |
| --- |
| $User = M(“User”);  $result = $User->add($data);  If($result){  //设置成功后跳转页面的地址，默认的返回页面是$\_SERVER['HTTP\_REFERER']  $this->success(“新增成功”,”/User/list”);  }else{  //错误默认跳转会前一页，通常不进行设置  $this->success(“新增失败”);  } |

Success和error方法的第一个参数是提示信息，第二个参数表示跳转地址，第三个参数是跳转时间(时间为秒)，

|  |
| --- |
| // 操作完成3秒后跳转到 /Article/index  $this->success('操作完成','/Article/index',3);  // 操作失败5秒后跳转到 /Article/error  $this->error('操作失败','/Article/error',5); |

跳转地址是可选的，success方法的默认跳转地址是$\_SERVER["HTTP\_REFERER"]，error方法的默认跳转地址是javascript:history.back(-1);。默认的等待时间success方法是1秒，error方法是3秒

Success和error方法都可以对应的模板，默认的设置是两个方法对应的模板都是:

|  |
| --- |
| //默认错误跳转对应的模板文件  “TMPL\_ACTION\_ERROR”=>THINK\_PATH.”Tpl/dispatch\_jump.tpl”  //默认成功跳转对应的模板文件  “TMPL\_ACTION\_SUCCESS”=>THINK\_PATH.”Tpl/dispatch\_jump.tpl” |

也可以使用项目内部的模板文件

|  |
| --- |
| //默认错误跳转对应的模板文件  “TMPL\_ACTION\_ERROR”=>”Public:error”;  //默认成功跳转对应的模板文件  “TMPL\_ACTION\_SUCCESS”=>”Public:success”; |

| **变量** | **含义** |
| --- | --- |
| $msgTitle | 操作标题 |
| $message | 页面提示信息 |
| $status | 操作状态 1表示成功 0 表示失败 具体还可以由项目本身定义规则 |
| $waitSecond | 跳转等待时间 单位为秒 |
| $jumpUrl | 跳转页面地址 |

Success和error方法会自动判断当前请求是否属于Ajax请求，如果属于Ajax请求则会调用ajaxReturn方法返回信息。Ajax方式下面，success和error方法会封装下面的数据返回:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* $data[‘info’] = $message //提示信息内容

\* $data[‘status’] = $status //状态，1->sucess、0->error

\* sdata[‘url’] = $jumpUrl

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

2.重定向

Controller的redirect方法可以实现页面的重定向功能。

Redirect方法的参数用法与U方法一致，例如：

|  |
| --- |
| //重定向到New模块的Category操作  $this->redirect('New/category', array('cate\_id' => 2), 5, '页面跳转中...');  上面的用法是停留5秒后跳转到New模块的category操作，并且显示页面跳转中字样，重定向后会改变当前的URL地址。 |

如果你仅仅是想重定向要一个指定的URL地址，而不是到某个模块的操作方法，可以直接使用redirect函数重定向，例如：

|  |
| --- |
| //重定向到指定的URL地址  redirect('/New/category/cate\_id/2', 5, '页面跳转中...') |

Redirect函数的第一个参数是一个URL地址。

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。

二十 输入变量

1.获取变量

|  |
| --- |
| 传统获取变量方法  $id = $\_GET['id']; // 获取get变量  $name = $\_POST['name']; // 获取post变量  $value = $\_SESSION['var']; // 获取session变量  $name = $\_COOKIE['name']; // 获取cookie变量  $file = $\_SERVER['PHP\_SELF']; // 获取server变量 |

传统的获取变量的方法没有统一的安全机制，ThinkPHP框架提供I函数进行变量获取和过滤。

I方法是众多单字母函数中的新成员，其命名来自于Input(输入)，主要用于更加方便和安全的获取系统输入变量，可以用于任何订单，调用格式如下:

|  |
| --- |
| I（‘变量类型.变量名’，[‘默认值’],[‘过滤方法’]） |

2.变量类型

变量类型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/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 |

注意:1.变量类型不区分大小写

2.变量名严格区分大小写

3.默认值和过滤规则为可选参数

3.调用实例

|  |
| --- |
| echo I(“get.id”); //相当于 $\_GET['id']  echo I(“get.name”,””);// 如果不存在$\_GET['name'] 则返回空字符串  // 采用htmlspecialchars方法对$\_GET['name'] 进行过滤，如果不存在则返回空字符串  Echo I(‘get.name’,’’,’htmlspecialchars’);  //获取整个变量数组  I(‘get.’);  //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：  I('id'); // 等同于 I('param.id')  I('name'); // 等同于 I('param.name') |

4.变量过滤

在调用I函数时没有指定过滤规则，系统会采用默认的过滤机制(DEFAULT\_FILTER): ‘DEFAULT\_FILTER’=>‘htmlspecialchars’;

该参数支持多个过滤：

‘DEFAULT\_FILTER’=>‘strip\_tags,htmlspecialchars,...’;

设置后等同于：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* // 等同于 htmlspecialchars(strip\_tags($\_GET['name']))

\* I('get.name');

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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：

|  |
| --- |
| 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：  I('post.email','','email'); |

|  |  |
| --- | --- |
| 过滤规则名称 | 含义 |
| int | 在指定的范围以整数验证值。 |
| boolean | 如果是 "1", "true", "on" 以及 "yes"，则返回 true，如果是 "0", "false", "off", "no" 以及 ""，则返回 false。否则返回 NULL。 |
| Float | 以浮点数验证值。 |
| Validate\_regexp | 根据 regexp，兼容 Perl 的正则表达式来验证值。 |
| Validate\_url | 把值作为 URL 来验证。 |
| Validate\_email | 把值作为 e-mail 来验证。 |
| Validate\_ip | 把值作为 IP 地址来验证。 |
| String | 去除标签，去除或编码特殊字符。 |
| Stripped | "string" 过滤器的别名。 |
| Encoded | URL-encode 字符串，去除或编码特殊字符。 |
| Special\_chars | HTML 转义字符 '"<>& 以及 ASCII 值小于 32 的字符。 |
| Unsafe\_raw | 不进行任何过滤，去除或编码特殊字符。 |
| Email | 删除所有字符，除了字母、数字以及 !#$%&'\*+-/=?^\_`{|}~@.[] |
| Url | 删除所有字符，除了字母、数字以及 $-\_.+!\*'(),{}|\\^~[]`<>#%";/?:@&= |
| Number\_int |  |
| Number\_float |  |
| Magic\_quotes | 应用 addslashes()。 |
| callback | 回调函数 |

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：

// 下面两种方式都不采用任何过滤方法

I('get.name','','');

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。

二一 请求类型

系统内置一些常量用于判断请求类型，包括:

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

调用实例

|  |
| --- |
| Namespace Home\Controller  Use Think\Controller  Class UserController extends Controller{  Public function update(){  If(IS\_GET){  $User = M(‘user’);  $User->create();  $User->save();  $this->success(‘更新成功 ’);  }else{  $this->error('非法请求');  }  }  } |

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。

二二 空操作

空操作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，可以实现错误页面和一些URL的优化。

例如:用空操作功能来实现一个城市切换的功能。给CityController类定义一个\_empty方法

|  |
| --- |
| <?php  Namespace Home\Controller  Use Think\Controller  Class CityController extends Controller{  Public function \_empty($name){  //把所有城市的操作解析到city方法  $this->city($name);  }  //注意 city方法 本身是 protected 方法  protected function city($name(){  //和$name这个城市相关的处理  echo '当前城市' . $name;  }  }  ?>  接下来，我们就可以在浏览器里面输入  <http://serverName/index.php/Home/City/beijing/>  <http://serverName/index.php/Home/City/shanghai>/  <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：  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。 |

二三 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），利用这个机制可以用来定制错误页面和进行URL的优化

对需求进一步优化：

|  |
| --- |
| <http://serverName/index.php/Home/City/shanghai/>  变成  <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。

可以给项目定义一个EmptyController类：

|  |
| --- |
| <?php  namespace Home\Controller  use Think\Controller  class EmptyController extends Controller{  public function index(){  //根据当前控制器名来判断要执行那个城市的操作  $city = CONTROLLER\_name;  $this->city($city);  }  protected function city($name){  echo “当前城市:”.$name;  }  }  ?>  接下来可以在浏览器内输入  <http://serverName/index.php/Home/beijing/>  <http://serverName/index.php/Home/shanghai/>  <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：  当前城市:beijing  当前城市:shanghai  当前城市:shenzhen |

二四 模型(MODEL)

在ThinkPHP中基础的模型类为Think\Model，该类基本完成了基本的CRUD，ActiveRecord模式、连贯操作、统计查询，一些高级特性被封装到另外的模型扩展中。

1.模型定义

模型类并非必须定义，只有当存在独立的业务逻辑或者属性的时候才需要定义。模型类通常要继承系统的Model类或者其子类，下面是一个UserModel类的定义

|  |
| --- |
| namespace Home\Model  use Think\Model  class UserModel extends Model{  } |

模型类大多用于操作数据表，按照系统的规范来命名模型类，大多数可以自动对应数据表。

模型类的命名规则除去表前缀的数据表名称，采用驼峰法命名，首字母大写，然后加上模型层的定义，例如：

|  |  |
| --- | --- |
| 模型名 | 约定对应数据表（表前缀为:think） |
| UserModel | think\_user |
| UserTypeModel | think\_user\_type |

如果你的规则和上面的系统约定不符合，那么需要设置Model类的数据表名称属性，以确保能够找到对应的数据表。

在ThinkPHP的模型里面，有几个关于数据表名称的属性定义：

| **属性** | **说明** |
| --- | --- |
| tablePrefix | 定义模型对应数据表的前缀，如果未定义则获取配置文件中的DB\_PREFIX参数 |
| tableName | 不包含表前缀的数据表名称，一般情况下默认和模型名称相同，只有当你的表名和当前的模型类的名称不同的时候才需要定义。 |
| trueTableName | 包含前缀的数据表名称，也就是数据库中的实际表名，该名称无需设置，只有当上面的规则都不适用的情况或者特殊情况下才需要设置。 |
| dbName | 定义模型当前对应的数据库名称，只有当你当前的模型类对应的数据库名称和配置文件不同的时候才需要定义。 |

例如：在数据库里有一个think\_categories的数据表，而我们定义的模型类名称为CategoryModel，按照系统的规则约定，这个模型的名称为Category，对应的数据表为think\_category，但现在数据库里数据表的名称为think\_categories，因此就需要设置tableName属性来改变默认的规则（假设我们在DB\_PREFIX）

|  |
| --- |
| namespace Home\Model  use Think\Model  class CategoryModel extends Model{  protected $tableName = ‘categories’  }  注意：这个属性不需要添加表前缀 |

如果我们需要CategoryModel模型操作数据表top\_category，设置表前缀属性

|  |
| --- |
| namespace Home\Model  use Think\Model  class CategoryModel extends Model{  protected $tablePrefix = ‘top\_’  } |

如果你的数据表直接就是category，而没有前缀，则可以设置tablePrefix为空字符串。

|  |
| --- |
| namespace Home\Model  use Think\Model  class CategoryModel extends Model{  protected $tablePrefix = ‘’  }  没有表前缀的情况必须设置，否则会获取当前配置文件中的 DB\_PREFIX。 |

另外一种特殊情况，需要操作的数据表为top\_categories，需要定义trueTableName属性

|  |
| --- |
| namespace Home\Model  use Think\Model  class CategoryModel extends Model{  protected $trueTableName = ‘top\_categories’  }  注意trueTableName需要完整的表名定义。 |

除了数据表的定义外，还可以对数据库进行定义（用于操作当前数据库以外的数据表），例如 top.top\_categories：

|  |
| --- |
| namespace Home\Model  use Think\Model  class CategoryModel extends Model{  protected $trueTableName = ‘top\_categories’;  protected $dbName = “top”;  }  注意trueTableName需要完整的表名定义。 |

二五 模型实例化(MODEL)

根据不同的模型定义，有几种实例化方法，根据需要进行选择

1.直接实例化

与实例化其他类库一样直接实例化模型类

|  |
| --- |
| $User = new \Home\Model\UserModel();  $Info = new \Home\Model\InfoModel();  //带参数实例化  $New = new \Home\Model\NewModel(‘blog’,’think\_’,$connection); |

模型类通常都是继承系统的\Think\Model类，该类的架构方法有三个参数:

|  |
| --- |
| Model([‘模型名’],[‘数据表前缀’],[‘数据库连接信息’]); |

三个参数都是可选的，大多数情况下，我们根本无需传入任何参数即可实例化。

| **参数** | **描述** |
| --- | --- |
| 模型名 | 模型的名称 和数据表前缀一起配合用于自动识别数据表名称 |
| 数据表前缀 | 当前数据表前缀 和模型名一起配合用于自动识别数据表名称 |
| 数据库连接信息 | 当前数据表的数据库连接信息 如果没有则获取配置文件中的 |

2.数据库连接信息三种格式

|  |
| --- |
| 1.字符串定义  type://username:passwd@hostname:port/DbName  $New = new \Home\Model\NewModel(‘blog’,’think\_’,’mysql://root:123456@loaclhost/imooc’)  2.数组定义  $dbconnection = array(  ‘db\_type’ => ’mysql’,  ‘db\_host’ => ‘localhost’,  ‘db\_user’ => ‘root’,  ‘db\_pwd’ => ‘’,  ‘db\_name’ = >’imooc’,  ‘db\_post’ => ‘’  );  $New = new \Home\Model\NewModel(‘blog’,’think\_’,’$connection’);   1. 配置定义   事先在配置文件定义好数据库连接信息，然后在实例化的时候直接传入配置的名称即可 |

3.D()方法实例化

直接实例化需传入完整的类名，系统提供了一个快捷方法D()用户数据模型的实例化操作,要实例化自定义模型类，可以用使用下面的方式:

|  |
| --- |
| <?php  //实例化模型  $User = D(‘User’);  //相当于执行 $User = new \Home\Model\UserModer();  //执行具体的数据库操作  $User->select();  当 \Home\Model\UserModel 类不存在的时候，D函数会尝试实例化公共模块下面的 \Common\Model\UserModel 类。  D方法可以自动检测模型类，如果存在自定义的模型类，则实例化自定义模型类，如果不存在，则会实例化系统的\Think\Model基类，同时对于已实例化过的模型，不会重复去实例化。  D方法还可以支持跨模块调用，需要使用：  //实例化Admin模块的User模型  D('Admin/User');  //实例化Extend扩展命名空间下的Info模型  D('Extend://Editor/Info');  注意：跨模块实例化模型类的时候 不支持自动加载公共模块的模型类。 |

3.M()方法实例化

D方法实例化模型类通常是实例化某个具体的模型类，如果仅仅是对数据表进行基本的CRUD操作，使用M方法实例化的话，由于不加载具体的模型类，性能会更高。

例如：

|  |
| --- |
| //使用M方法实例化  $User = M(‘User’);  //和用法 $User = new \Think\Model('User'); 等效  // 使用M方法实例化 操作db\_name数据库的ot\_user表  $User = M('db\_name.User','ot\_');  // 执行其他的数据操作  $User->select();  M方法的参数和\Think\Model类的参数是一样的，也就是说，我们也可以这样实例化：  $New = M('new','think\_',$connection);  // 等效于  $New = new \Think\Model('new','think\_',$connection);  M方法实例化的时候，默认情况下是直接实例化系统的\Think\Model类，如果我们希望实例化其他的公共模型类的话，可以使用如下方法：  $User = M('\Home\Model\CommonModel:User','think\_','db\_config');  // 相当于  $User = new \Home\Model\CommonModel('User','think\_','db\_config'); |

4.实例化空模型类

如果你仅仅是使用原生SQL查询的话，不需要使用额外的模型类，实例化一个空模型类即可进行操作了，例如：

|  |
| --- |
| //实例化空模型  $Model = new Model();  //或者使用M快捷方法是等效的  $Model = M();  //进行原生的SQL查询  $Model->query('SELECT \* FROM think\_user WHERE status = 1'); |

实例化的过程中，经常使用D方法和M方法，这两个方法的区别在于M方法实例化模型无需用户为每个数据表定义模型类，如果D方法没有找到定义的模型类，则会自动调用M方法

二六 字段定义

通常每个模型类是操作某个数据表，在大多数情况下，系统会自动获取当前数据表的字段信息。

系统会在模型首次实例化的时候自动获取数据表的字段信息（而且只需要一次，以后会永久缓存字段信息，除非设置不缓存或者删除），如果是调试模式则不会生成字段缓存文件，则表示每次都会重新获取数据表字段信息。

字段缓存保存在Runtime/Data/\_fields/ 目录下面，缓存机制是每个模型对应一个字段缓存文件（注意：并非每个数据表对应一个字段缓存文件），命名格式是：

##### 数据库名.模型名（小写）.php

例如：

demo.user.php // User模型生成的字段缓存文件

demo.article.php // Article模型生成的字段缓存文件

字段缓存包括数据表的字段信息、主键字段和是否自动增长，如果开启字段类型验证的话还包括字段类型信息等等，无论是用M方法还是D方法，或者用原生的实例化模型类一般情况下只要是不开启调试模式都会生成字段缓存（字段缓存可以单独设置关闭）。

二七 连接数据库

1.全局配置定义

|  |
| --- |
| 1. //数据库配置信息 2. 'DB\_TYPE' => 'mysql', // 数据库类型 3. 'DB\_HOST' => 'localhost', // 服务器地址 4. 'DB\_NAME' => 'thinkphp', // 数据库名 5. 'DB\_USER' => 'root', // 用户名 6. 'DB\_PWD' => '123456', // 密码 7. 'DB\_PORT' => 3306, // 端口 8. 'DB\_PREFIX' => 'think\_', // 数据库表前缀 |

2.模型类内定义

如果在某个模型类里面定义了connection属性的话，则实例化该自定义模型的时候会采用定义的数据库连接信息，而不是配置文件中设置的默认连接信息，通常用于某些数据表位于当前数据库连接之外的其它数据库，例如：

|  |
| --- |
| //在模型里单独设置的数据库连接信息  namespace Home\Model  use Think\Model  class UserModel extends Model{  protected $connection = array(  ‘db\_type’ => ‘mysql’,  ‘db\_host’ =>’localhost’,  ‘db\_user’ =>’root’,  ‘db\_pwd’ =>’123456’,  ‘db\_pord’=>’3306’,  ‘db\_name’=>’shopimooc’,  ‘db\_prefix’=>’imooc\_’  );  } |

二八 连贯操作

ThinkPHP模型基础类提供的连贯操作方法（也有些框架称之为链式操作），可以有效的提高数据存取的代码清晰度和开发效率，并且支持所有的CURD操作。

1.【where】

1.1 字符串条件直接查询和操作

|  |
| --- |
| //实例化操作  $User = M(‘User’);  $User->where(‘id = 1 and statu = 2’)->select;  最后生成的语句  SELECT \* FROM think\_user WHERE type=1 AND status=1  如果使用3.1版本以上，可以使用预处理机制  $Model->where("id=%d and username='%s' and xx='%f'",$id,$username,$xx)->select();  如果$id变量来自用户提交或者URL地址的话，如果传入的是非数字类型，则会强制格式化为数字格式后进行查询操作。  字符串预处理格式类型支持指定数字、字符串等，具体可以参考vsprintf方法的参数说明。 |

1.2数组条件-thinkphp推荐的方法

1.2.1 普通查询

|  |
| --- |
| $User = M(‘User’);  $map[‘name’] = ‘Lin’;  $map[‘age’]=’30’;  $User->where($map)->select();  //最后生成的SQL语句  SELECT \* FROM think\_user WHERE `name`=Lin’ AND age = 30; |

1.2.1 or逻辑查询

|  |
| --- |
| $User = M(‘User’);  $map[‘name’] = ‘Lin’;  $map[‘age’]=’30’;  $map[‘\_logic’]=’or’; //修改逻辑连接符  $User->where($map)->select();  //最后生成的SQL语句  SELECT \* FROM think\_user WHERE `name`=Lin’ or age=30 |

1.2.3 表达式查询

上面的查询条件仅仅是一个简单的相等判断，可以使用查询表达式支持更多的SQL查询语法，查询表达式的使用格式：

|  |
| --- |
| $map[‘id’] = array(‘elt’,’5’);  $map[‘字段名’]=array(‘表达式’,’条件值’);  $map->where($map)->select;  //最后生成的SQL语句  SELECT \* FROM think\_user WHERE id <= 5; |

|  |  |
| --- | --- |
| 表达式 | 含义 |
| eq | 等于 |
| lt | 小于 |
| elt | 小于等于 |
| gt | 大于 |
| egt | 大于等于 |
| link | 模糊查询 |
| [not]between | 区间查询 |
| [not]in | in查询 |
| exp | 表达式查询，支持SQL语法 |

1.2.3多次调用

3.1.3版本开始，where方法支持多次调用，但字符串条件只能出现一次，例如：

|  |
| --- |
| $map['a'] = array('gt',1);  $where['b']=1;  $Model->where($map)->where($where)->where('status=1')->select();多次的数组条件表达式会最终合并，但字符串条件则只支持一次。 |

2.【table】

table方法也属于模型类的连贯操作方法之一，主要用于指定操作的数据表。

一般情况下，操作模型的时候系统能够自动识别当前对应的数据表，所以，使用table方法的情况通常是为了：

●切换操作的数据表；

●对多表进行操作；

|  |
| --- |
| $Model->table(‘think\_user’)->where(‘status>1’)->select  也可以在table方法里指定数据库  $Model->talbe(‘db\_name.think\_user’)->where(‘status>1’)->select(); |

3.【alias】

alias用于设置当前数据表的别名，便于使用其他的连贯操作例如join方法等。

|  |
| --- |
| $Model = M('User');  $Model->alias('a')->join('\_\_DEPT\_\_ b ON b.user\_id= a.id')->select();  最终生成的SQL语句类似于：  SELECT \* FROM think\_user a INNER JOIN think\_dept b ON b.user\_id= a.id |

4.【field】

field方法页属于连贯操作方法之一，主要目的是标识要返回或者操作的字段。

4.1用于查询

在查询中field的使用的频繁的

|  |
| --- |
| $User = M(“User”);  $User->field(‘id,username’)->select();  SELECT id,username FROM think\_user;  可以给字段设置别名  $User->field(‘id,username AS name’)->select();  SELECT id,username AS name FROM think\_user; |

4.2 可以直接使用SQL语句

|  |
| --- |
| $Model->field('id,SUM(score)')->select();  执行的SQL相当于：  SELECT id,SUM(score) FROM table |

4.3数组查询

|  |
| --- |
| $Model->field(array('id','nickname'=>'name'))->select();  执行的SQL相当于：  SELECT id,nickname as name FROM table |

4.4排除字段

|  |
| --- |
| $User = M(“User”);  $User->field(‘id,username’,true)->select();  获取除了id,username外的所有字段 |

5.【order】

order方法属于模型的连贯操作方法之一，用于对操作的结果排序

|  |
| --- |
| $User->where(‘status = 1’)->order(‘id desc’)->select();  $Model->where('status=1')->order('id desc,status')->limit(5)->select();  如果没有指定desc或者asc排序规则的话，默认为asc。 |

6.【limit】

6.1限制结果数量

|  |
| --- |
| 例如获取满足要求的10个用户，如下调用即可：  $User = M(“User”);  $User->where(‘id>10’)->limit(10)->select();  limit方法也可以用于写操作，例如更新满足要求的3条数据：  $User->where('score=100')->limit(3)->save(array('level'=>'A')); |

6.2分页查询

|  |
| --- |
| 用于文章分页查询是limit方法比较常用的场合，例如：  $Article = M('Article');  $Article->limit('10,25')->select();  表示查询文章数据，从第10行开始的25条数据 |

7.【page】

page方法也是模型的连贯操作方法之一，是完全为分页查询而诞生的一个人性化操作方法。

我们在前面已经了解了关于limit方法用于分页查询的情况，而page方法则是更人性化的进行分页查询的方法，例如还是以文章列表分页为例来说，如果使用limit方法，我们要查询第一页和第二页（假设我们每页输出10条数据）写法如下：

|  |
| --- |
| $Article = M('Article');  $Article->limit('0,10')->select(); // 查询第一页数据  $Article->limit('10,10')->select(); // 查询第二页数据 |

虽然利用扩展类库中的分页类Page可以自动计算出每个分页的limit参数，但是如果要自己写就比较费力了，如果用page方法来写则简单多了，例如：

|  |
| --- |
| $Article = M('Article');  $Article->page('1,10')->select(); // 查询第一页数据  $Article->page('2,10')->select(); // 查询第二页数据 |

显而易见的是，使用page方法你不需要计算每个分页数据的起始位置，page方法内部会自动计算

page方法还可以和limit方法配合使用，例如：

|  |
| --- |
| $Article->limit(25)->page(3)->select(); |

当page方法只有一个值传入的时候，表示第几页，而limit方法则用于设置每页显示的数量，也就是说上面的写法等同于：

|  |
| --- |
| $Article->page('3,25')->select(); |

8.【group】

GROUP方法也是连贯操作方法之一，通常用于结合合计函数，根据一个或多个列对结果集进行分组

group方法只有一个参数，并且只能使用字符串

例如，我们都查询结果按照用户id进行分组统计：

|  |
| --- |
| $this->field(‘id,max(score)’)->group(‘user\_id’)->select();  生成的SQL语句是：  SELECT username,max(score) FROM think\_score GROUP BY user\_id  也支持对多个字段进行分组，例如：  $this->field('username,max(score)')->group('user\_id,test\_time')->select();  生成的SQL语句是：  SELECT username,max(score) FROM think\_score GROUP BY user\_id,test\_time |

9.【having】

HAVING方法也是连贯操作之一，用于配合group方法完成从分组的结果中筛选（通常是聚合条件）数据。

having方法只有一个参数，并且只能使用字符串，例如：

|  |
| --- |
| $this->field('username,max(score)')->group('user\_id')->having('count(test\_time)>3')->select();  生成的SQL语句是：SELECT username,max(score) FROM think\_score GROUP BY user\_id HAVING count(test\_time)>3 |

10. 【join】

join方法也是连贯操作之一，用于根据两个或多个表之间的关系，从这些表中查询数据。

join通常有下面几种类型：

|  |  |
| --- | --- |
| INNER JOIN | 如果表中有至少一个匹配，则返回行，等同于 JOIN |
| LEFT JOIN | 即使右表中没有匹配，也从左表返回所有的行 |
| RIGHT JOIN | 即使左表中没有匹配，也从右表返回所有的行 |
| FULL JOIN | 只要其中一个表中存在匹配，就返回行 |

10.1 INNER JOIN——内关联

|  |
| --- |
| $admin->field('id,username')->join("INNER JOIN imooc\_user ON imooc\_admin.id = imooc\_user.id")->select();  执行SQL语句：  select id,username from imooc\_admin join imooc\_user ON imooc\_admin.id = imooc\_user.id |

10.2 LEFT JOIN（默认）——左外关联

|  |
| --- |
| $admin->field('id,username')->join("imooc\_user ON imooc\_admin.id = imooc\_user.id")->select(); |

10.3 RIGHT JOIN——右外关联

|  |
| --- |
| $admin->field('id,username')->join("RIGHT JOIN imooc\_user ON imooc\_admin.id = imooc\_user.id")->select(); |

11. 【union】

UNION操作用于合并两个或多个 SELECT 语句的结果集.

使用实例

|  |
| --- |
| 1. $Model->field('name') 2. ->table('think\_user\_0') 3. ->union('SELECT name FROM think\_user\_1') 4. ->union('SELECT name FROM think\_user\_2') 5. ->select(); |

数组用法：

|  |
| --- |
| 1. $Model->field('name') 2. ->table('think\_user\_0') 3. ->union('SELECT name FROM think\_user\_1',true) 4. ->union('SELECT name FROM think\_user\_2',true) 5. ->select(); 6. 或者 7. $Model->field('name') 8. ->table('think\_user\_0') 9. ->union(array('SELECT name FROM think\_user\_1','SELECT name FROM think\_user\_2')) 10. ->select(); |

支持union all操作（不过滤重复数据）

|  |
| --- |
| 1. $Model->field('name') 2. ->table('think\_user\_0') 3. ->union('SELECT name FROM think\_user\_1',true) 4. ->union('SELECT name FROM think\_user\_2',true) 5. ->select(); |

12. 【distinct】

distinct消除重复行;group by是分组语句

|  |
| --- |
| $Model->distinct(true)->field('name')->select();  生成的SQL语句是：  SELECT DISTINCT name FROM think\_user |

二九 CURD操作

1.数据创建

在进行数据操作之前，我们往往要对数据进行创建，例如对于提交的表单：

|  |
| --- |
| // 获取表单的POST数据  $data['name'] = $\_POST['name'];  $data['email'] = $\_POST['email'];  // 更多的表单数据值获取  //…… |

ThinkPHP可以帮助你快速地创建数据对象，最典型的应用就是自动根据表单数据创建数据对象，这个优势在一个数据表的字段非常之多的情况下尤其明显。

调用实例：

|  |
| --- |
| //实例化User模型  $User = M(‘User’);  //根据表单提交的POST数据创建数据对象  $User->create(); |

2.数据写入

ThinkPHP的数据写入操作使用add方法，使用示例如下：

|  |
| --- |
| $User = M(‘User’);  $data[‘name’] = ‘ThinkPHP’;  $data[‘email’] = [‘thinkphp@think.com’;](mailto:‘thinkphp@think.com’;)  $User->add($data);  或者使用连贯方法  $Userp>data($data)->add(); |

如果在add之前已经创建数据对象的话（例如使用了create或者data方法），add方法就不需要再传入数据了。使用create方法的例子：

|  |
| --- |
| $User = M(‘User’);  // 根据表单提交的POST数据创建数据对象  if($User->create()){  $result = $User->add(); // 写入数据到数据库  if($result){  // 如果主键是自动增长型 成功后返回值就是最新插入的值  $inserID = $result;  }  }  create方法并不算是连贯操作，因为其返回值可能是布尔值，所以必须要进行严格判断。 |

3.数据读取

在ThinkPHP中读取数据的方式很多，包括读取数据、读取数据集、读取字段值

3.1读取数据

读取数据指读取数据表中的一行记录，主要通过find完成，例如：

|  |
| --- |
| $User = M(‘user’); //实例化User对象  // 查找status值为1name值为think的用户数据  $data = $User->where(‘status=1 AND name=”think”’)->find;  dump($data);  find方法查询数据的时候可以配合相关的连贯操作方法，其中最关键的则是where方法，如何使用where方法我们会在查询语言章节中详细描述。  如果查询出错，find方法返回false，如果查询结果为空返回NULL，查询成功则返回一个索引数组（键值是字段名或者别名）。如果上面的查询成功的话，会输出：  array (size=3)  'name' => string 'thinkphp' (length=8)  'email' => string 'thinkphp@gmail.com' (length=18)  'status'=> int 1  即使满足条件的数据不止一个，find方法也只会返回第一条记录（可以通过order方法排序后查询）。  还可以用data方法获取查询后的数据对象（查询成功后）  $User = M("User"); // 实例化User对象  // 查找status值为1name值为think的用户数据  $User->where('status=1 AND name="thinkphp"')->find();  dump($User->data()); |

3.2读取数据集

读取数据集其实就是获取数据表中的多行记录（以及关联数据），使用select方法，使用示例：

|  |
| --- |
| $User = M("User"); // 实例化User对象  // 查找status值为1的用户数据 以创建时间排序 返回10条数据  $list=$User->where('status=1')->order('create\_time')->limit(10)->select();  如果查询出错，select的返回值是false，如果查询结果为空，则返回NULL，否则返回二维数组。 |

3.3读取字段值

读取字段值其实就是获取数据表中的某个列的多个或者单个数据，最常用的方法是 getField方法

|  |
| --- |
| $User = M("User"); // 实例化User对象  // 获取ID为3的用户的昵称  $nickname = $User->where('id=3')->getField('nickname');  默认情况下，当只有一个字段的时候，返回满足条件的数据表中的该字段的第一行的值。  如果需要返回整个列的数据，可以用：  $this->getField('id',true); // 获取id数组  如果传入多个字段的话，默认返回一个关联数组：  $User = M("User"); // 实例化User对象  // 获取所有用户的ID和昵称列表  $list = $User->getField('id,nickname');  这样返回的list是一个数组，键名是用户的id，键值是用户的昵称nickname  如果传入多个字段的名称，例如：  $list = $User->getField('id,nickname,email');  返回的是一个二维数组，类似select方法的返回结果，区别的是这个二维数组的键名是用户的id（准确的说是getField方法的第一个字段名）。  如果我们传入一个字符串分隔符：  $list = $User->getField('id,nickname,email',':');  那么返回的结果就是一个数组，键名是用户id，键值是 nickname:email的输出字符串。  getField方法还可以支持限制数量，例如：  $this->getField('id,name',5); // 限制返回5条记录  $this->getField('id',3); // 获取id数组 限制3条记录 |

4.数据更新

数据更新包括更新数据和更新字段方法

4.1更新数据

更新数据使用save方法，例如：

|  |
| --- |
| $User = M(‘User’);//实例化User模型  $data[‘name’] = ‘think’;  $data[‘email’] = [‘think@think.com’;](mailto:‘think@think.com’;)  $User->where(‘id=2’)->save($data);  也可以改成对象方式来操作：  $User->M(‘User’);  $User->name = ‘think’;  [$User->email=’think@think.com’](mailto:$User->email=’think@think.com’)  $User->where(‘id = 1’)->save(); |

数据对象赋值的方式，save方法无需传入数据，会自动识别。

注意：save方法的返回值是影响的记录数，如果返回false则表示更新出错，因此一定要用恒等来判断是否更新失败。

为了保证数据库的安全，避免出错更新整个数据表，如果没有任何更新条件，数据对象本身也不包含主键字段的话，save方法不会更新任何数据库的记录。

4.2更新字段值

更新字段使用setField方法，参考getField方法。

5.数据删除

ThinkPHP删除数据使用DELETE方法，例如：

|  |
| --- |
| $Form = M(‘Form’);  $From->delete(5); |

表示删除主键为5的数据，delete方法可以删除单个数据，也可以删除多个数据，这取决于删除条件，例如：

|  |
| --- |
| $User = M(‘User’);  $where[‘id’] = array(‘gt’,’5’);  $User->where($where)->delete(); |

delete方法的返回值是删除的记录数，如果返回值是false则表示SQL出错，返回值如果为0表示没有删除任何数据。

也可以用order和limit方法来限制要删除的个数，例如:

|  |
| --- |
| // 删除所有状态为0的5 个用户数据 按照创建时间排序  $User->where(‘status = 0’)->limit(5)->order(‘create\_time’)->delete(); |

为了避免错删数据，如果没有传入任何条件进行删除操作的话，不会执行删除操作，例如：

|  |
| --- |
| $User = M("User"); // 实例化User对象  $User->delete(); |

不会删除任何数据，如果你确实要删除所有的记录，除非使用下面的方式：

|  |
| --- |
| $User = M("User"); // 实例化User对象  $User->where('1')->delete(); |

三十 ActiveRecord——活动记录

三一 字段映射

字段映射功能可以让你在表单中隐藏真正的数据表字段，而不用担心放弃自动创建表单对象的功能。假设在User表里面有Username和email字段，我们需要映射成另外的字段，定义方式如下：

|  |
| --- |
| namespace Home\Model  use Think\Model  Class UserModel extends Model{  protected $\_map = array(  ‘name’ =>’username’, // 把表单中name映射到数据表的username字段  ‘mail’=>’email’ //把表单中的mail映射到数据表email字段  );  } |

这样，在表单里面就可以直接使用name和mail名称作为表单数据提交了。我们使用create方法创建数据对象的时候，会自动转换成定义的实际数据表字段。

字段映射还可以支持对主键的映射。

使用字段映射后，默认不会对读取的数据自动处理，

|  |
| --- |
| //实例化User模型  $User = M(‘User’);  $data = $User->find(3);  dump($data)  输出结果类似：  array(size=4)  'id' => int 3  'username' => string 'thinkphp'(length=8)  'email' => string 'thinkphp@gmail.com' (length=18)  'status' => int 1 |

这个时候取出的data数据包含的是实际的username和email字段。

如果我们需要在数据获取的时候自动处理的话，设置开启READ\_DATA\_MAP参数，

|  |
| --- |
| 'READ\_DATA\_MAP'=>true  这个时候，输出结果类似：  array(size=4)  'id' => int 3  'name' => string 'thinkphp'(length=8)  'mail' => string 'thinkphp@gmail.com' (length=18)  'status' => int 1 |

或者直接使用parseFieldsMap方法进行转换处理，例如：

|  |
| --- |
| // 实例化User模型  $User = D('User');  $data = $User->find(3);  $data = $User->parseFieldsMap($data); |

通过上面的两种方式后，无论是find还是select方法读取后的data数据中就包含了name和mail字段数据了，而不再有username和email字段数据了。

三二 查询语言

1.查询方式

ThinkPHP可以支持直接使用字符串作为查询条件，但是大多数情况推荐使用索引数组或者对象来作为查询条件，因为会更加安全。

1.1使用字符串作为查询条件

这是最传统的方式但安全性不高，例如：

|  |
| --- |
| $User = M(‘User’);  $User->where(‘status=1 AND id >5’)->select();  最后生成的SQL语句  SELECT \* FROM think\_user WHERE status=1 AND id >5; |

1.2使用数组作为查询条件

这种方式是最常用的查询方式：

|  |
| --- |
| $User = M(‘User’);  $where[‘status’]= 1;  $where[‘id’]=array(‘gt’,’5’);  $User->where($where)->select();  最后生成的SQL语句  SELECT \* FROM think\_user WHERE status=1 AND id >5; |

如果进行多字段查询，那么字段之间的默认逻辑关系是 逻辑与 AND，但是用下面的规则可以更改默认的逻辑判断，通过使用 \_logic 定义查询逻辑：

|  |
| --- |
| $User = M(‘User’);  $where[‘status’]= 1;  $where[‘id’]=array(‘gt’,’5’);  $where[‘\_logic’]=’or’;  $User->where($where)->select();  最后生成的SQL语句  SELECT \* FROM think\_user WHERE status=1 OR id >5; |

1.3 使用对象方式查询

这里以stdClass内置对象为例：

|  |
| --- |
| $User = M(‘User’);  $where->status = 1;  $where->name = ‘Lin’;  $User->where($where)->select(); |

使用对象方式查询和使用数组查询的效果是相同的，并且是可以互换的，大多数情况下，我们建议采用数组方式更加高效。

2.表达式查询

上面的查询条件仅仅是一个简单的相等判断，可以使用查询表达式支持更多的SQL查询语法，也是ThinkPHP查询语言的精髓，查询表达式的使用格式：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* $map['字段名'] = array('表达式','查询条件');

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

3. 快捷查询

快捷查询是一种多字段查询的简化方法，可以进一步简化查询条件的写法，在多个字段之间用 | 分隔表示OR查询，用 & 表示AND查询，可以实现下面的查询，例如：

3.1 不同字段相同的查询条件

|  |
| --- |
| $User = M(‘User’);  $map[‘name | title’] = ‘think’;  $User->where($map)->select();  相当于如下:  $map[‘name’] = ‘think’;  $map[‘title’]=’think’;  $map[‘\_logic’]=’or’;  $User->where($map)->select(); |

3.2 不同字段不同的查询条件

|  |
| --- |
| $User = M(‘User’);  $map[‘status|name’] = array(‘1’,’Lin’,’\_multi’=>true);  $User->where($map)->select();  相当于如下：  $map[‘status’] = 1;  $map[‘name’]=’Lin’;  $User->where($map)->select();  '\_multi'=>true必须加在数组的最后，表示当前是多条件匹配，这样查询条件就变成 status= 1 AND title = 'thinkphp' |

4.区间查询

ThinkPHP支持对某个字段的区间查询，例如：

|  |
| --- |
| $map[‘id’] = array(array(‘gt’,’5’),array(‘lt’,’10’));  得到的查询条件是： ( id >5) AND ( id < 10)  $map[‘id’] = array(array(‘lt’,’5’),array(‘gt’,’10’),’or’);  得到的查询条件是:(id<5) or (id>10);  最后一个可以是AND、 OR或者 XOR运算符，如果不写，默认是AND运算。 |

区间查询的条件可以支持普通查询的所有表达式，也就是说类似LIKE、GT和EXP这样的表达式都可以支持。另外区间查询还可以支持更多的条件，只要是针对一个字段的条件都可以写到一起，例如：

5.组合查询

组合查询的主体还是采用数组方式查询，只是加入了一些特殊的查询支持，包括字符串模式查询（\_string）、复合查询（\_complex）、请求字符串查询（\_query），混合查询中的特殊查询每次查询只能定义一个，由于采用数组的索引方式，索引相同的特殊查询会被覆盖。

5.1字符串模式查询

数组条件可以和字符串条件（采用\_string 作为查询条件）混合使用，例如：

|  |
| --- |
| $User = M(‘User’);  $map[‘id’] = array(‘neq’,’1’);  $map[‘name’]=’think’;  $map[‘\_string’]=’status=1 AND score >10’;  $User->where($map)->select();  最后得到的查询条件就成了：  ( `id` != 1 ) AND ( `name` = 'ok' ) AND ( status=1 AND score>10 ) |

5.2 请求字符串查询

请求字符串查询是一种类似于URL传参的方式，可以支持简单的条件相等判断。

|  |
| --- |
| $map[‘id’] = array(‘gt’,’100’);  $map[‘\_query’]=’status=1&score=100&\_logic=or’;  得到的查询条件是：  `id`>100 AND (`status` = '1' OR `score` = '100'); |

5.3 复合查询

复合查询相当于封装了一个新的查询条件，然后并入原来的查询条件之中，所以可以完成比较复杂的查询条件组装。例如：

|  |
| --- |
| $User = M(’User’);  $where[‘title’] = array(‘like’,’%think%’);  $where[‘name’] = array(‘like’,’%think%’);  $where[‘\_logic’]=’or’;  $map[‘\_complex’] = $where;  $map[‘id’] = array(‘gt’,1);  查询条件是:  ( id > 1) AND ( ( name like '%thinkphp%') OR ( title like '%thinkphp%') ) |

复合查询使用了\_complex作为子查询条件来定义，配合之前的查询方式，可以非常灵活的制定更加复杂的查询条件。很多查询方式可以相互转换，例如上面的查询条件可以改成：

|  |
| --- |
| $where['id'] = array('gt',1);  $where['\_string'] = ' (name like "%thinkphp%") OR ( title like "%thinkphp") '; |

最后生成的SQL语句是一致的。

6. 统计查询

在应用中我们经常会用到一些统计数据，例如当前所有（或者满足某些条件）的用户数、所有用户的最大积分、用户的平均成绩等等，ThinkPHP为这些统计操作提供了一系列的内置方法，包括：

| **方法** | **说明** |
| --- | --- |
| Count | 统计数量，参数是要统计的字段名（可选） |
| Max | 获取最大值，参数是要统计的字段名（必须） |
| Min | 获取最小值，参数是要统计的字段名（必须） |
| Avg | 获取平均值，参数是要统计的字段名（必须） |
| Sum | 获取总分，参数是要统计的字段名（必须） |

用法实例：

|  |
| --- |
| //实例化对象  $User = M(‘User’);  //获取用户数  $UserCount = $User->count();  //或者根据字段统计  $UserCount = $User->count(‘id’);  //获取用户最大积分  $UserMax = $User->min(‘score’);  //获取积分大于0的用户的最小积分  $UserMin = $User->where(‘score>0’)->max(‘score’);  //获取用户平均积分  $UserAvg = $User->avg(‘score’);  //获取用户总积分  $UserSum = $User->sum(‘score’); |

7. SQL查询

ThinkPHP内置的ORM和ActiveRecord模式实现了方便的数据存取操作，而且新版增加的连贯操作功能更是让这个数据操作更加清晰，但是ThinkPHP仍然保留了原生的SQL查询和执行操作支持，为了满足复杂查询的需要和一些特殊的数据操作，SQL查询的返回值因为是直接返回的Db类的查询结果，没有做任何的处理。

主要包括下面两个方法：

7.1 query方法

query 执行SQL查询操作用法 query($sql,$parse=false) 参数 sql（必须）：要查询的SQL语句 parse（可选）：是否需要解析SQL 返回值 如果数据非法或者查询错误则返回false。

否则返回查询结果数据集（同select方法）使用示例：

|  |
| --- |
| $Model = new Model() // 实例化一个model对象 没有对应任何数据表  $Model->query("select \* from think\_user where status=1"); |

如果你当前采用了分布式数据库，并且设置了读写分离的话，query方法始终是在读服务器执行，因此query方法对应的都是读操作，而不管你的SQL语句是什么。

7.2 execute方法

execute用于更新和写入数据的sql操作用法 execute($sql,$parse=false) 参数 sql（必须）：要执行的SQL语句 parse（可选）：是否需要解析SQL 返回值 如果数据非法或者查询错误则返回false 否则返回影响的记录数使用示例：

|  |
| --- |
| $Model = new Model() // 实例化一个model对象 没有对应任何数据表$Model->execute("update think\_user set name='thinkPHP' where status=1"); |

如果你当前采用了分布式数据库，并且设置了读写分离的话，execute方法始终是在写服务器执行，因此execute方法对应的都是写操作，而不管你的SQL语句是什么。

8. 动态查询

借助PHP5语言的特性，ThinkPHP实现了动态查询，核心模型的动态查询方法包括下面几种：

| **方法名** | **说明** | **举例** |
| --- | --- | --- |
| getBy | 根据字段的值查询数据 | 例如，getByName,getByEmail |
| getFieldBy | 根据字段查询并返回某个字段的值 | 例如，getFieldByName |

1.getBy动态查询

该查询方式针对数据表的字段进行查询。例如，User对象拥有id,name,email,address 等属性，那么我们就可以使用下面的查询方法来直接根据某个属性来查询符合条件的记录。

|  |
| --- |
| $user = $User->getByName('liu21st');  $user = [$User->getByEmail('liu21st@gmail.com');](mailto:$User->getByEmail('liu21st@gmail.com');)  $user = $User->getByAddress('中国深圳'); |

暂时不支持多数据字段的动态查询方法，请使用find方法和select方法进行查询。

2.getFieldBy动态查询

针对某个字段查询并返回某个字段的值，例如

|  |
| --- |
| $userId = $User->getFieldByName('liu21st','id'); |

表示根据用户的name获取用户的id值。

9.子查询

9.1使用buildSql方法

|  |
| --- |
| $subQuery = $model->field('id,name')->table('tablename')->group('field')->where($where)->order('status')->buildSql(); |

调用buildSql方法后不会进行实际的查询操作，而只是生成该次查询的SQL语句（为了避免混淆，会在SQL两边加上括号），然后我们直接在后续的查询中直接调用。

|  |
| --- |
| // 利用子查询进行查询  $model->table($subQuery.' a')->where()->order()->select() |

构造的子查询SQL可用于ThinkPHP的连贯操作方法，例如table where等

三三 自动验证

自动验证是ThinkPHP提供的数据验证方法，可以在使用create方法创建数据对象的时候自动进行数据验证。

1. 验证规则

数据验证可以进行数据类型、业务规则、安全判断等方面的验证操作。

数据验证有两种方式：

● 静态方式：在模型类里面通过$\_validate属性定义验证规则。

● 动态方式：使用模型类的validate方法动态创建自动验证规则。

无论是何种方式，验证规则的定义的统一的规则，定义格式为：

|  |
| --- |
| array(  array(验证字段1,验证规则,错误提示,[验证条件,附加规则,验证时间]),  array(验证字段2,验证规则,错误提示,[验证条件,附加规则,验证时间]),  ......  ) |

1. 验证字段（必须）

验证字段为需要验证的表单字段名称，这个字段不一定是数据库字段，也可以是表单的一些辅助字段。例如确认密码和验证码等等。有个别验证规则和字段无关的情况下，验证规则是可以随意设置的。例如expire有效期规则和表单字段无关的。如果定义了字段映射的话，这里的验证字段名称应该是实际的数据表字段而不是表单字段。

3. 验证规则（必须）

要进行验证的规则，需要结合附加规则，如果在使用正则验证的附加规则情况下，系统还内置了一些常用正则验证的规则，可以直接作为验证规则使用，包括：require 字段必须、email 邮箱、url URL地址、currency 货币、number 数字。

4. 错误提示（必须）

用于验证失败后的提示信息定义

5.验证条件（可选）

包含下面几种情况：

● self::EXISTS\_VALIDATE 或者0 存在字段就验证（默认）

● self::MUST\_VALIDATE 或者1 必须验证

● self::VALUE\_VALIDATE或者2 值不为空的时候验证

6.附件规则（可选）

配合验证规则使用，包括下面一些规则：

| **规则** | **说明** |
| --- | --- |
| regex | 正则验证，定义的验证规则是一个正则表达式（默认） |
| function | 函数验证，定义的验证规则是一个函数名 |
| callback | 方法验证，定义的验证规则是当前模型类的一个方法 |
| confirm | 验证表单中的两个字段是否相同，定义的验证规则是一个字段名 |
| equal | 验证是否等于某个值，该值由前面的验证规则定义 |
| notequal | 验证是否不等于某个值，该值由前面的验证规则定义（3.1.2版本新增） |
| in | 验证是否在某个范围内，定义的验证规则可以是一个数组或者逗号分割的字符串 |
| notin | 验证是否不在某个范围内，定义的验证规则可以是一个数组或者逗号分割的字符串（3.1.2版本新增） |
| length | 验证长度，定义的验证规则可以是一个数字（表示固定长度）或者数字范围（例如3,12 表示长度从3到12的范围） |
| between | 验证范围，定义的验证规则表示范围，可以使用字符串或者数组，例如1,31或者array(1,31) |
| notbetween | 验证不在某个范围，定义的验证规则表示范围，可以使用字符串或者数组（3.1.2版本新增） |
| expire | 验证是否在有效期，定义的验证规则表示时间范围，可以到时间，例如可以使用 2012-1-15,2013-1-15 表示当前提交有效期在2012-1-15到2013-1-15之间，也可以使用时间戳定义 |
| ip\_allow | 验证IP是否允许，定义的验证规则表示允许的IP地址列表，用逗号分隔，例如201.12.2.5,201.12.2.6 |
| ip\_deny | 验证IP是否禁止，定义的验证规则表示禁止的ip地址列表，用逗号分隔，例如201.12.2.5,201.12.2.6 |
| unique | 验证是否唯一，系统会根据字段目前的值查询数据库来判断是否存在相同的值，当表单数据中包含主键字段时unique不可用于判断主键字段本身 |

7. 验证时间

self::MODEL\_INSERT或者1新增数据时候验证

self::MODEL\_UPDATE或者2编辑数据时候验证

self::MODEL\_BOTH或者3全部情况下验证（默认）

这里的验证时间需要注意，并非只有这三种情况，你可以根据业务需要增加其他的验证时间。

8. 静态定义

在模型类里面预先定义好该模型的自动验证规则，我们称为静态定义。

举例说明，我们在模型类里面定义了$\_validate属性如下：

|  |
| --- |
| namespace Home\Model  use Think\Model  class UserModel extends Model{  protected $\_validate = array(  array(‘verify’,’require’,’验证码必须填写!’), //默认情况下用正则进行验证  array(‘name’,’’,’账号名称已经存在!’,’0’,’unique’,’1’) // 在新增的时候验证name字段是否唯一  array(‘value’,array(1,2,3),’值不在范围内’,’2’,’in’)// 当值不为空的时候判断是否在一个范围内  array(‘repassword’,’password’,’确认密码不正确’,’0’,’confirm’)// 验证确认密码是否和密码一致  array(‘password’,’checkpwd’,’密码格式不正确’,’0’,’function’)// 自定义函数验证密码格式  );  } |

定义好验证规则后，就可以在使用create方法创建数据对象的时候自动调用：

|  |
| --- |
| $User = D(‘User’);  if(!$User->create){  // 如果创建失败 表示验证没有通过 输出错误提示信息  exit($User->getError());  }else{  // 验证通过 可以进行其他数据操作  } |

在进行自动验证的时候，系统会对定义好的验证规则进行依次验证。如果某一条验证规则没有通过，则会报错，getError方法返回的错误信息（字符串）就是对应字段的验证规则里面的错误提示信息。

静态定义方式因为必须定义模型类，所以只能用D函数实例化模型

默认情况下，create方法是对表单提交的POST数据进行自动验证，如果你的数据来源不是表单post，仍然也可以进行自动验证，方法改进如下：

|  |
| --- |
| $User = D(‘User’);  $data = getData();  if(!$User->create($data)){  // 如果创建失败 表示验证没有通过 输出错误提示信息  exit($User->getError());  }else{  // 验证通过 可以进行其他数据操作  } |

一般情况下，create方法会自动判断当前是新增数据还是编辑数据（主要是通过表单的隐藏数据添加主键信息），你也可以明确指定当前创建的数据对象是新增还是编辑，例如：

|  |
| --- |
| $User = D("User"); // 实例化User对象  if (!$User->create($\_POST,1)){ // 指定新增数据  // 如果创建失败 表示验证没有通过 输出错误提示信息  exit($User->getError());  }else{  // 验证通过 可以进行其他数据操作  } |

create方法的第二个参数就用于指定自动验证规则中的验证时间，也就是说create方法的自动验证只会验证符合验证时间的验证规则。

我们在上面提到这里的验证时间并非只有这几种情况，你可以根据业务需要增加其他的验证时间，例如，你可以给登录操作专门指定验证时间为4。我们定义验证规则如下：

|  |
| --- |
| namespace Home\Model  use Think\Model  class UserModel extend Model{  protected $\_validate = array(  array(‘verify’,’require’,’验证码必须’), //都有时间验证  array(‘name’,’checkName’,’账号错误’,’1’,’function’,’4’), //登录时验证  array('password','checkPwd','密码错误！',1,'function',4), // 只在登录时候验证  );  } |

那么我们就可以在登录的时候使用

|  |
| --- |
| $User = D(‘User’);  if(!$user->create($\_POST,4)){ // 登录验证数据  // 验证没有通过 输出错误提示信息  exit($User->getError());  }else{  // 验证通过 执行登录操作  } |

9.动态验证

如果采用动态验证的方式，就比较灵活，可以根据不同的需要，在操作同一个模型的时候使用不同的验证规则，例如上面的静态验证方式可以改为：

|  |
| --- |
| $rules = array(  array('verify','require','验证码必须！'), //默认情况下用正则进行验证 array('name','','帐号名称已经存在！',0,'unique',1), // 在新增的时候验证name字段是否唯一  array('value',array(1,2,3),'值的范围不正确！',2,'in'), // 当值不为空的时候判断是否在一个范围内  array('repassword','password','确认密码不正确',0,'confirm'), // 验证确认密码是否和密码一致  array('password','checkPwd','密码格式不正确',0,'function'), // 自定义函数验证密码格式  );  $User = M("User"); // 实例化User对象  if (!$User->validate($rules)->create()){  // 如果创建失败 表示验证没有通过 输出错误提示信息 exit($User->getError());  }else{  // 验证通过 可以进行其他数据操作  }  动态验证不依赖模型类的定义，所以通常用M函数实例化模型就可以 |

10. 批量验证

系统支持数据的批量验证功能，只需要在模型类里面设置patchValidate属性为true（ 默认为false），

protected $patchValidate = true;

设置批处理验证后，getError() 方法返回的错误信息是一个数组，返回格式是：

array("字段名1"=>"错误提示1","字段名2"=>"错误提示2"... )

前端可以根据需要需要自行处理，例如转换成json格式返回：

|  |
| --- |
| $User = D("User"); // 实例化User对象  f (!$User->create()){  // 如果创建失败 表示验证没有通过 输出错误提示信息 $this->ajaxReturn($User->getError());  }else{  // 验证通过 可以进行其他数据操作  } |

三四 自动完成

自动完成是ThinkPHP提供用来完成数据自动处理和过滤的方法，使用create方法创建数据对象的时候会自动完成数据处理。

因此，ThinkPHP中使用create方法来创建数据对象是更加安全的方式，而不是直接通过add和save方法实现数据写入。

1.规则定义

自动完成通常用来完成默认字段写入，安全字段过滤以及业务逻辑的自动处理等，和自动验证的定义方式类似，自动完成的定义也支持静态定义和动态定义两种方式。

● 静态方式：在模型类里面通过$\_auto属性定义处理规则

● 动态方式：使用模型类的auto方法动态创建自动处理规则

两种规则的定义格式都采用：

|  |
| --- |
| array(  array(完成字段1,完成规则[,完成时间,附加规则]),  array(完成字段2,完成规则[,完成时间,附加规则]),  ..........  ); |

2. 完成字段（必须）

需要进行处理的数据表实际字段名称。

3. 完成规则（必须）

需要处理的规则，配合附件规则完成。

4.完成时间（可选）

设置自动完成的时间，包括：

| **设置** | **说明** |
| --- | --- |
| self::MODEL\_INSERT或者1 | 新增数据的时候处理（默认） |
| self::MODEL\_UPDATE或者2 | 更新数据的时候处理 |
| self::MODEL\_BOTH或者3 | 所有情况都进行处理 |

5.附加规则（可选）

包括：

| **规则** | **说明** |
| --- | --- |
| function | 使用函数，表示填充的内容是一个函数名 |
| callback | 回调方法 ，表示填充的内容是一个当前模型的方法 |
| field | 用其它字段填充，表示填充的内容是一个其他字段的值 |
| string | 字符串（默认方式） |
| ignore | 为空则忽略（3.1.2新增） |

5.静态定义

预先在模型类里面定义好自动完成的规则，我们称之为静态定义，例如：

|  |
| --- |
| namespace Home\Model  use Think\Model  Class UserModel extends Model{  protected $\_auto = arary(  array(‘staus’,’1’,’1’,’string’),// 新增的时候把status字段设置为1  array(‘password’,’MD5’,’3’,’function’)// 对password字段在新增和编辑的时候使md5函数处理  array(‘name’,’getName’,’3’,’callback’)// 对name字段在新增和编辑的时候回调getName方法  array(‘update\_time’,’time’,’2’,’function’)// 对update\_time字段在更新的时候写入当前时间戳  );  } |

然后，就可以在使用create方法创建数据对象的时候自动处理：

|  |
| --- |
| $User = D(‘User’);  $data = getData();  if(!$User->create($data)){  //如果创建失败 表示验证没有通过 输出错误提示信息  exit($User->getError());  }else{  //验证通过 写入新增数据  $User->add();  } |

如果你没有定义任何自动验证规则的话，则不需要判断create方法的返回值：

|  |
| --- |
| $User = D("User"); // 实例化User对象  $User->create(); // 生成数据对象  $User->add();// 新增用户数据  或者更简单的使用：  $User = D("User"); // 实例化User对象  $User->create()->add(); // 生成数据对象并写入数据 |

create方法默认情况下是根据表单提交的post数据生成数据对象，我们也可以根据其他的数据源来生成数据对象，你也可以明确指定当前创建的数据对象自动处理的时间是新增还是编辑数据，例如：

|  |
| --- |
| $User = D("User"); // 实例化User对象  $userData = getUserData(); // 通过方法获取用户数据  $User->create($userData,2); // 根据userData数据创建数据对象，并指定为更新数据$User->add(); |

create方法的第二个参数就用于指定自动完成规则中的完成时间，也就是说create方法的自动处理规则只会处理符合完成时间的自动完成规则。 create方法在创建数据的时候，已经自动过滤了非数据表字段数据信息，因此不需要担心表单会提交其他的非法字段信息而导致数据对象写入出错，甚至还可以自动过滤不希望用户在表单提交的字段信息（详见字段合法性过滤）。

3.1.2版本开始新增了ignore完成规则，这一规则表示某个字段如果留空的话则忽略，通常可用于修改用户资料时候密码的输入，定义如下：

|  |
| --- |
| array('password','',2,'ignore') |

表示password字段编辑的时候留空则忽略。

6. 动态完成

除了静态定义之外，我们也可以采用动态完成的方式来解决不同的处理规则。

|  |
| --- |
| $rules = array ( array('status','1'), // 新增的时候把status字段设置为1  array('password','md5',3,'function') , // 对password字段在新增和编辑的时候使md5函数处理  array('update\_time','time',2,'function'), // 对update\_time字段在更新的时候写入当前时间戳  );  $User = M('User');  $User->auto($rules)->create()->add(); |

7. 修改数据对象

在使用create方法创建好数据对象之后，此时的数据对象保存在内存中，因此仍然可以操作数据对象，包括修改或者增加数据对象的值，例如：

|  |
| --- |
| $User = D("User"); // 实例化User对象  $User->create(); // 生成数据对象  $User->status = 2; // 修改数据对象的status属性  $User->register\_time = NOW\_TIME; // 增加register\_time属性  $User->add(); // 新增用户数据 |

一旦调用了add方法（或者save方法），创建在内存中的数据对象就会失效，如果希望创建好的数据对象在后面的数据处理中再次调用，可以保存数据对象先，例如：

|  |
| --- |
| $User = D("User"); // 实例化User对象  $data = $User->create(); // 保存生成的数据对象  $User->add(); |

不过要记得，如果你修改了内存中的数据对象并不会自动更新保存的数据对象，因此下面的用法是错误的：

|  |
| --- |
| $User = D("User"); // 实例化User对象  $data = $User->create(); // 保存生成的数据对象  $User->status = 2; // 修改数据对象的status属性  $User->register\_time = NOW\_TIME; // 增加register\_time属性  $User->add($data); |

上面的代码我们修改了数据对象，但是仍然写入的是之前保存的数据对象，因此对数据对象的更改操作将会无效。

三五 虚拟模型

虚拟模型是指虽然是模型类，但并不会真正的操作数据库的模型。简历模型类但又不需要进行数据库操作，仅仅是借助模型来实现一些业务逻辑。可以借助虚拟模型来完成。虚拟模型不会自动连接数据库，因此也不会自动检测数据表和字段信息，有两种方式可以定义虚拟模型：

1. 继承Model类

|  |
| --- |
| namespace Home\Model;  Class UserModel extends \Think\Model {  Protected $autoCheckFields = false;  } |

设置autoCheckFields属性为false后，就会关闭字段信息的自动检测，因为ThinkPHP采用的是惰性数据库连接，只要你不进行数据库查询操作，是不会连接数据库的。

2.不继承Model类

|  |
| --- |
| namespace Home\Model;  Class UserModel {  } |

这种方式下面自定义模型类就是一个单纯的业务逻辑类，不能再使用模型的CURD操作方法，但是可以实例化其他的模型类进行相关操作，也可以在需要的时候直接实例化Db类进行数据库操作。

三六 模型分层

ThinkPHP支持模型的分层 ，除了Model层之外，我们可以项目的需要设计和创建其他的模型层

通常情况下，不同的分层模型仍然是继承系统的\Think\Model类或其子类，所以，其基本操作和Model类的操作是一致的。

例如，在Home模块的设计中需要区分数据层、逻辑层、服务层，我们可以在模块目录下创建Model、Logic、Service目录，把对用户表的所有模型操作分为三层：

● 数据层：Home\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

● 逻辑层：Home\Logic\UserLogic 用于定义用户相关的业务逻辑

● 服务层：Home\Service\UserService 用于定于用户先关的服务接口等

三个模型定义如下：

Model类：Home\Model\UserModel.class.php

|  |
| --- |
| namespace Home\Model  use Think\Model  calss UserModel extends Model{  } |

实例化方法$User = D(‘User’);

Logic类：Home\UserLogic\UserLogic.class.php

|  |
| --- |
| namespace Home\Logic  use Think\Model  calss UserLogic extends Model{  } |

实例化方法$User = D(‘User’,’Logic’);

Api类：Home\Api\UserApi.class.php

|  |
| --- |
| namespace Home\Api  use Think\Model  calss UserApi extends Model{  } |

实例化方法$User =D(‘User’,’Api’);

D方法默认操作的模型层由DEFAULT\_M\_LAYER参数配置，我们可以改变默认操作的模型层为Logic层，例如：

|  |
| --- |
| 'DEFAULT\_M\_LAYER' => 'Logic', // 默认的模型层名称 |

这样，当我们调用：

|  |
| --- |
| $User = D('User'); |

的时候其实是实例化的 UserLogic类，而不是UserModel类。

三七 视图模型

视图通常是指数据库的视图，视图是一个虚拟表，其内容由查询定义。同真实的表一样，视图包含一系列带有名称的列和行数据。但是，视图并不在数据库中以存储的数据值集形式存在。行和列数据来自由定义视图的查询所引用的表，并且在引用视图时动态生成。对其中所引用的基础表来说，视图的作用类似于筛选。定义视图的筛选可以来自当前或其它数据库的一个或多个表，或者其它视图。分布式查询也可用于定义使用多个异类源数据的视图。如果有几台不同的服务器分别存储组织中不同地区的数据，而您需要将这些服务器上相似结构的数据组合起来，这种方式就很有用。视图在有些数据库下面并不被支持，但是ThinkPHP模拟实现了数据库的视图，该功能可以用于多表联合查询。非常适合解决HAS\_ONE 和 BELONGS\_TO 类型的关联查询。

要定义视图模型，只需要继承Think\Model\ViewModel，然后设置viewFields属性即可。

例如下面的例子，我们定义了一个BlogView模型对象，其中包括了Blog模型的id、name、title和User模型的name，以及Category模型的title字段，我们通过创建BlogView模型来快速读取一个包含了User名称和类别名称的Blog记录（集）。

|  |
| --- |
| namespace Home\Model  use Think\Model\ViewModel  class BlogViewModel extends ViewModel{  public $viewFields = array(  ‘Blog’ =>array(‘id’,’name’,’title’),  ‘Category’=>array(‘title’=>’category\_name’,’\_on’=>’Blog.category\_id=Category.id’),  ‘User’=>array(‘name’=>’username’,’\_on’=>’Blog.user\_id=User.id’);  );  } |

$viewFields 属性表示视图模型包含的字段，每个元素定义了某个数据表或者模型的字段。

|  |
| --- |
| 'Blog'=>array('id','name','title');  表示BlogView视图模型要包含Blog模型中的id、name和title字段属性，这个其实很容易理解，就和数据库的视图要包含某个数据表的字段一样。而Blog相当于是给Blog模型对应的数据表定义了一个别名。  默认情况下会根据定义的名称自动获取表名，如果希望指定数据表，可以使用：  '\_table'=>"test\_db.test\_table"  如果希望给当前数据表定义另外的别名，可以使用  '\_as'=>'myBlog' |

BlogView视图模式除了包含Blog模型之外，还包含了Category和User模型，下面的定义：

|  |
| --- |
| 'Category'=>array('title'=>'category\_name'); |

和上面类似，表示BlogView视图模型还要包含Category模型的title字段，因为视图模型里面已经存在了一个title字段，所以我们通过

|  |
| --- |
| 'title'=>'category\_name' |

把Category模型的title字段映射为category\_name字段，如果有多个字段，可以使用同样的方式添加。

可以通过\_on来给视图模型定义关联查询条件，例如：

|  |
| --- |
| '\_on'=>'Blog.category\_id=Category.id' |

理解之后，User模型的定义方式同样也就很容易理解了。

|  |
| --- |
| Blog.categoryId=Category.id AND Blog.userId=User.id |

最后，我们把视图模型的定义翻译成SQL语句就更加容易理解视图模型的原理了。假设我们不带任何其他条件查询全部的字段，那么查询的SQL语句就是

|  |
| --- |
| Select  Blog.id as id,  Blog.name as name,  Blog.title as title,  Category.title as category\_name,  User.name as username  from think\_blog Blog JOIN think\_category Category JOIN think\_user User  where Blog.category\_id=Category.id AND Blog.user\_id=User.id |

视图模型的定义并不需要先单独定义其中的模型类，系统会默认按照系统的规则进行数据表的定位。如果Blog模型并没有定义，那么系统会自动根据当前模型的表前缀和后缀来自动获取对应的数据表。也就是说，如果我们并没有定义Blog模型类，那么上面的定义后，系统在进行视图模型的操作的时候会根据Blog这个名称和当前的表前缀设置（假设为Think\_ ）获取到对应的数据表可能是think\_blog。

ThinkPHP还可以支持视图模型的JOIN类型定义，我们可以把上面的视图定义改成：

|  |
| --- |
| public $viewFields = array(  'Blog'=>array('id','name','title','\_type'=>'LEFT'), 'Category'=>array('title'=>'category\_name','\_on'=>'Category.id=Blog.category\_id','\_type'=>'RIGHT'),  'User'=>array('name'=>'username','\_on'=>'User.id=Blog.user\_id'),  ); |

需要注意的是，这里的\_type定义对下一个表有效，因此要注意视图模型的定义顺序。Blog模型的

|  |
| --- |
| '\_type'=>'LEFT' |

针对的是下一个模型Category而言，通过上面的定义，我们在查询的时候最终生成的SQL语句就变成：

|  |
| --- |
| Select  Blog.id as id,  Blog.name as name,  Blog.title as title,  Category.title as category\_name,  User.name as username  from think\_blog Blog LEFT JOIN think\_category Category ON Blog.category\_id=Category.id RIGHT JOIN think\_user User ON Blog.user\_id=User.id |

我们可以在试图模型里面定义特殊的字段，例如下面的例子定义了一个统计字段

|  |
| --- |
| 'Category'=>array('title'=>'category\_name','COUNT(Blog.id)'=>'count','\_on'=>'Category.id=Blog.category\_id'), |

2.视图查询

|  |
| --- |
| $Model = D("BlogView");  $Model->field('id,name,title,category\_name,username')->where('id>10')->order('id desc')->select(); |

看起来和普通的模型操作并没有什么大的区别，可以和使用普通模型一样进行查询。如果发现查询的结果存在重复数据，还可以使用group方法来处理。

|  |
| --- |
| $Model->field('id,name,title,category\_name,username')->order('id desc')->group('id')->select(); |

我们可以看到，即使不定义视图模型，其实我们也可以通过方法来操作，但是显然非常繁琐。而定义了视图模型之后，所有的字段会进行自动处理，添加表别名和字段别名，从而简化了原来视图的复杂查询。如果不使用视图模型，也可以用连贯操作的JOIN方法实现相同的功能。

三八 关联模型

1.关联关系

通常我们指的关联关系有三种

|  |
| --- |
| 一对一关联：ONE\_TO\_ONE,包括HAS\_ONE和BELONGS\_TO  一对多关联：ONE\_TO\_MANY,包括HAS\_MANS和BELONGS\_TO  多对多关联：MANY\_TO\_MANY |

关联关系必然有一个参照表

● 有一个员工档案管理系统项目，这个项目要包括下面的一些数据表：基本信息表、员工档案表、部门表、项目组表、银行卡表（用来记录员工的银行卡资料）。

● 这些数据表之间存在一定的关联关系，我们以员工基本信息表为参照来分析和其他表之间的关联：

● 每个员工必然有对应的员工档案资料，所以属于HAS\_ONE关联；

● 每个员工必须属于某个部门，所以属于BELONGS\_TO关联；

● 每个员工可以有多个银行卡，但是每张银行卡只可能属于一个员工，因此属于HAS\_MANY关联；

● 每个员工可以同时在多个项目组，每个项目组同时有多个员工，因此属于MANY\_TO\_MANY关联；

分析清楚数据表之前的关联关系后，我们才可以进行关联定义和关联操作。

2.关联定义

ThinkPHP可以很轻松的完成数据表的关联CURD操作，目前支持的关联关系包括下面四种：HAS\_ONE、BELONGS\_TO、HAS\_MANY和MANY\_TO\_MANY。

一个模型根据业务模型的复杂程度可以同时定义多个关联，不受限制。所有关联定义都统一在模型类的$\_link成员变量内定义，并且支持动态定义。要支持关联操作，模型类必须继承Think\Model\RelationModel类，关联定义的格式如下：

|  |
| --- |
| namespace Home\Model  use Think\Model\RelationModel  class UserModel extends RelationModel{  protected $\_link = array(  ‘关联1’ => array(  ‘关联属性1’=>’定义’,  ‘关联属性N’=>’定义’,  ),  ‘关联2’ => array(  ‘关联属性1’=>’定义’,  ‘关联属性N’=>’定义’,  ),  ‘关联3’=> HAS\_ONE, //快捷定义  .......  );  } |

3. HAS\_ONE

HAS\_ONE关联表示当前模型拥有一个子对象，例如，每个员工都有一个人事档案。我们可以建立一个用户模型UserModel，并且添加如下关联定义：

|  |
| --- |
| namespace Home\Model;  use Think\Model\RelationModel;  class UserModel extends RelationModel{  protected $\_link = array(  'Profile'=> self::HAS\_ONE,  );  } |

上面是最简单的方式，表示其遵循了系统内置的数据库规范，完整的定义方式是：

|  |
| --- |
| namespace Home\Model;  use Think\Model\RelationModel;  class UserModel extends RelationModel{  protected $\_link = array(  'Profile'=>array(  'mapping\_type' => self::HAS\_ONE,  'class\_name' => 'Profile',  // 定义更多的关联性  ……  ),  );  } |

关联HAS\_ONE支持的关联属性有：

|  |  |
| --- | --- |
| 属性名 | 含义 |
| mapping\_type | 关联类型，在HAS\_ONE管理内必须使用HAS\_ONE常量定义 |
| class\_name | 要关联的模型类名，例如，class\_name 定义为Profile的话则表示和另外的Profile模型类关联，这个Profile模型类是无需定义的，系统会自动定位到相关的数据表进行关联。 |
| mapping\_name | 关联的映射名称，用于获取数据用。  该名称不要和当前模型的字段有重复，否则会导致关联数据获取的冲突。如果mapping\_name没有定义的话，会取class\_name的定义作为mapping\_name。如果class\_name也没有定义，则以数组的索引作为mapping\_name。 |
| foreign\_name | 关联的外键名称  外键的默认规则是当前数据对象名称\_id，例如： UserModel对应的可能是表think\_user （注意：think只是一个表前缀，可以随意配置） 那么think\_user表的外键默认为 user\_id，如果不是，就必须在定义关联的时候显式定义 foreign\_key 。 |
| condition | 关联条件  关联查询的时候会自动带上外键的值，如果有额外的查询条件，可以通过定义关联的condition属性。 |
| mapping\_fields | 关联要查询的字段  默认情况下，关联查询的关联数据是关联表的全部字段，如果只是需要查询个别字段，可以定义关联的mapping\_fields属性。 |
| as\_fields | 直接把关联的字段值映射成数据对象中的某个字段  这个特性是ONE\_TO\_ONE 关联特有的，可以直接把关联数据映射到数据对象中，而不是作为一个关联数据。当关联数据的字段名和当前数据对象的字段名称有冲突时，还可以使用映射定义。 |

三九 高级模型

高级模型提供了更多的查询功能和模型增强功能，利用了模型类的扩展机制实现。需要继承Think\Model\AdvModel类或者采用动态模型

示例都假设UserModel类继承自Think\Model\AdvModel类：

1.字段过滤

基础模型类内置有数据自动完成功能，可以对字段进行过滤，但是必须通过Creare方法调用才能生效。高级模型类的字段过滤功能却可以不受create方法的调用限制，可以在模型里面定义各个字段的过滤机制。

字段过滤机制的设置方式只需要再Model类里面添加$\_filter属性，并且加入过滤因子，格式如下：

|  |
| --- |
| protected $\_filter = array(  ‘过滤的字段’ => array(‘写入过滤规则’,’读取过滤规则’,是否传入整个数据对象)  );  过滤的规则是一个函数，如果设置传入整个数据对象，那么函数的参数就是整个数据对象，默认是传入数据对象中该字段的值。 |

举例说明，例如我们需要在发表文章的时候对文章内容进行安全过滤，并且希望在读取的时候进行截取前面255个字符，那么可以设置：

|  |
| --- |
| protected $\_filter = array(  'content'=>array('contentWriteFilter','contentReadFilter'),  ) |

其中，contentWriteFilter是自定义的对字符串进行安全过滤的函数，而contentReadFilter是自定义的一个对内容进行截取的函数。通常我们可以在项目的公共函数文件里面定义这些函数。

2.序列化字段

序列化字段功能可以把简单的数据表字段完成复杂的表单数据验证，尤其是动态的表单数据字段。使用序列化字段的功能需在模型中定义serializeField属性，定义格式如下：

|  |
| --- |
| protected $serializeFelid = array(  ‘info’ => array(‘name’,’email’,’address’),  ); |

Info是数据表中的实际存在的字段，保存到其中的值是name、email和address三个表单字段的序列化结果。序列化字段功能可以在数据写入的时候进行自动序列化，并且在读出数据表的时候自动反序列化，这一切都无需手动进行。

下面还是是User数据表为例，假设其中并不存在name、email和address字段，但是设计了一个文本类型的info字段，那么下面的代码是可行的：

|  |
| --- |
| $User = D(‘User’);  $User->name = ‘Think’;  $User->email = [‘Think@think.com’](mailto:‘Think@think.com’);  $User->address = ‘China’;  //把数据对象添加到数据库 name email和address会自动序列化后保存到info字段  $User->add();  //查询用户数据信息  $User->find(8);  //查询结果会自动把info字段的值反序列化后生成name、email和address属性  //输出序列化字段  echo $User->name;  echo $User->email;  echo $User->address; |

3.文本字段

ThinkPHP支持数据模型中的个别字段采用文本方式存储，这些字段就称为文本字段，通常可以用于某些Text或者Blob字段，或者是经常更新的数据表字段。

要使用文本字段非常简单，只要在模型里面定义blobFields属性就行了。例如，我们需要对Blog模型的content字段使用文本字段，那么就可以使用下面的定义：

|  |
| --- |
| Protected $blobFields = array('content'); |

系统在查询和写入数据库的时候会自动检测文本字段，并且支持多个字段的定义。

需要注意的是：对于定义的文本字段并不需要数据库有对应的字段，完全是另外的。而且，暂时不支持对文本字段的搜索功能。

4.只读字段

只读字段是用来保护某些特殊的字段值不被更改，这个字段的值一旦写入，就无法更改。要使用只读字段的功能，只需要在模型中定义readonlyField属性：

|  |
| --- |
| protected $readonlyField = array(‘name’,’email’);  例如，上面定义了当前模型的name和email字段为只读字段，不允许被更改。也就是说当执行save方法之前会自动过滤到只读字段的值，避免更新到数据库。  $User = D("User"); // 实例化User对象  $User->find(8); // 更改某些字段的值  $User->name = 'TOPThink'  ;$User->email = ['Topthink@gmail.com';](mailto:'Topthink@gmail.com';)  $User->address = '上海静安区';  // 保存更改后的用户数据  $User->save(); |

事实上，由于我们对name和email字段设置了只读，因此只有address字段的值被更新了，而name和email的值仍然还是更新之前的值。

5.悲观锁和乐观锁

业务逻辑的实现过程中，往往需要保证数据访问的排他性。如在金融系统的日终结算处理中，我们希望针对某个时间点的数据进行处理，而不希望在结算进行过程中（可能是几秒种，也可能是几个小时），数据再发生变化。此时，我们就需要通过一些机制来保证这些数据在某个操作过程中不会被外界修改，这样的机制，在这里，也就是所谓的 “ 锁 ” ，即给我们选定的目标数据上锁，使其无法被其他程序修改。 ThinkPHP支持两种锁机制：即通常所说的 “ 悲观锁（ Pessimistic Locking ） ”和 “ 乐观锁（ Optimistic Locking ） ” 。

5.1悲观锁

悲观锁，正如其名，它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。 通常是使用for update子句来实现悲观锁机制。

ThinkPHP支持悲观锁机制，默认情况下，是关闭悲观锁功能的，要在查询和更新的时候启用悲观锁功能，可以通过使用之前提到的查询锁定方法，例如：

|  |
| --- |
| $User->lock(true)->save($data);// 使用悲观锁功能 |

5.2 乐观锁

相对悲观锁而言，乐观锁机制采取了更加宽松的加锁机制。悲观锁大多数情况下依靠数据库的锁机制实现，以保证操作最大程度的独占性。但随之而来的就是数据库性能的大量开销，特别是对长事务而言，这样的开销往往无法承受。 如一个金融系统，当某个操作员读取用户的数据，并在读出的用户数据的基础上进行修改时（如更改用户帐户余额），如果采用悲观锁机制，也就意味着整个操作过程中（从操作员读出数据、开始修改直至提交修改结果的全过程，甚至还包括操作员中途去煮咖啡的时间），数据库记录始终处于加锁状态，可以想见，如果面对几百上千个并发，这样的情况将导致怎样的后果。乐观锁机制在一定程度上解决了这个问题。乐观锁，大多是基于数据版本（ Version ）记录机制实现。何谓数据版本？即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个 “version” 字段来实现。

ThinkPHP也可以支持乐观锁机制，要启用乐观锁，只需要继承高级模型类并定义模型的optimLock属性，并且在数据表字段里面增加相应的字段就可以自动启用乐观锁机制了。默认的optimLock属性是lock\_version，也就是说如果要在User表里面启用乐观锁机制，只需要在User表里面增加lock\_version字段，如果有已经存在的其它字段作为乐观锁用途，可以修改模型类的optimLock属性即可。如果存在optimLock属性对应的字段，但是需要临时关闭乐观锁机制，把optimLock属性设置为false就可以了。

6. 延迟更新

我们经常需要给某些数据表添加一些需要经常更新的统计字段，例如用户的积分、文件的下载次数等等，而当这些数据更新的频率比较频繁的时候，数据库的压力也随之增大不少，我们可以利用高级模型的延迟更新功能缓解。

延迟更新功能是指我们可以给统计字段的更新设置一个延迟时间，在这个时间段内所有的更新会被累积缓存起来，然后定时地统一更新数据库。这比较适合某个字段经常需要递增或者递减，并且对实时性要求没有那么严格的情况。我们先来看递增的情况，如果我们需要给会员累积积分，可以使用

|  |
| --- |
| $User = D(‘User’);  $User->where(‘id=3’)->setInc(‘score’,10); //用户的积分+10;  $User->where(‘id=3’)->setInc(‘score’,30); //用户的积分+30; |

上面的操作更新了两次用户积分，并且都实时保存到数据库如果我们使用延迟更新方法，例如下面对用户的积分延迟更新60秒

|  |
| --- |
| $User->where('id=3')->setLazyInc("score",10,60);  $User->where('id=3')->setLazyInc("score",30,60);  $User->where('id=3')->setLazyInc("score",10,60); |

那么60秒内执行的所有积分更新操作都会被延迟，实际会在60秒后统一更新积分到数据库，而不是每次都更新数据库。临时积分会被累积并缓存起来，最后到了延迟更新时间，再统一更新。相当于在60秒后执行了：

|  |
| --- |
| $User->where('id=3')->setInc("score",50); |

效果是等效。区别在于用户数据库中的积分不是实时的。同样，还可以使用setLazyDec进行延迟更新操作。

7. 数据分表

对于大数据量的应用，经常会对数据进行分表，有些情况是可以利用数据库的分区功能，但并不是所有的数据库或者版本都支持，因此我们可以利用ThinkPHP内置的数据分表功能来实现。帮助我们更方便的进行数据的分表和读取操作。

和数据库分区功能不同，内置的数据分表功能需要根据分表规则手动创建相应的数据表。

在需要分表的模型中定义partition属性即可。

|  |
| --- |
| protected $partition = array(  'field' => 'name',// 要分表的字段 通常数据会根据某个字段的值按照规则进行分表  'type' => 'md5',// 分表的规则 包括id year mod md5 函数 和首字母  'expr' => 'name',// 分表辅助表达式 可选 配合不同的分表规则  'num' => 'name',// 分表的数目 可选 实际分表的数量  ); |

定义好了分表属性后，我们就可以来进行CURD操作了，唯一不同的是，获取当前的数据表不再使用getTableName方法，而是使用getPartitionTableName方法，而且必须传入当前的数据。然后根据数据分析应该实际操作哪个数据表。因此，分表的字段值必须存在于传入的数据中，否则会进行联合查询。

8. 返回类型

系统默认的数据库查询返回的是数组，我们可以给单个数据设置返回类型，以满足特殊情况的需要，例如：

|  |
| --- |
| $User = M("User"); // 实例化User对象  // 返回结果是一个数组数据  $data = $User->find(6);  // 返回结果是一个stdClass对象  $data = $User->returnResult($data, "object");  // 还可以返回自定义的类  $data = $User->returnResult($data, "User"); |

返回自定义的User类，类的架构方法的参数是传入的数据。例如：

|  |
| --- |
| Class User {  public function \_\_construct($data){  // 对$data数据进行处理  }  } |

四十 模板定义

每个模块的模板文件都是独立的，ThinkPHP对模板文件进行目录划分，默认的模板文件定义规则:

视图目录/[模板主题/]控制器名/操作名+模板后缀

默认的视图目录是模块的View目录（模块可以有多个视图文件目录，这取决于你的应用需要），框架的默认视图文件后缀是.html。 新版模板主题默认是空（表示不启用模板主题功能）。

在每个模板主题下面，是以模块下面的控制器名为目录，然后是每个控制器的具体操作模板文件，例如：

User控制器的add操作 对应的模板文件就应该是：View/User/add.html

如果你的默认视图层不是View，例如：

|  |
| --- |
| ‘DEFAULT\_V\_LAYER’ => ‘Template’, //设置的默认视图名称 |

那么，对应的模板文件就变成了：Template/User/add.html。

模板文件的默认后缀的情况是.html，也可以通过 TMPL\_TEMPLATE\_SUFFIX 来配置成其他的。例如，我们可以配置：

|  |
| --- |
| ‘TMPL\_TEMPLATE\_SUFFIX’ =>’.tpl’, //设置视图后缀 |

定义后，User控制器的add操作 对应的模板文件就变成是： View/User/add.tpl

如果觉得目录结构太深，可以通过设置 TMPL\_FILE\_DEPR 参数来配置简化模板的目录层次，例如设置：

|  |
| --- |
| 'TMPL\_FILE\_DEPR'=>'\_' |

默认的模板文件就变成了：View/User\_add.html

除了采用系统默认的模板定位规则之外，我们可以在模块的配置文件中单独定义视图目录，例如：

|  |
| --- |
| 'VIEW\_PATH'=>'./Theme/' |

把视图目录指定到最外层的Theme目录下面，而不是放到当前模块的View目录下面。

原来的./Application/Home/User/add.html变成了./Theme/Home/User/add.html。

利用这一功能可以让很多系统的模板目录独立出来更加方便修改。

四一 模板主题

一个模块如果需要支持多套模板文件的话，就可以使用模板主题功能。默认情况下，没有开启模板主题功能，如果需要开启，设置 DEFAULT\_THEME 参数即可：

|  |
| --- |
| //设置默认的模板主题  ‘DEFAULT\_THEME’ => ’default’, |

采用模板主题后，需要在视图目录下面创建对应的主题目录，和不启用模板主题的情况相比，模板文件只是多了一层目录：

|  |
| --- |
| View/User/add.html  View/default/User/add.html |

在视图渲染输出之前，我们可以通过动态设置来改变需要使用的模板主题。

|  |
| --- |
| $this->theme(‘default’)->display(‘add’); |

四二 模板赋值

如要在模板中输出变量，必须在控制器中把变量传递给模板，系统提供了assign方法对模板变量赋值。无论何种变量类型都是用统一的assign赋值。

|  |
| --- |
| $this->assign(‘name’,$value);  //下面的写法等效于  $this->name = $value |

assign方法必须在display()方法和show()方法之前调用。并且系统只会输出设定的变量。其它变量不会输出（系统变量例外），一定程度上保证了变量的安全性。

系统变量可以通过特殊的标签输出，无需赋值模板变量

赋值后，就可以在模板中输出变量，如果使用内置的模板，就可以{$name};

如果要同时输出多个模板变量，可以使用下面的方式：

|  |
| --- |
| $array[‘name’] = ‘think’;  $array[‘email’] = [‘think@think.com’;](mailto:‘think@think.com’;)  $array[‘address’] = ‘China’;  $this->assign($array);  这样，就可以在模板文件中同时输出name、email和phone三个变量。 |

模板变量的输出根据不同的模板引擎有不同的方法，我们在后面会专门讲解内置模板引擎的用法。如果你使用的是PHP本身作为模板引擎的话 ，就可以直接在模板文件里面输出了： <?php echo $name.'['.$email.''.$phone.']';?>

如果采用内置的模板引擎，可以使用： {$name} [ {$email} {$phone} ] 输出同样的内容。

四三 模板渲染

模板定义后就可以渲染模板输出，系统也支持直接渲染内容输出，模板赋值必须在模板渲染之前操作。

渲染模板输出最常用的是使用display方法，调用格式：

display(‘[模板文件]’,’[字符编码]’,’[输出类型]’);

1. 模板文件的写法支持以下写法

|  |  |
| --- | --- |
| 用法 | 描述 |
| 不带任何参数 | 自动定位当前操作的模板文件 |
| [模块@][控制器名:][操作名] | 常用写法，支持跨模块 模板主题可以和theme方法配合 |
| 完整的模板文件名 | 直接使用完整的模板文件名（包括模板后缀） |

调用实例：

|  |
| --- |
| $this->display(); //不带任何参数 |

表示系统会按照默认规则自动定位模板文件，其规则是：

如果当前没有启用模板主题则定位到：当前模块/默认视图目录/当前控制器/当前操作.html 如果有启用模板主题则定位到：当前模块/默认视图目录/当前主题/当前控制器/当前操作.html

果有更改TMPL\_FILE\_DEPR设置（假设 'TMPL\_FILE\_DEPR'=>'\_'）的话，则上面的自动定位规则变成： 当前模块/默认视图目录/当前控制器\_当前操作.html 和 当前模块/默认视图目录/当前主题/当前控制器\_当前操作.html。

所以通常display方法无需带任何参数即可输出对应的模板，这是模板输出的最简单的用法。

如果没有按照模板定义规则来定义模板文件（或者需要调用其他控制器下面的某个模板），可以使用：

|  |
| --- |
| // 指定模板输出  $this->display('edit'); |

表示调用当前模块下面的edit模板

|  |
| --- |
| $this->display('Member:read'); |

表示调用Member模块下面的read模板。

如果我们使用了模板主题功能，那么也可以支持跨主题调用，使用：

|  |
| --- |
| $this->theme('blue')->display('User:edit'); |

表示调用blue主题下面的User控制器的edit模板。

如果你不希望每个主题都重复定义一些相同的模版文件的话，还可以启用差异主题定义方式，设置：

|  |
| --- |
| 'TMPL\_LOAD\_DEFAULTTHEME'=>true |

设置后，如果blue主题下面不存在edit模板的话，就会自动定位到默认主题中的edit模板。

渲染输出不需要写模板文件的路径和后缀，确切地说，这里面的控制器和操作并不一定需要有实际对应的控制器和操作，只是一个目录名称和文件名称而已，例如，你的项目里面可能根本没有Public控制器，更没有Public控制器的menu操作，但是一样可以使用

|  |
| --- |
| $this->display('Public:menu'); |

输出这个模板文件。理解了这个，模板输出就清晰了。

display方法支持在渲染输出的时候指定输出编码和类型，例如，可以指定编码和类型：

|  |
| --- |
| $this->display('read', 'utf-8', 'text/xml'); |

表示输出XML页面类型（配合你的应用需求可以输出很多类型）。

事情总有特例，如果的模板目录是自定义的，或者根本不需要按模块进行分目录存放，那么默认的display渲染规则就不能处理，这个时候，我们就需要使用另外一种方式来应对，直接传入模板文件名即可，例如：

|  |
| --- |
| $this->display('./Template/Public/menu.html'); |

这种方式需要指定模板路径和后缀，这里的Template/Public目录是位于当前项目入口文件位置下面。如果是其他的后缀文件，也支持直接输出，例如：$this->display('./Template/Public/menu.tpl');

只要./Template/Public/menu.tpl是一个实际存在的模板文件。

四四 获取模板地址

为了更方便的输出模板文件，新版封装了一个T函数用于生成模板文件名。

T([资源://][模块@][主题/][控制器/]操作,[视图分层]);

T函数的返回值是一个完整的模板文件名，可以直接用于display和fetch方法进行渲染输出。

|  |
| --- |
| T('Public/menu');// 返回 当前模块/View/Public/menu.html  T('blue/Public/menu');// 返回 当前模块/View/blue/Public/menu.html  T('Public/menu','Tpl');// 返回 当前模块/Tpl/Public/menu.html  T('Public/menu');// 如果TMPL\_FILE\_DEPR 为 \_ 返回 当前模块/Tpl/Public\_menu.html  T('Public/menu');// 如果TMPL\_TEMPLATE\_SUFFIX 为.tpl 返回 当前模块/Tpl/Public/menu.tpl  T('Admin@Public/menu');// 返回 Admin/View/Public/menu.html  T('Extend://Admin@Public/menu');// 返回 Extend/Admin/View/Public/menu.html (Extend目录取决于AUTOLOAD\_NAMESPACE中的配置） |

在display方法中直接使用T函数：

|  |
| --- |
| // 使用T函数输出模板  $this->display(T('Admin@Public/menu')); |

T函数可以输出不同的视图分层模板。

四五 获取内容

如果需要获取渲染模板的输出内容而不是直接输出，可以使用fetch方法。

fetch方法的用法和display基本一致（只是不需要指定输出编码和输出类型）：

fetch('模板文件');

模板文件的调用方法和display方法完全一样，区别就在于fetch方法渲染后不是直接输出，而是返回渲染后的内容，例如：

|  |
| --- |
| $content = $this->fetch('Member:edit'); |

使用fetch方法获取渲染内容后，你可以进行过滤和替换等操作，或者用于对输出的复杂需求。

渲染内容:

如果你没有定义任何模板文件，或者把模板内容存储到数据库中的话，你就需要使用show方法来渲染输出了，show方法的调用格式：

show('渲染内容'[,'字符编码'][,'输出类型'])

例如，$this->show($content);

也可以指定编码和类型： $this->show($content, 'utf-8', 'text/xml');

show方法中的内容也可以支持模板解析。

四六 模板引擎

系统支持原生的PHP模板，而且本身内置了一个基于XML的高效的编译型模板引擎，系统默认使用的模板引擎是内置模板引擎，关于这个模板引擎的标签详细使用可以参考模版引擎部分。

内置的模板引擎也可以直接支持在模板文件中采用PHP原生代码和模板标签的混合使用，如果需要完全使用PHP本身作为模板引擎，可以配置： 'TMPL\_ENGINE\_TYPE' =>'PHP' 可以达到最佳的效率。

如果你使用了其他的模板引擎，只需要设置TMPL\_ENGINE\_TYPE参数为相关的模板引擎名称即可。

四七 模板变量输出

在模板中输出变量的方法很简单，例如，在控制器中我们给模板变量赋值：

|  |
| --- |
| $name = 'ThinkPHP';  $this->assign('name',$name);  $this->display(); |

然后就可以在模板中使用：

|  |
| --- |
| Hello,{$name}！ |

模板编译后的结果就是：

|  |
| --- |
| Hello,<?php echo($name);?>！ |

这样，运行的时候就会在模板中显示： Hello,ThinkPHP！

注意模板标签的{和$之间不能有任何的空格，否则标签无效。所以，下面的标签

|  |
| --- |
| Hello,{ $name}！ |

将不会正常输出name变量，而是直接保持不变输出： Hello,{ $name}！

普通标签默认开始标记是{，结束标记是 }。也可以通过设置TMPL\_L\_DELIM和TMPL\_R\_DELIM进行更改。例如，我们在项目配置文件中定义：

|  |
| --- |
| 'TMPL\_L\_DELIM'=>'<{',  'TMPL\_R\_DELIM'=>'}>', |

那么，上面的变量输出标签就应该改成：

|  |
| --- |
| Hello,<{$name}>！ |

模板标签的变量输出根据变量类型有所区别，刚才我们输出的是字符串变量，如果是数组变量，

|  |
| --- |
| $data['name'] = 'ThinkPHP';  $data['email'] = ['thinkphp@qq.com';](mailto:'thinkphp@qq.com';)  $this->assign('data',$data); |

那么，在模板中我们可以用下面的方式输出：

|  |
| --- |
| Name：{$data.name}  Email：{$data.email} |

或者用下面的方式也是有效：

|  |
| --- |
| Name：{$data['name']}  Email：{$data['email']}  当我们要输出多维数组的时候，往往要采用后面一种方式。 |

如果data变量是一个对象（并且包含有name和email两个属性），那么可以用下面的方式输出：

|  |
| --- |
| Name：{$data:name}  Email：{$data:email} |

或者

|  |
| --- |
| Name：{$data->name}  Email：{$data->email} |

四八 系统变量输出

1.系统变量输出

普通的模板变量输出需要先赋值在输出。系统变量不需要赋值直接在模板中输出。系统变量通常以$Think开头:

|  |
| --- |
| {$Think.server.script\_name} //输出$\_SERVER['SCRIPT\_NAME']变量  {$Think.session.user\_id} //输出$\_SESSION['user\_id']变量  {$Think.get.pageNumber} //输出$\_GET['pageNumber']变量  {$Think.cookie.name} //输出$\_COOKIE[‘name’]变量 |

支持输出$\_SERVER、$\_SESSION、$\_COOKIE、$\_GET、$\_POST、$\_REQUEST、$\_ENV

2.常量输出

可以常量输出，以$Think.const开头:

|  |
| --- |
| {$Think.const.MODULE\_NAME}; |

或者

|  |
| --- |
| {$Think.MODULE\_NAME}; |

3.配置输出

可以配置输出，以$Think.config开头:

|  |
| --- |
| {$Think.config.db\_charset};  {$Think.config.url\_model}; |

4.语言变量

可以语言输出，以$Think.lang开头:

|  |
| --- |
| {$Think.lang.page\_error}  {$Think.lang.var\_error} |

四九 模板中使用函数

对模板变量使用函数，如下：

|  |
| --- |
| {$data.name|MD5}; |

编译后的结果是：

|  |
| --- |
| <?php echo (md5($data['name'])); ?> |

如果函数有多个参数需要调用，则使用：

|  |
| --- |
| {$create\_time|date="y-m-d",###} |

表示date函数传入两个参数，每个参数用逗号分割，这里第一个参数是y-m-d，第二个参数是前面要输出的create\_time变量，因为该变量是第二个参数，因此需要用###标识变量位置，编译后的结果是：

|  |
| --- |
| <?php echo (date("y-m-d",$create\_time)); ?> |

如果前面输出的变量在后面定义的函数的第一个参数，则可以直接使用：

|  |
| --- |
| {$data.name|substr=0,3} |

表示输出

|  |
| --- |
| <?php echo (substr($data['name'],0,3)); ?> |

虽然也可以使用：

|  |
| --- |
| {$data.name|substr=###,0,3} |

还可以支持多个函数过滤，多个函数之间用“|”分割即可，例如

|  |
| --- |
| {$name|md5|strtoupper|substr=0,3} |

编译后的结果是：

|  |
| --- |
| <?php echo (substr(strtoupper(md5($name)),0,3)); ?> |

函数会按照从左到右的顺序依次调用。

如果你觉得这样写起来比较麻烦，也可以直接这样写：

|  |
| --- |
| {:substr(strtoupper(md5($name)),0,3)} |

变量输出使用的函数可以支持内置的PHP函数或者用户自定义函数，甚至是静态方法。

五十 默认值输出

我们可以给变量输出提供默认值，例如：

|  |
| --- |
| {$user.nickname|default="这家伙很懒，什么也没留下"} |

对系统变量依然可以支持默认值输出，例如：

|  |
| --- |
| {$Think.get.name|default="名称为空"} |

默认值和函数可以同时使用，例如：

|  |
| --- |
| {$Think.get.name|getName|default="名称为空"} |

五一 使用运算符

我们可以对模板输出使用运算符，包括对“+”“-” “\*” “/”和“%”的支持。

例如：

|  |  |
| --- | --- |
| 运算符 | 使用实例 |
| + | {$a+$b} |
| - | {$a-$b} |
| \* | {$a\*$b} |
| / | {$a/$n} |
| % | {$a%$b} |
| ++ | {$a++}或者{++$a} |
| -- | {$a--}或者{--$a} |

在使用运算符的时候不支持点语法和常规的函数用法，例如：

|  |
| --- |
| {$user.score+10} //错误的  {$user['score']+10} //正确的  {$user['score']\*$user['level']} //正确的  {$user['score']|myFun\*10} //错误的  {$user['score']+myFun($user['level'])} //正确的 |

五二 标签库

内置的模板引擎除了支持普通变量的输出之外，更强大的地方在于标签库功能。

标签库类似于Java的Struts中的JSP标签库，每一个标签库是一个独立的标签库文件，标签库中的每一个标签完成某个功能，采用XML标签方式（包括开放标签和闭合标签）。

标签库分为内置和扩展标签库，内置标签库是Cx标签库。

1. 导入标签库

使用taglib标签导入当前模板中需要使用的标签库，例如：

|  |
| --- |
| <taglib name="html" />  如果没有定义html标签库的话，则导入无效。 |

也可以导入多个标签库，使用：

|  |
| --- |
| <taglib name="html,article" /> |

导入标签库后，就可以使用标签库中定义的标签了，假设article标签库中定义了read标签：

|  |
| --- |
| <article:read name="hello" id="data" >  {$data.id}:{$data.title}  </article:read> |

在上面的标签中，<article:read>... </article:read> 就是闭合标签，起始和结束标签必须成对出现。

如果是 <article:read name="hello" /> 就是开放标签。

闭合和开放标签取决于标签库中的定义，一旦定义后就不能混淆使用，否则就会出现错误。

2. 内置标签库

内置标签库无需导入即可使用，并且不需要加XML中的标签库前缀，ThinkPHP内置的标签库是Cx标签库，所以，Cx标签库中的所有标签，我们可以在模板文件中直接使用，我们可以这样使用：

|  |
| --- |
| <eq name="status" value="1 >  正常  </eq> |

如果Cx不是内置标签的话，可能就需要这么使用了：

|  |
| --- |
| <cx:eq name="status" value="1 >  正常  </cx:eq> |

内置标签库可以简化模板中标签的使用，所以，我们还可以把其他的标签库定义为内置标签库（前提是多个标签库没有标签冲突的情况），例如：

|  |
| --- |
| 'TAGLIB\_BUILD\_IN' => 'cx,article' |

配置后，上面的标签用法就可以改为：

|  |
| --- |
| <read name="hello" id="data" >  {$data.id}:{$data.title}  </read> |

3. 标签库预加载

标签库预加载是指无需手动在模板文件中导入标签库即可使用标签库中的标签，通常用于某个标签库需要被大多数模板使用的情况。

在应用或者模块的配置文件中添加：

|  |
| --- |
| 'TAGLIB\_PRE\_LOAD' => 'article,html' |

设置后，模板文件就不再需要使用

|  |
| --- |
| <taglib name="html,article" /> |

但是仍然可以在模板中调用：

|  |
| --- |
| <article:read name="hello" id="data" >  {$data.id}:{$data.title}  </article:read> |

五三 模板继承

模板继承是一项更加灵活的模板布局方式，模板继承不同于模板布局，甚至来说，应该在模板布局的上层。模板继承其实并不难理解，就好比类的继承一样，模板也可以定义一个基础模板（或者是布局），并且其中定义相关的区块（block），然后继承（extend）该基础模板的子模板中就可以对基础模板中定义的区块进行重载。

因此，模板继承的优势其实是设计基础模板中的区块（block）和子模板中替换这些区块。

每个区块由<block></block>标签组成。下面就是基础模板中的一个典型的区块设计（用于设计网站标题）：

|  |
| --- |
| <block name="title"><title>网站标题</title></block> |

block标签必须指定name属性来标识当前区块的名称，这个标识在当前模板中应该是唯一的，block标签中可以包含任何模板内容，包括其他标签和变量，例如：

|  |
| --- |
| <block name="title"><title>{$web\_title}</title></block> |

你甚至还可以在区块中加载外部文件：

|  |
| --- |
| <block name="include"><include file="Public:header" /></block> |

一个模板中可以定义任意多个名称标识不重复的区块，例如下面定义了一个base.html基础模板：

|  |
| --- |
| <html>  <head>  <meta http-equiv = “Content-Type” content=”text/html,charset=utf-8”>  <block name="title"><title>标题</title></block>  </head>  <body>  <block name="menu">菜单</block>  <block name="left">左边分栏</block>  <block name="main">主内容</block>  <block name="right">右边分栏</block>  <block name="footer">底部</block>  </body>  </html> |

然后我们在子模板（其实是当前操作的入口模板）中使用继承：

|  |
| --- |
| <extend name="base" />  <block name="title"><title>{$title}</title></block>  <block name="menu">  <a href="/" >首页</a>  <a href="/info/" >资讯</a>  <a href="/bbs/" >论坛</a>  </block><block name="left">  </block>  <block name="content">  <volist name="list" id="vo">  <a href="/new/{$vo.id}">{$vo.title}</a>  <br/>  {$vo.content}  </volist>  </block>  <block name="right"> 最新资讯：  <volist name="news" id="new">  <a href="/new/{$new.id}">{$new.title}</a><br/>  </volist>  </block>  <block name="footer">  @ThinkPHP2012 版权所有  </block> |

可以看到，子模板中使用了extend标签定义需要继承的模板，extend标签的用法和include标签一样，你也可以加载其他模板：

|  |
| --- |
| <extend name="Public:base" /> |

或者使用绝对文件路径加载

|  |
| --- |
| <extend name="./Template/Public/base.html" /> |

在当前子模板中，只能定义区块而不能定义其他的模板内容，否则将会直接忽略，并且只能定义基础模板中已经定义的区块。

例如，如果采用下面的定义：

|  |
| --- |
| <block name="title"><title>{$title}</title></block>  <a href="/" >首页</a>  <a href="/info/" >资讯</a>  <a href="/bbs/" >论坛</a> |

导航部分将是无效的，不会显示在模板中。

在子模板中，可以对基础模板中的区块进行重载定义，如果没有重新定义的话，则表示沿用基础模板中的区块定义，如果定义了一个空的区块，则表示删除基础模板中的该区块内容。上面的例子，我们就把left区块的内容删除了，其他的区块都进行了重载。

子模板中的区块定义顺序是随意的，模板继承的用法关键在于基础模板如何布局和设计规划了，如果结合原来的布局功能，则会更加灵活。

五四 模板三元运算符

模板可以支持三元运算符，例如：

|  |
| --- |
| {$status?'正常':'错误'}  {$info['status']?$info['msg']:$info['error']} |

注意：三元运算符中暂时不支持点语法。

五五 包含文件

在当前模板中包含其他模板文件使用include标签。标签用法:

|  |
| --- |
| <include file=”模板文件1或者模板表达式,模板文件2或者模板表达式,...”/> |

1. 模板表达式

模板表达式的定义规则为：模块@主题/模块/控制器/操作

例如：

|  |
| --- |
| <include file=”Public/header” /> //包含头部模板header  <include file=”Public/footer” /> //包含尾部模板 footer  <include file=”Public/menu” /> //包含菜单模板 menu  <include file=”Blue/Public/menu”> //包含主题菜单模板 menu |

为了兼容3.1的写法，也可以支持：

|  |
| --- |
| <include file=”Public:header” /> //包含头部模板header  <include file=”Public:footer” /> //包含尾部模板 footer  <include file=”Public:menu” /> //包含菜单模板 menu  <include file=”Blue:Public:menu”> //包含主题菜单模板 menu |

可以一次包含多个模版，例如：

|  |
| --- |
| <include file="Public/header,Public/menu" /> |

注意，包含模版文件并不会自动调用控制器的方法，也就是说包含的其他模版文件中的变量赋值需要在当前操作中完成。

2. 使用模板文件

可以直接包含一个模版文件名（包含完整路径），例如：

|  |
| --- |
| <include file="./Application/Home/View/default/Public/header.html" /> |

3. 传入参数

无论你使用什么方式包含外部模板，Include标签支持在包含文件的同时传入参数，例如，下面的例子我们在包含header模板的时候传入了title和keywords变量：

|  |
| --- |
| <include file="Public/header" title="ThinkPHP框架" keywords="开源WEB开发框架" /> |

就可以在包含的header.html文件里面使用title和keywords变量，如下：

|  |
| --- |
| <html xmlns="http://www.w3.org/1999/xhtml">  <head>  <title>[title]</title>  <meta name="keywords" content="[keywords]" />  </head> |

注意：由于模板解析的特点，从入口模板开始解析，如果外部模板有所更改，模板引擎并不会重新编译模板，除非在调试模式下或者缓存已经过期。如果部署模式下修改了包含的外部模板文件后，需要把模块的缓存目录清空，否则无法生效。

五六 内置标签

1.volist标签

volist通常用于查询数据集(select方法)的结果输出。通常模型的select方法返回的结果是一个二维数组，可以直接使用volist标签进行输出。在控制中首先对模板进行赋值。

|  |
| --- |
| $User = D(‘User’);  $list = $User->where(‘id>10’)->limit(10)->select;  $this->assign(‘list’,$list);  $this->display(); |

在模板中定义如下，循环输出用户的编号和姓名：

|  |
| --- |
| <volist name=’list’ id=’vo’> //id类似于foreach中的key  编号:{$vo.id}&nbsp&nbsp姓名:{$vo.name}<br/>  </volist> |

Volist标签的name属性表示模板赋值的变量名称，因此不可随意在模板文件中改变。id表示当前的循环变量，可以随意指定，但确保不要和name属性冲突，例如：

|  |
| --- |
| <volist name="list" id="data">  {$data.id}:{$data.name}<br/>  </volist> |

支持输出查询结果中的部分数据，例如输出其中的第5～15条记录

|  |
| --- |
| <volist name="list" id="vo" offset="5" length='10'>  {$vo.name}  </volist> |

输出偶数记录

|  |
| --- |
| <volist name=’list’ id=’vo’ mod=’2’> //记录对2取余数  {$vo.name}  </volist> |

Mod属性还用于控制一定记录的换行，例如：

|  |
| --- |
| <volist name="list" id="vo" mod="5" > //记录对5取余数  {$vo.name}  <eq name="mod" value="4"><br/></eq> //eq标签，值等于4时输出  </volist> |

为空的时候输出提示：

|  |
| --- |
| <volist name=’list’ id=’vo’ empty=’请填写资料’>  {$vo.name}  </volist>  不支持HTML语法输入，支持变量输入,先在操作中变量赋值  $empty = ‘<span class=’red’>请填写资料</span>’  $this->assign(‘empty’,$empty);  $this->display();  然后在模板中引用变量  <volist name=’list’ id=’vo’ empty=”$empty”>  {$$vo.name}  </volist> |

输出循环变量

|  |
| --- |
| <volist name=’list’ id=’vo’ key=’k’>  {$k}:{$vo.name}  </volist>  如果未定义循环变量，默认为 i; |

如果要输出数组的索引，可以直接使用key变量，和循环变量不同的是，这个key是由数据本身决定，而不是循环控制的，例如：

|  |
| --- |
| <volist name="list" id="vo" >  {$key}.{$vo.name}  </volist> |

模板中可以直接使用函数设定数据集，而不需要在控制器中给模板变量赋值传入数据集变量，如：

|  |
| --- |
| <volist name=":fun('arg')" id="vo">  {$vo.name}  </volist> |

2.foreach标签

foreach标签类似与volist标签，只是更加简单，没有太多额外的属性，例如： {$vo.id}:{$vo.name}

name表示数据源 item表示循环变量。

可以输出索引，如下：

|  |
| --- |
| <foreach name="list" item="vo" >  {$key}|{$vo.id}:{$vo.name}  </foreach> |

也可以定义索引的变量名

|  |
| --- |
| <foreach name="list" item="vo" key="k" >  {$k}|{$vo.id}:{$vo.name}  </foreach> |

3.for标签

|  |
| --- |
| <for start="开始值" end="结束值" comparison="" step="步进值" name="循环变量名" >  </for> |

开始值、结束值、步进值和循环变量都可以支持变量，开始值和结束值是必须，其他是可选。comparison 的默认值是lt;；name的默认值是i，步进值的默认值是1，举例如下：

|  |
| --- |
| <for start="1" end="100">  {$i}  </for> |

解析后的代码是

|  |
| --- |
| for ($i=1;$i<100;$i+=1){  echo $i;  } |

4.switch标签

用法:

|  |
| --- |
| <switch name=’变量’>  <case value=’值1’ break=’0 or 1’>输出内容1</case>  <case value=’值2’>输出内容2</case>  <default/>默认情况  </switch> |

使用方法:

|  |
| --- |
| <switch name=’User.level’>  <case value=’1’>value1</case>  <case value=’2’>value2</case>  </switch> |

其中name属性可以使用函数以及系统变量，例如:

|  |
| --- |
| <switch name=’Think.get.userId|abs’>  <case value=’’1>admin</case>  </switch> |

对于case的value属性可以支持多个条件的判断，使用”|”进行分割，例如：

|  |
| --- |
| <switch name="Think.get.type">  <case value="gif|png|jpg">图像格式</case>  <default />其他格式  </switch> |

表示如果$\_GET["type"] 是gif、png或者jpg的话，就判断为图像格式。

Case标签还有一个break属性，表示是否需要break，默认是会自动添加break，如果不要break，可以使用：

|  |
| --- |
| <switch name="Think.get.userId|abs">  <case value="1" break="0">admin</case>  <case value="2">admin</case>  <default />default  </switch> |

也可以对case的value属性使用变量，例如：

|  |
| --- |
| <switch name="User.userId">  <case value="$adminId">admin</case>  <case value="$memberId">member</case>  <default />default  </switch> |

使用变量方式的情况下，不再支持多个条件的同时判断。

5.比较标签

比较标签用于简单的变量比较，复杂的判断条件可以用if标签替换，比较标签是一组标签的集合，基本上用法都一致，如下：

|  |
| --- |
| <比较标签 name=’变量名’ value=’条件值’>  内容  </比较标签> |

系统支持的比较标签以及所表示的含义分别是：

|  |  |
| --- | --- |
| 标签 | 含义 |
| eq | 等于 |
| neq | 不等于 |
| lt | 小于 |
| elt | 小于等于 |
| gt | 大于 |
| egt | 大于等于 |
| heq | 恒等于 |
| nheq | 不恒等于 |

他们的用法基本是一致的，区别在于判断的条件不同，并且所有的比较标签都可以和else标签一起使用。

|  |
| --- |
| <eq name="name" value="value">  相等  <else/>  不相等  </eq> |

5.范围判断

范围判断标签包括in notin between notbetween四个标签，都用于判断变量是否中某个范围。

5.1 IN和NOTIN

用法：假设我们中控制器中给id赋值为1：

|  |
| --- |
| $this->assign(‘id’,1); |

我们可以使用in标签来判断模板变量是否在某个范围内，例如：

|  |
| --- |
| <in name=’id’ value=’1,2,4’>  在范围内  </in>  最后会输出：在范围内 |

如果判断不在某个范围内，可以使用： id不在范围内 可以把上面两个标签合并成为：

|  |
| --- |
| <in name="id" value="1,2,3">  id在范围内  <else/>  id不在范围内  </in> |

name属性还可以支持直接判断系统变量，例如：

|  |
| --- |
| <in name="Think.get.id" value="1,2,3">  $\_GET['id'] 在范围内  </in> |

value属性也可以使用变量，例如：

|  |
| --- |
| <in name="id" value="$Think.post.ids">  id在范围内  </in> |

5.2BETWEEN和NOTBETWEEN

可以使用between标签来判断变量是否在某个区间范围内，可以使用：

|  |
| --- |
| <between name="id" value="1,10">  输出内容1  </between> |

同样，可以使用notbetween标签来判断变量不在某个范围内：

|  |
| --- |
| <notbetween name="id" value="1,10">  输出内容1  </notbetween> |

当使用between标签的时候，value只需要一个区间范围，也就是只支持两个值，后面的值无效，例如

|  |
| --- |
| <between name="id" value="1,3,10">  输出内容1  </between> |

实际判断的范围区间是1~3，而不是1~10，也可以支持字符串判断，例如：

|  |
| --- |
| <between name="id" value="A,Z">  输出内容1  </between> |

name属性可以直接使用系统变量，例如：

|  |
| --- |
| <between name="Think.post.id" value="1,5">  输出内容1  </between> |

value属性也可以使用变量，例如：

|  |
| --- |
| <between name="id" value="$range">  输出内容1  </between> |

变量的值可以是字符串或者数组，还可以支持系统变量。

|  |
| --- |
| <between name="id" value="$Think.get.range">  输出内容1  </between> |

5.3rang标签

也可以直接使用range标签，替换前面的判断用法：

|  |
| --- |
| <range name="id" value="1,2,3" type="in">  输出内容1  </range> |

其中type属性的值可以用in/notin/between/notbetween，其它属性的用法和IN或者BETWEEN一致。

6.IF标签

用法实例：

|  |
| --- |
| <if condition=”($name gt 10) or ($name lt 5)” >  value1  <else/>  value2  </if> |

在condition属性中可以支持eq等判断表达式，同上面的比较标签，但是不支持带有”>”、”<”等符号的用法，因为会混淆模板解析，所以下面的用法是错误的：

|  |
| --- |
| <if condition="$id < 5 ">  value1  <else />  value2  </if> |

必须改成：

|  |
| --- |
| <if condition="$id lt 5 ">  value1  <else />  value2  </if> |

除此之外，我们可以在condition属性里面使用php代码，例如：

|  |
| --- |
| <if condition="strtoupper($user['name']) neq 'THINKPHP'">  ThinkPHP  <else />  other Framework  </if> |

condition属性可以支持点语法和对象语法，例如：自动判断user变量是数组还是对象

|  |
| --- |
| <if condition="$user.name neq 'ThinkPHP'">  ThinkPHP  <else />  other Framework  </if> |

或者知道user变量是对象

|  |
| --- |
| <if condition="$user:name neq 'ThinkPHP'">  ThinkPHP  <else />  other Framework  </if> |

由于if标签的condition属性里面基本上使用的是php语法，尽可能使用判断标签和Switch标签会更加简洁，原则上来说，能够用switch和比较标签解决的尽量不用if标签完成。因为switch和比较标签可以使用变量调节器和系统变量。如果某些特殊的要求下面，IF标签仍然无法满足要求的话，可以使用原生php代码或者PHP标签来直接书写代码。

7.present标签

present标签用于判断某个变量是否已经定义，用法：

|  |
| --- |
| <present name="name">name  已经赋值  </present> |

如果判断没有赋值，可以使用：

|  |
| --- |
| <notpresent name="name">  name还没有赋值  </notpresent> |

可以把上面两个标签合并成为：

|  |
| --- |
| <present name="name">  name已经赋值  <else />  name还没有赋值  </present> |

present标签的name属性可以直接使用系统变量，例如：

|  |
| --- |
| <present name="Think.get.name">  $\_GET['name']已经赋值  </present> |

8.empty标签

empty标签用于判断某个变量是否为空，用法：

|  |
| --- |
| <empty name="name">  name为空值  </empty> |

如果判断没有赋值，可以使用：

|  |
| --- |
| <notempty name="name">  name不为空  </notempty> |

可以把上面两个标签合并成为：

|  |
| --- |
| <empty name="name">  name为空  <else />  name不为空  </empty> |

name属性可以直接使用系统变量，例如：

|  |
| --- |
| <empty name="Think.get.name">  $\_GET['name']为空值  </empty> |

9.defined标签

DEFINED标签用于判断某个常量是否有定义，用法如下：

|  |
| --- |
| <defined name="NAME">  NAME常量已经定义  </defined> |

name属性的值要注意严格大小写

如果判断没有被定义，可以使用：

|  |
| --- |
| <notdefined name="NAME">  NAME常量未定义  </notdefined> |

可以把上面两个标签合并成为：

|  |
| --- |
| <defined name="NAME">  NAME常量已经定义  <else />  NAME常量未定义  </defined> |

10. assign标签

ASSIGN标签用于在模板文件中赋值变量，用法如下：

|  |
| --- |
| <assign name="var" value="123" /> |

在运行模板的时候，赋值了一个var的变量，值是123。

name属性支持系统变量，例如：

|  |
| --- |
| <assign name="Think.get.id" value="123" /> |

表示在模板中给$\_GET['id'] 赋值了 123

value属性也支持变量，例如：

|  |
| --- |
| <assign name="var" value="$val" /> |

或者直接把系统变量赋值给var变量，例如：

|  |
| --- |
| <assign name="var" value="$Think.get.name" /> |

相当于，执行了：$var = $\_GET['name'];

10. define标签

DEFINE标签用于中模板中定义常量，用法如下：

|  |
| --- |
| <define name="MY\_DEFINE\_NAME" value="3" /> |

在运行模板的时候，就会定义一个MY\_DEFINE\_NAME的常量。

value属性可以支持变量（包括系统变量），例如：

|  |
| --- |
| <define name="MY\_DEFINE\_NAME" value="$Think.get.name" /> |

11. import标签

传统方式的导入外部JS和CSS文件的方法是直接在模板文件使用：

|  |
| --- |
| <script type='text/javascript' src='/Public/Js/Util/Array.js'>  <link rel="stylesheet" type="text/css" href="/App/Tpl/default/Public/css/style.css" /> |

系统提供了专门的标签来简化上面的导入：

第一个是import标签 ，导入方式采用类似ThinkPHP的import函数的命名空间方式，例如：

|  |
| --- |
| <import type='js' file="Js.Util.Array" /> |

Type属性默认是js， 所以下面的效果是相同的：

|  |
| --- |
| <import file="Js.Util.Array" /> |

还可以支持多个文件批量导入，例如：

|  |
| --- |
| <import file="Js.Util.Array,Js.Util.Date" /> |

导入外部CSS文件必须指定type属性的值，例如：

|  |
| --- |
| <import type='css' file="Css.common" /> |

上面的方式默认的import的起始路径是网站的Public目录，如果需要指定其他的目录，可以使用basepath属性，例如：

|  |
| --- |
| <import file="Js.Util.Array" basepath="./Common" /> |

第二个是load标签，通过文件方式导入当前项目的公共JS或者CSS

|  |
| --- |
| <load href="/Public/Js/Common.js" />  <load href="/Public/Css/common.css" /> |

在href属性中可以使用特殊模板标签替换，例如：

|  |
| --- |
| <load href="\_\_PUBLIC\_\_/Js/Common.js" /> |

Load标签可以无需指定type属性，系统会自动根据后缀自动判断。

系统还提供了两个标签别名js和css 用法和load一致，例如：

|  |
| --- |
| <js href="/Public/Js/Common.js" />  <css href="/Public/Css/common.css" /> |

五七 原样输出

可以使用literal标签来防止模板标签被解析，例如：

|  |
| --- |
| <literal>  <if condition="$name eq 1 ">  value1  <elseif condition="$name eq 2"/>  value2  <else />  value3  </if>  </literal> |

上面的if标签被literal标签包含，因此if标签里面的内容并不会被模板引擎解析，而是保持原样输出。

如果你的php标签中需要输出类似{$user} 或者 XML标签的情况，可以通过添加literal标签解决混淆问题，例如：

|  |
| --- |
| <php>echo '{$Think.config.CUSTOM.'.$key.'}';</php> |

这个php标签中的{$Think 可能会被模板引擎误当做标签解析，解决的办法就是加上literal，例如：

|  |
| --- |
| <php><literal>echo '{$Think.config.CUSTOM.'.$key.'}';</literal></php> |

Literal标签还可以用于页面的JS代码外层，确保JS代码中的某些用法和模板引擎不产生混淆。

总之，所有可能和内置模板引擎的解析规则冲突的地方都可以使用literal标签处理。

五八 模板注释

模板支持注释功能，该注释文字在最终页面不会显示，仅供模板制作人员参考和识别。

1.单行注释

格式：

|  |
| --- |
| {/\* 注释内容 \*/ } 或 {// 注释内容 }  例如：  {// 这是模板注释内容 } 注意{和注释标记之间不能有空格。 |

2.多行注释

支持多行注释，例如：

|  |
| --- |
| {/\* 这是模板  注释内容\*/ } |

模板注释支持多行，模板注释在生成编译缓存文件后会自动删除，这一点和Html的注释不同。

五八 模板布局

ThinkPHP的模板引擎内置了布局模板功能支持，可以方便的实现模板布局以及布局嵌套功能。包含三种布局方式：

1.全局配置模式

这种方式仅需在项目配置文件中添加相关的布局模板配置，就可以简单实现模板布局功能，比较适用于全站使用相同布局的情况，需要配置开启LAYOUT\_ON 参数（默认不开启），并且设置布局入口文件名LAYOUT\_NAME（默认为layout）。

|  |
| --- |
| ‘LAYOUT\_ON’=>true,  ‘LAYOUT\_NAME’=>’layout’, |

开启LAYOUT\_ON后，我们的模板渲染流程就有所变化，例如：

|  |
| --- |
| namespace Home\Controller  use Think\Controller  class UserController extends Controller{  Public function add(){  $this->display(‘add’);  }  } |

在不开启LAYOUT\_ON布局模板之前，会直接渲染 Application/Home/View/User/add.html 模板文件,开启之后，首先会渲染Application/Home/View/layout.html 模板，布局模板的写法和其他模板的写法类似，本身也可以支持所有的模板标签以及包含文件，区别在于有一个特定的输出替换变量{\_\_CONTENT\_\_}，例如，下面是一个典型的layout.html模板的写法：

|  |
| --- |
| <include file="Public:header" />  {\_\_CONTENT\_\_}  <include file="Public:footer" /> |

读取layout模板之后，会再解析User/add.html 模板文件，并把解析后的内容替换到layout布局模板文件的{CONTENT} 特定字符串。

当然可以通过设置来改变这个特定的替换字符串，例如：

|  |
| --- |
| 'TMPL\_LAYOUT\_ITEM' => '{\_\_REPLACE\_\_}' |

采用这种布局方式的情况下，一旦User/add.html 模板文件或者layout.html布局模板文件发生修改，都会导致模板重新编译。

如果需要指定其他位置的布局模板，可以使用：

|  |
| --- |
| 'LAYOUT\_NAME'=>'Layout/layoutname', |

就表示采用Application/Home/View/Layout/layoutname.html作为布局模板。

如果某些页面不需要使用布局模板功能，可以在模板文件开头加上 {\_\_NOLAYOUT\_\_} 字符串。

如果上面的User/add.html 模板文件里面包含有{\_\_NOLAYOUT\_\_}，则即使当前开启布局模板，也不会进行布局模板解析。

2.模板标签模式

这种布局模板不需要在配置文件中设置任何参数，也不需要开启LAYOUT\_ON，直接在模板文件中指定布局模板即可，相关的布局模板调整也在模板中进行。

以前面的输出模板为例，这种方式的入口还是在User/add.html 模板，但是我们可以修改下add模板文件的内容，在头部增加下面的布局标签（记得首先关闭前面的LAYOUT\_ON设置，否则可能出现布局循环）：

|  |
| --- |
| <layout name=’layout’> |

表示当前模板文件需要使用layout.html 布局模板文件，而布局模板文件的写法和上面第一种方式是一样的。当渲染User/add.html 模板文件的时候，如果读取到layout标签，则会把当前模板的解析内容替换到layout布局模板的{CONTENT} 特定字符串。

一个模板文件中只能使用一个布局模板，如果模板文件中没有使用任何layout标签则表示当前模板不使用任何布局。

如果需要使用其他的布局模板，可以改变layout的name属性，例如：

|  |
| --- |
| <layout name="newlayout" /> |

还可以在layout标签里面指定要替换的特定字符串：

|  |
| --- |
| <layout name="Layout/newlayout" replace="{\_\_REPLACE\_\_}" /> |

由于所有include标签引入的文件都支持layout标签，所以，我们可以借助layout标签和include标签相结合的方式实现布局模板的嵌套。例如，上面的例子

|  |
| --- |
| <include file="Public:header" />  <div id="main" class="main" >  {\_\_CONTENT\_\_}  </div>  <include file="Public:footer" /> |

在引入的header和footer模板文件中也可以添加layout标签，例如header模板文件的开头添加如下标签：

|  |
| --- |
| <layout name="menu" /> |

这样就实现了在头部模板中引用了menu布局模板。

也可以采用两种布局方式的结合，可以实现更加复杂的模板布局以及嵌套功能。

3.使用layout控制模板布局

使用内置的layout方法可以更灵活的在程序中控制模板输出的布局功能，尤其适用于局部需要布局或者关闭布局的情况，这种方式也不需要在配置文件中开启LAYOUT\_ON。例如：

|  |
| --- |
| namespace Home\Controller;  use Think\Controller;  class UserController extends Controller{  public function add(){  layout(true);  $this->display(‘add’);  }  } |

表示当前的模板输出启用了布局模板，并且采用默认的layout布局模板。

如果当前输出需要使用不同的布局模板，可以动态的指定布局模板名称，例如：

|  |
| --- |
| namespace Home\Controller;  use Think\Controller;  class UserController extends Controller{  public function add(){  layout(“Layout/newlayout”);  $this->display(‘add’);  }  } |

或者使用layout方法动态关闭当前模板的布局功能（这种用法可以配合第一种布局方式，例如全局配置已经开启了布局，可以在某个页面单独关闭）：

|  |
| --- |
| namespace Home\Controller;  use Think\Controller;  Class UserController extends Controller{  Public function add() {  layout(false); // 临时关闭当前模板的布局功能  $this->display('add');  }  } |

三种模板布局方式中，第一种和第三种是在程序中配置实现模板布局，第二种方式则是单纯通过模板标签在模板中使用布局。具体选择什么方式，需要根据项目的实际情况来了。

五九 调试模式

开启调试模式很简单，只需要在入口文件中增加一行常量定义代码：

|  |
| --- |
| <?php  // 开启调试模式  define('APP\_DEBUG', true);  // 定义应用目录  define('APP\_PATH', './Application/');  // 加载框架入口文件  require './ThinkPHP/ThinkPHP.php'; |

在完成开发阶段部署到生产环境后，只需要关闭调试模式或者删除调试模式定义代码即可切换到部署模式。

|  |
| --- |
| <?php  // 关闭调试模式  define('APP\_DEBUG', false);  // 定义应用目录  define('APP\_PATH', './Application/');  // 加载框架入口文件  require './ThinkPHP/ThinkPHP.php'; |

六十 异常处理

调试模式下面一旦系统发生严重错误会自动抛出异常，也可以用ThinkPHP内置的E方法手动抛出异常。

|  |
| --- |
| E(‘新增失败’); |

也可以支持异常代码（默认为0），例如：

|  |
| --- |
| E('信息录入错误',25); |

同样也可以使用throw 关键字来抛出异常，下面的写法是等效的：

|  |
| --- |
| throw new \Think\ThinkException('新增失败'); |

我们可以自定义异常页面的显示，系统内置的异常模板在系统目录的Tpl/think\_exception.tpl，可以通过修改系统模板来修改异常页面的显示。也通过设置TMPL\_EXCEPTION\_FILE配置参数来修改系统默认的异常模板文件， 例如：

|  |
| --- |
| 'TMPL\_EXCEPTION\_FILE' => APP\_PATH.'/Public/exception.tpl' |

异常模板中可以使用的异常变量有：

|  |
| --- |
| $e['file']异常文件名  $e['line']异常发生的文件行数  $e['message']异常信息  $e['trace']异常的详细Trace信息 |

因为异常模板使用的是原生PHP代码，所以还可以支持任何的PHP方法和系统变量使用。

抛出异常后通常会显示具体的错误信息，如果不想让用户看到具体的错误信息，可以设置关闭错误信息的显示并设置统一的错误提示信息，例如：

|  |
| --- |
| 'SHOW\_ERROR\_MSG' => false,  'ERROR\_MESSAGE' => '发生错误！' |

设置之后，所有的异常页面只会显示“发生错误！”这样的提示信息，但是日志文件中仍然可以查看具体的错误信息。

系统的默认情况下，调试模式是开启错误信息显示的，部署模式则关闭错误信息显示。

另外一种方式是配置ERROR\_PAGE参数，把所有异常和错误都指向一个统一页面，从而避免让用户看到异常信息，通常在部署模式下面使用。ERROR\_PAGE参数必须是一个完整的URL地址，例如：

|  |
| --- |
| 'ERROR\_PAGE' =>'/Public/error.html' |

如果不在当前域名，还可以指定域名：

|  |
| --- |
| 'ERROR\_PAGE' =>'http://www.myDomain.com/Public/error.html' |

注意ERROR\_PAGE所指向的页面不能再使用异常的模板变量了。

六一 日志记录

日志的处理工作是由系统自动进行的，在开启日志记录的情况下，会记录下允许的日志级别的所有日志信息。

其中，为了性能考虑，SQL日志级别必须在调试模式开启下有效，否则就不会记录。系统的日志记录由核心的Think\Log类及其驱动完成，提供了多种方式记录了不同的级别的日志信息。

默认情况下只是在调试模式记录日志，要在部署模式开启日志记录，必须在配置中开启LOG\_RECORD参数，以及可以在应用配置文件中配置需要记录的日志级别，例如：

|  |
| --- |
| 'LOG\_RECORD' => true, // 开启日志记录  'LOG\_LEVEL' =>'EMERG,ALERT,CRIT,ERR', // 只记录EMERG ALERT CRIT ERR 错误 |

日志的记录并非实时保存的，只有当当前请求完成或者异常结束后才会实际写入日志信息，否则只是记录在内存中。

日志级别

ThinkPHP对系统的日志按照级别来分类，包括：

● EMERG 严重错误，导致系统崩溃无法使用

● ALERT 警戒性错误， 必须被立即修改的错误

● CRIT 临界值错误， 超过临界值的错误

● ERR 一般性错误

● WARN 警告性错误， 需要发出警告的错误

● NOTICE 通知，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，该级别只在调试模式开启时有效

日志的记录方式默认是文件方式，可以通过驱动的方式来扩展支持更多的记录方式。

记录方式由LOG\_TYPE参数配置，例如：

|  |
| --- |
| 'LOG\_TYPE' => 'File', // 日志记录类型 默认为文件方式 |

File方式记录，对应的驱动文件位于系统的Library/Think/Log/Driver/File.class.php。

六二 页面trace

调试模式并不能完全满足我们调试的需要，有时候我们需要手动的输出一些调试信息。除了本身可以借助一些开发工具进行调试外，ThinkPHP还提供了一些内置的调试工具和函数。例如，页面Trace功能就是ThinkPHP提供给开发人员的一个用于开发调试的辅助工具。可以实时显示当前页面的操作的请求信息、运行情况、SQL执行、错误提示等，并支持自定义显示。

页面Trace功能对调试模式和部署模式都有效，不过只能用于有页面输出的情况（如果你的操作没有任何输出，那么可能页面Trace功能对你帮助不大，你可能需要使用后面的调试方法）。

要开启页面Trace功能，需要在项目配置文件中设置：

|  |
| --- |
| // 显示页面Trace信息  'SHOW\_PAGE\_TRACE' =>true, |

六三 trace方法

页面Trace只能用于有页面输出的情况，但是trace方法可以用在任何情况，而且trace方法可以用于AJAX等操作。

Trace方法的格式：

|  |
| --- |
| trace(‘变量’,’标签’,’级别’,’是否有记录日志’);  例如:  $info = '测试信息';  trace($info,'提示');  如果希望把变量调试输出到页面Trace的某个选项卡里面，可以使用：  trace($info,'提示','user'); |

表示输出到user选项卡，如果没有指定选项卡的话，默认会输出到debug选项卡。 trace方法也可以直接抛出异常，如果是输出到ERR选项卡，并且开启 'TRACE\_EXCEPTION'=>true

的话，

|  |
| --- |
| trace($info,'错误','ERR'); |

会抛出异常。有三种情况下，trace方法会记录日志：

1. AJAX请求

2. SHOW\_PAGE\_TRACE为false，也就是页面Trace关闭的情况下

3. trace方法的第四个参数为true

在这种情况下，trace方法的第三个参数就表示记录的日志级别，通常包括：

|  |
| --- |
| 'ERR' // 一般错误: 一般性错误  'WARN' // 警告性错误: 需要发出警告的错误  'NOTIC' // 通知: 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: 调试信息  'SQL' // SQL：SQL语句 |

凭借强大的页面Trace信息功能支持，ThinkPHP可以支持断点调试功能。我们只需要在不同的位置对某个变量进行trace输出即可，例如：

|  |
| --- |
| $blog = D("Blog");  $vo = $blog->create();  trace($vo,'create vo');  $vo = $blog->find();  trace($vo,'find vo'); |

六四 变量调试

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

用法：dump($var, $echo=true, $label=null, $strict=true)

|  |
| --- |
| $Blog = D("Blog");  $blog = $Blog->find(3);  dump($blog); |

六五 性能调试

开发过程中，有些时候为了测试性能，经常需要调试某段代码的运行时间或者内存占用开销，系统提供了G方法可以很方便的获取某个区间的运行时间和内存占用情况。 例如：

|  |
| --- |
| G('begin');  // ...其他代码段  G('end');  // ...也许这里还有其他代码  // 进行统计区间  echo G('begin','end').'s'; |

G('begin','end') 表示统计begin位置到end位置的执行时间（单位是秒），begin必须是一个已经标记过的位置，如果这个时候end位置还没被标记过，则会自动把当前位置标记为end标签，输出的结果类似于：0.0056s

默认的统计精度是小数点后4位，如果觉得这个统计精度不够，还可以设置例如：

|  |
| --- |
| G('begin','end',6).'s'; |

可能的输出会变成：0.005587s

如果你的环境支持内存占用统计的话，还可以使用G方法进行区间内存开销统计（单位为kb），例如：

|  |
| --- |
| echo G('begin','end','m').'kb'; |

第三个参数使用m表示进行内存开销统计，输出的结果可能是：625kb

同样，如果end标签没有被标记的话，会自动把当前位置先标记位end标签。

如果环境不支持内存统计，则该参数无效，仍然会进行区间运行时间统计。

六六 模型调试

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

|  |
| --- |
| $User = M("User"); // 实例化User对象  $User->find(1);  echo $User->getLastSql(); |

输出结果是 SELECT \* FROM think\_user WHERE id = 1

并且每个模型都使用独立的最后SQL记录，互不干扰，但是可以用空模型的getLastSql方法获取全局的最后SQL记录。

|  |
| --- |
| $User = M("User"); // 实例化User模型  $Info = M("Info"); // 实例化Info模型  $User->find(1);  $Info->find(2);  echo M()->getLastSql();  echo $User->getLastSql();  echo $Info->getLastSql(); |

输出结果是

|  |
| --- |
| SELECT \* FROM think\_info WHERE id = 2  SELECT \* FROM think\_user WHERE id = 1  SELECT \* FROM think\_info WHERE id = 2 |

getLastSql方法只能获取最后执行的sql记录，如果需要了解更多的SQL日志，可以通过查看当前的页面Trace或者日志文件。

在模型操作中，还可以获取数据库的错误信息，例如：

|  |
| --- |
| $User = M("User"); // 实例化User对象  $result = $User->find(1);  if(false === $result){  echo $User->getDbError();  } |

CURD操作如果返回值为false，表示数据库操作发生错误，这个时候就需要使用模型的getDbError方法来查看数据库返回的具体错误信息