ThinkPHP Framework 1.0 WhiteBook

ThinkPHP 1.0 特性概要

编写：ThinkPHP 文档组

最后更新：2008-06-26

目录

1 版权信息

对 ThinkPHP 有任何疑问或者建议，请进入官方论坛［ http://bbs.thinkphp.cn ］发布相关讨论。并在此感谢 ThinkPHP 团队的所有成员和所有关注和支持 ThinkPHP 的朋友。

有关 ThinkPHP 项目及本文档的最新资料，请及时访问 ThinkPHP 项目主站 http://thinkphp.cn 。

2 特性介绍

2.1 MVC 和执行过程

和很多应用框架一样，ThinkPHP 也采用了 MVC 模式和单一入口模式。

模型（M）：模型的定义由 Model 类来完成。Model 类位于项目目录下面的 LibModel 目录。

控制器（C）：应用控制器（核心控制器）和 Action 控制器都承担了控制器的角色，区别在于 Action 控制器完成业务过程，而应用控制器（App 类）负责调度控制。Action 控制器位于项目目录下面的 LibAction 目录。

视图（V）：模板的实现是和框架无关的，做到了 100%分离，可以独立预览和制作。模板目录位于项目目录下面的 Tpl 目录。

执行过程

ThinkPHP 采用模块和操作的方式来执行，首先，用户的请求会通过入口文件生成一个应用实例，应用控制器（我们称之为核心控制器）会管理整个用户执行的过程，并负责模块的调度和操作的执行，并且在最后销毁该应用实例。任何一个 WEB 行为都可以认为是一个模块的某个操作，而模块的概念在 ThinkPHP 中的体现就是 Action 类，也类似于我们平常提到控制器的概念，而具体的操作就是 Action 类中的某个公共的方法。

ThinkPHP 里面会根据当前的 URL 来分析要执行的模块和操作。这个分析工作由 URL 调度器来实现，官方内置了 Dispatcher 来完成该调度。在 Dispatcher 调度器中，会根据

```
http://servername/appName/moduleName/actionName/params
```

来获取当前需要执行的项目（appName）、模块（moduleName）和操作（actionName），在某些情况下，appName 可以不需要（通常是网站的首页，因为项目名称可以在入口文件中指定，这种情况下，appName

就会被入口文件替代）

每个模块名称是一个 Action 文件，类似于我们平常所说的控制器，系统会自动寻找项目类库 Action 目录下面的相关类，如果没有找到，会尝试搜索应用目录下面的组件类中包含的模块类，如果依然没有，则会判断是否存在空模块（EmpatyAction），否则抛出异常。

而 actionName 操作是首先判断是否存在 Action 类的公共方法，如果不存在则会继续寻找父类中的方法，如果依然不存在，就会判断是否存在空操作（_empty），然后就会寻找是否存在自动匹配的模版文件。如果存在模版文件，那么就直接渲染模版输出，否则将抛出异常。

因此应用开发中的一个重要过程就是给不同的模块定义具体的操作。一个应用如果不需要和数据库交互的时候可以不需要定义模型类，但是必须定义 Action 控制器。

Action 控制器的定义非常简单，只要继承 Action 基础类就可以了，例如：

```
Class UserAction extends Action{
}
```

你甚至不需要定义任何操作方法，就可以完成很多默认的操作，因为 Action 基础类已经为你定义了很多常用的操作方法。例如，我们可以执行（如果已经存在对应模板文件的情况）

```
http://servername/index.php/User/
http://servername/index.php/User/add
```

如果你需要增加或者重新定义自己的操作方法，增加一个方法就可以了，例如

```
Class UserAction extends Action{
  // 定义一个 select 操作方法，注意操作方法不需要任何参数
  Public function select(){
   // select 操作方法的逻辑实现
   // ……
   $this->display(); // 输出模板页面
}
}
```

我们就可以执行http://servername/index.php/User/select/了，系统会自动定位当前操作的模板文件

而默认的模板文件应该位于项目目录下面的 Tpl\default\User\select.html

2.2 项目自动生成

我们知道，使用框架开发项目的时候，都需要遵守框架的一些目录规范，所以我们经常需要在每次项目的开始来——创建这些目录结构，虽然 Tp 已经有很多第三方的目录生成工具，他们能够最大程度的减轻你手动创建的工作量。

但是如果你手边没有那些自动生成工具怎么办，其实 ThinkPHP 框架已经内置了项目目录的自动生成功能。

生成目录

新版具有项目目录自动创建功能，你只需要定义好项目的入口文件，第一次执行入口文件的时候，系统会自动创建项目的相关目录结构，如果是 linux 环境下面需要给项目入口文件里面指定的路径设置可写权限。

例如，我们在 WEB 的根目录下面创建一个 App1 目录，并且在下面定义了一个项目入口文件 index.php 如下：

```
// 定义 ThinkPHP 框架路径
define('THINK_PATH', '../ThinkPHP/');
//定义项目名称和路径
define('APP_NAME', 'App1');
define('APP_PATH', '.');
// 加载框架入口文件
require(THINK_PATH."/ThinkPHP.php");
//实例化一个网站应用实例
```

```
$App = new App();
//应用程序初始化
$App->run();
```

然后，我们什么都不用做，仅仅是在浏览器里面输入

```
http://localhost/App1/
```

系统在第一次执行的时候会自动判断项目的目录结构是否存在，如果不存在的话就会完成整个项目的目录结构的自动创建，并自动创建了一个 Hello,world 应用，你可以修改 Lib\Action\IndexAction.class.php 文件里面的 index 方法来开始你的第一个 ThinkPHP 应用。

安全文件

在 1.0.3 版本以上，如果在入口文件里面设置了 BUILD_DIR_SECURE 参数为 True，还会自动给项目目录生成目录安全文件（在相关的目录下面生成空白的 htm 文件），并且可以自定义安全文件的文件名 DIR_SECURE_FILENAME ，默认是 index.html，如果你想给你们的安全文件定义为 default.html 可以使用

```
define('DIR_SECURE_FILENAME', 'default.html');
```

还可以支持多个安全文件写入，例如你想同时写入 index.html 和 index.htm 两个文件，以满足不同的服务器部署环境，可以这样定义：

```
define('DIR_SECURE_FILENAME', 'index.html,index.htm');
```

默认的安全文件只是写入一个空白字符串，如果需要写入其他内容，可以通过 DIR_SECURE_CONTENT 参数来指定，例如：

```
define('DIR_SECURE_CONTENT', 'deney Access!');
```

下面是一个完整的使用目录安全写入的例子

```
define('BUILD_DIR_SECURE',true);
define('DIR_SECURE_FILENAME', 'default.html');
define('DIR_SECURE_CONTENT', 'deney Access!');
```

注意事项

如果你使用的是 1.0.3 版本以上，如果定义的项目目录不存在会自己进行创建，如果连项目名称没有

定义的话，系统会自动生成一个系统路径 MD5 后的字符串作为项目名称，并且会以这个名称自动创建目录。

另外系统是根据核心编译缓存文件来判断是否需要检测目录结构的，如果你删除了核心缓存文件的话，这个过程会重新检查，但是不会覆盖已有的文件和目录。

2.3 项目编译机制

ThinkPHP 正式版本开始引入了新的项目编译机制，而在 1.0.3 版本中又进行了一些优化设置。

所谓的项目编译机制是指系统第一次运行的时候会自动生成核心缓存文件~runtime.php 和项目编译缓存文件~app.php，第二次就会直接载入编译过的缓存文件，从而省去很多 IO 开销，加快执行速度，并且编译缓存文件默认是去掉空白和注释的，因此存在一个预编译的过程。项目编译机制对运行没有任何影响 预编译操作和其他的目录检测机制只会执行一次 因此无论在预编译过程中做了多少复杂的操作，对后面的执行没有任何效率的缺失。

1.0.2 版本之前默认的生成路径是项目的根目录，1.0.2 版本开始，默认的编译缓存目录为 TEMP_PATH (项目的临时文件目录)。

还可以在入口文件里面设置 RUNTIME_PATH，例如：

```
defined('RUNTIME_PATH','./MyApp/runtime/');
```

系统会自动把编译缓存文件放到该目录下面生成，注意这样的话在 Linux 环境下面需要对 RUNTIME_PATH 目录设置可写权限。

注意在调试模式下面不会生成项目编译缓存，但是依然会生成核心缓存。

如果不希望生成核心缓存文件的话，可以在项目入口文件里面设置 CACHE_RUNTIME

例如：

```
define('CACHE_RUNTIME',false);
```

如果在非调试模式下面，修改了配置文件或者项目公共文件，需要删除项目编译文件。

1.0.4 版本开始，修改配置文件无需删除编译缓存文件。

1.0.3 版本以上，还可以设置对编译缓存的内容是否进行去空白和注释，例如：

```
define('STRIP_RUNTIME_SPACE',false);
```

则生成的编译缓存文件是没有经过去注释和空白的，仅仅是把文件合并到一起，这样的好处是便于调试的错误定位，建议部署模式的时候把上面的设置为 True 或者删除该定义。

注意事项

下面的一些问题需要引起注意，便于在开发过程中更快的解决问题：

1、编译缓存目录必须设置为可写

2、如果有修改核心文件，可能需要删除核心编译缓存~runtime.php

3、如果在非调试模式下有修改配置文件或者项目的 common 文件，需要删除项目编译缓存文件~app.php

2.4 灵活简单的项目配置

ThinkPHP 提供了灵活的配置功能，采用最有效率的 PHP 返回数组方式定义，支持惯例配置、项目配置、调试配置和模块配置，并且会自动生成配置缓存文件，无需重复解析的开销。对于有些简单的应用，你无需配置任何配置文件，而对于复杂的要求，你还可以增加模块配置文件，另外 ThinkPHP 的动态配置使得你在开发过程中可以灵活的调整配置参数。

如何配置

ThinkPHP 在项目配置上面创造了自己独有的分层配置模式，其配置层次体现在：

惯例配置--> 项目配置--> 模块配置--> 操作（动态）配置

// 优先顺序从右到左（在没有生效的前提下）

1、配置格式

ThinkPHP 框架对配置文件的定义格式采用返回数据的方式

格式为：

```
<?php return array( 'DEBUG_MODE' => true,);?>
```

配置参数不区分大小写～无论大小写定义都会转换成小写。原则上，每个项目配置文件除了定义 ThinkPHP 所需要的配置参数之外，开发人员可以在里面添加项目需要的一些配置参数，系统必须的配置参数请查看后面的配置参数列表。

2、惯例配置

Rail 的重要思想就是惯例重于配置，新版的 ThinkPHP 吸收了这一特性，引入了惯例配置的支持。系统内置有一个惯例配置文件（位于 Think\Common\convention.php），按照大多数的使用对常用参数进行了默认配置。所以，对于应用项目的配置文件，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。

3、模块配置

新版的 ThinkPHP 支持对某些参数进行动态配置，针对这一特性，ThinkPHP 还特别引入了模块配置文件的支持，这其实也是动态配置的体现。模块配置文件的命名规则是

模块名称+_config.php

4、操作（动态）配置

而在具体的Action方法里面，我们仍然可以对某些参数进行动态配置，主要是指那些还没有使用的参数。

```
//获取已经设置的参数值：
C('参数名称')
//设置新的值：
C('参数名称','新的参数值');
```

模板支持

ThinkPHP 的配置参数的存取采用静态变量，以保证存取的速度。而在视图层还可以通过模板标签来直接显示配置参数的指，而无需进行模板赋值。

读取配置参数的标签用法如下：

```
{$Think.config.参数名称}
```

2.5 ActiveRecord 模式

ThinkPHP 实现了 ActiveRecords 模式的 ORM 模型，采用了非标准的 ORM 模型：表映射到类，记录（集）映射到对象，字段属性映射到对象的虚拟属性。最大的特点就是使用方便，从而达到敏捷开发的目的。

开发过程中，只需要定义好模型类就可以进行方便的数据操作了，例如我们定义了一个 UserModel 类：

```
class UserModel extends Model{
}
```

甚至无需增加任何属性和方法，我们就可以进行下面的操作了。

```
$User = D("User"); // 实例化 User 对象
// 或者 $User = new UserModel();
```

```
$User->find(1); // 查找 id 为 1 的记录

$User->name = 'ThinkPHP'; // 把查找到的记录的名称字段修改为 ThinkPHP

$User->save(); // 保存修改的数据
```

比 ActiveRecord 模式更加高级的是，ThinkPHP 可以把记录集映射到对象，例如

```
$User->findAll();
foreach ($User as $user){

        echo $user->name;// 可以结合配置来使用 $user['name']

}
```

注意事项

在 ThinkPHP 里面 AR 模式和普通模式结合起来使用，根据实际的项目需要。因此有时候，也不要拘泥于 AR 模式的操作~

2.6 富模型支持

ThinkPHP 中的模型是一个丰富的模型类（Rich Model），除了包含有 ActiveRecord 模式、关联操作的实现和领域模型外，ThinkPHP 还支持聚合模型、静态模型、视图模型，给模型赋予了更多的内涵。关于 ActiveRecord 和普通模型的介绍请参考之前的资料。

聚合模型

聚合模型是一种虚拟模型，用于把数据表的字段抽象话，更加容易理解和操作。

我们可以把数据表中的某些属性进行数据封装，这样就把枯燥的数据表字段赋予更好的可读性。例如，在数据库有一个 User 表，字段定义如下：

```
CREATE TABLE `topthink_user` (
`id` int(11) unsigned NOT NULL auto_increment,
`name` varchar(30) NOT NULL default '',
`nickname` varchar(50) NOT NULL default '',
`password` varchar(32) NOT NULL default '',
`email` varchar(255) NOT NULL default '',
```

```
`url` varchar(255) NOT NULL default '',
`status` tinyint(1) unsigned NOT NULL default '0',
`remark` varchar(255) NOT NULL,
`city` varchar(50) NOT NULL,
`qq` varchar(15) NOT NULL,
PRIMARY KEY (`id`)
)
```

我们把 User 表中的有关用户信息的字段封装成一个 Info 对象（或者数组），例如，我们希望 Info 对象包括 nickname、email、url、city、qq 这些字段属性 ,这样 ,看起来的 User 属性变成 id、name、password、Info（聚合对象属性，包括 nickname、email、url、city、qq ）、status

所以在进行数据操作的时候，我们只需要把 Info 对象赋值传递到 User 对象就可以完成 Info 对象所封装的属性的数据写入。这个 Info 对象就称为聚合对象，或者组合对象。在 ThinkPHP 里面，这个聚合对象并不一定要定义 Model，因为可以动态的创建这个聚合对象，例如：

```
$User = D("User");
$User->Info=array('nickname'=>'ThinkPHP','email'=>'ThinkPHP@gmail.com';
$User->add();
```

这里就使用了 Info 聚合对象来完成数据写入操作，在写入数据库之前，ThinkPHP 会自动把聚合对象的属性值转换成 User 对象的属性来完成数据写入。当然，要使用聚合对象，我们还要对 UserModel 进行一些定义，确保系统可以识别该属性是聚合对象属性还是普通属性。

我们需要在 UserModel 类里面增加聚合对象的定义

```
$aggregation = array('Info');
```

我们可以同时定义多个聚合对象，例如：

```
$aggregation = array('Info','Log');
```

默认情况下，对聚合对象的属性是不受限制，而是由应用代码把握，为了避免无效的字段写入数据库而导致错误，我们可以限制属性，通过使用下面的方式定义：

```
$aggregation = array(array('Info','nickname,email'));
```

这样一来，聚合对象的属性就只能包括 nickname 和 email 属性，其它属性的值将无效，如：

```
$User = D("User");
$User->Info=array('nickname'=>'ThinkPHP','msn'=>'ThinkPHP@gmail.com');
$User->add();
```

上面的操作，msn 属性就不会写入数据库。

有些情况下，我们可能需要把数据表的字段映射成我们想要的属性名称，例如，我想把 url 字段映射成 web 属性定义，但是不想修改数据表的定义，我们可以进行下面的定义

```
$aggregation = array(array('Info',array('nickname','email','web'=>'url')));
```

于是，下面的操作

```
$User->Info=array('nickname'=>'ThinkPHP','web'=>'http://thinkphp.cn');
$User->add();
```

系统会自动把 web 属性转换成 url 字段名称写入数据库。

我们还可以单独定义一个 Info 模型对象，来更好的配合 User 对象完成一些复杂操作

在定义 Info 对象的时候，我们要注意，这个对象是一个虚拟对象，并没有对应数据库的任何数据表，所以有些特殊的属性要定义，例如：

```
Class InfoModel extends Model{
function _initialize(){
// 表示该对象是复合对象
$this->composite = true;
}
}
```

定义了 composite 为 true 后，系统就不会对 Info 对象进行数据库初始化操作，然后我们就可以进行下面的操作：

```
$User = D("User");
$Info = D("Info");
$Info->nickname = "ThinkPHP";
$Info->email = 'ThinkPHP@gmail.com';
$User->Info=$Info;
$User->add();
```

静态模型

静态模型可以让模型的数据静态化，而不需要频繁和数据库打交道，可以用于不经常不变的数据表。而且 ThinkPHP 在 ORM 层上面模拟实现了数据库的视图功能，这是 ThinkPHP 独创的一项技术，使得多表

的关联查询具有更高的效率和可操作性，而无需定义复杂的关联关系，让关联关系更加浅显易懂。

有些时候，数据表的数据一旦添加后就不再容易变化，我们更希望把这样的模型数据静态化，而不需要再次访问数据库。ThinkPHP 支持静态模型的概念，一旦把模型设置为静态，那么会在第一次初始化的时候获取数据表的全部数据，并生成缓存，以后不会再连接数据库。而只需要直接访问模型的 dataList 数据即可。

例如，我们把分类 CategoryModel 设置为静态模型

```
class CategoryModel extends Model
$staticModel = true;
}
```

一旦初始化完成后，我们就可以用 S('Category')来获取 Category 数据表的数据了，并且如果我们以后再次访问 CategoryModel 对象的话，也可以直接获取到缓存的数据列表，但是这个时候已经不能再进行数据更改和其它数据库操作了。如果需要更新数据，请首先把 staticModel 设置为 false 再进行操作。

视图模型

ThinkPHP 在 ORM 模型里面模拟实现了数据库的视图模型，该功能可以用于多表查询。

要定义视图对象，需要设置 viewModel 为 true，然后设置 viewFields 属性即可，例如下面的例子，我们设置了一个 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）。

```
class BlogViewModel extends Model
{
        var $viewModel = true;
        var $viewFields = array(
                'Category'=>array('title'=>'categoryName'),
                'User'=>array('name'=>'userName'),
                'Blog'=>array('id','name','title'),
                );
}
```

接下来，我们就可以对视图对象进行操作了

```
$Model = D("BlogView");
$Model->findAll();
```

因为视图查询返回的结果有时候存在重复数据，我们可以通过定义 viewConditon 属性来设置查询的基础条件，例如

```
$viewCondition = array(
'Blog.categoryId'=>array('egf','Category.id'),
'Blog.userId'=>array('egf','User.id'),
);
```

注意 egf 指的是后面的条件不是字符串，而是 SQL 字段操作，这样定义后，在进行其它查询的时候，会自动带上这个基础条件。视图模型的查询操作尽可能使用 Map 对象或者数组方式，否则就需要手动添加基础条件。例如

```
$Model->findAll('Blog.categoryId=Category.id AND Blog.userId=User.id AN
D Blog.id in (1,3,6)');
```

视图模型的定义一般需要先单独定义其中的模型类，但是这并不是必须的，如果没有定义其中的模型类，系统会默认按照系统的规则进行数据表的定位。

目前的视图模型仅仅针对查询操作

2.7 路由和 SEO 支持

我们来通过一个简单的例子，分别用 URL 路由、空操作和空模块三个角度来实现，由此来看 ThinkPHP 对 SEO 和 URL 友好的支持到底有多方便。

因为 URL_REWRITE 比较特殊，需要服务器支持，所以下面的例子不涉及 URL_REWRITE，所以比较具有通用性。

例子说明如下：

如果我们有一个 City 模块，而我们希望能够通过类似下面这样的 URL 地址来访问具体某个城市的操作：

http://<serverName>/index.php/City/shanghai/

一、使用 URL 路由功能实现

由于 shanghai 这个操作方法是不存在的，我们给相关的城市操作定义了一个 city 方法，如下：

```
Class CityAction extends Action{
public function city(){
// 读取城市名称
$cityName = $_GET['name'];
Echo ('当前城市:'.$cityName);
}
}
```

接下来我们来定义路由文件，实现类似于

http://<serverName>/index.php/City/shanghai/

这样的解析，路由文件名称是 routes.php 放在项目的 Conf 目录下面，内容如下：

```
Return array(
'City'=>array('City','city','name');
);
```

这样，URL 里面所有的 City 模块（其实已经不是 City 模块了，而是一个路由定义）都会被路由到 City

模块的 city 操作，而后面的第二个参数会被解析成 $_GET['name']

接下来，我们就可以在浏览器里面输入

http://<serverName>/index.php/City/beijing/
http://<serverName>/index.php/City/shanghai/
http://<serverName>/index.php/City/shenzhen/
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、空操作实现

同样是上面的例子，我们用空操作的思路来实现。

我们定义了一个 CityAction 类，代表了 City 模块，而我们希望能够通过类似下面这样的 URL 地址来访问具体某个城市的操作：

http://<serverName>/index.php/City/shanghai/

由于第二个参数表示的含义是 City 模块的操作名称，我们不可能给每个城市都定义一个操作方法，那样的工作量将是相当巨大的(尤其是在中国 ,呵呵~),当然 ,我们可以利于 URL 路由功能来解决这个问题。

但是我们还可以用空操作来更加高效地解决这个问题，我们来看下具体如何实现。

我们只需要给 CityAction 类定义一个_emtpy （空操作）方法：

```
Class CityAction extends Action{
Public function _empty(){
// 把所有城市的操作都解析到 city 方法
// 注意 city 方法本身是 protected 方法
$cityName = ACTION_NAME;
$this->city($cityName);
}
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/City/beijing/
http://<serverName>/index.php/City/shanghai/
http://<serverName>/index.php/City/shenzhen/

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

三、空模块实现

现在需求进一步，要求我们把 URL 由原来的

http://<serverName>/index.php/City/shanghai/

变成

http://<serverName>/index.php/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 Action 类，然后在每个 Action 类的 index 方法里面进行处理。

可是如果使用空模块功能，这个问题就可以迎刃而解了。

我们可以给项目定义一个 EmptyAction 类

```
Class EmptyAction extends Action{
Public function index(){
// 根据当前模块名称来判断要执行哪个城市的操作
$cityName = MODULE_NAME;
$this->city($cityName);
}
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/beijing/
http://<serverName>/index.php/shanghai/
http://<serverName>/index.php/shenzhen/

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

2.8 CURD 和自动化支持

ThinkPHP 提供了灵活和方便的数据操作方法，不仅实现了对数据库操作的四大基本操作（CURD）：创建、读取、更新和删除的实现，还内置了很多实用的数据操作方法，提供了 ActiveRecords 模式的最佳体验。

新建记录

```
// 实例化一个 User 模型对象
$User = new UserModel();
// 然后给数据对象赋值
$User->name = 'ThinkPHP';
$User->email = 'ThinkPHP@gmail.com';
// 然后就可以保存新建的 User 对象了
$User->add();
// 如果需要锁实例化模型对象的时候传入数据，可以使用
$data['name'] = 'ThinkPHP';
$data['email'] = 'ThinkPHP@gmail.com';
$User = new UserModel($data);
$User->add();
// 或者直接在 add 方法传入要新建的数据
$data['name'] = 'ThinkPHP';
$data['email'] = 'ThinkPHP@gmail.com';
$User = new UserModel();
$User->add($data);
```

如果你的主键是自动增长类型，不需要传入主键的值就可以新建数据，并且如果插入数据成功的话，Add 方法的返回值就是最新插入的主键值，可以直接获取。

```
$insertId = $User->add($data);
```

一般情况下，应用中的数据对象不太可能通过手动赋值的方式写入，而是有个数据对象的创建过程。

ThinkPHP 提供了一个 create 方法来创建数据对象，然后进行其它的新增或者编辑操作。

```
$User = D("User");
```

```
$User->create(); // 创建 User 数据对象，默认通过表单提交的数据进行创建

$User->add(); // 新增表单提交的数据
```

Create 方法支持从其它方式创建数据对象，例如，从其它的数据对象，或者数组等

```
$data['name'] = 'ThinkPHP';
$data['email'] = 'ThinkPHP@gmail.com';
$User->create($data);

// 从 User 数据对象创建新的 Member 数据对象

$Member = D("Member");
$Member->create($User);
```

支持新增多条记录

```
$User = new UserModel();
$data[0]['name'] = 'ThinkPHP';
$data[0]['email'] = 'ThinkPHP@gmail.com';

$data[1]['name'] = '流年';

$data[1]['email'] = 'liu21st@gmail.com';
$User->addAll($data);
```

在 MySql 数据库下面，会自动使用一条 SQL 语句实现多数据的插入。

查询记录

读取数据库的记录我觉得是数据库操作中的最有意思的一件事情了，写过文本数据库的人都知道，保存和删除数据不难（无非是规范和效率问题），难在可以通过各种方式来查找需要的数据。ThinkPHP 通过各种努力，让数据库的查询操作变得轻而易举，也让 ThinkPHP 变得富有内涵。

ThinkPHP 有一个非常明确的约定，就是单个数据查询和多个数据查询的方法是分开的，或者你会认为有时候自己也不知道要查询的数据是单个还是多个，但是有一点是明确的，你需要的是返回一个数据还是希望返回的是一个数据集。因为对两种类型的返回数据的操作方式是截然不同的，无论何种方式的返回，我们都可以直接在模型对象里面操作，当然也一样可以作为数据传递给你需要的变量。

先举个最简单的例子，假如我们要查询主键为 8 的某个用户记录，我们可以使用下面的一些方法：

```
$User->find(8);
```

这个作为查询语言来说是最为直观的，如果查询成功，查询的结果直接保存在当前的数据对象中，在进行下一次查询操作之前，我们都可以提取，例如获取查询的结果数据：

```
$name = $User->name;
$email = $User->email;
```

遍历查询到的数据对象属性

```
foreach ($User as $key=>$val){
        echo($key.':'.$val);
}
```

// 或者进行相关的数据更改和保存操作

也可以用变量保存下来以便随时使用。

```
$user = $User->find(8);
```

对于上面的查询条件，我们还可以使用 getById 来完成相同的查询

```
$User->getById(8);
```

需要注意的是，对于 find 方法来说，即使查询结果有多条记录，也只会返回符合条件的第一条记录，如果要返回符合要求的所有记录，请使用 findAll 方法。

// 查询主键为 1、3、8 的记录集

```
$User->findAll('1,3,8');
// 遍历数据列表
foreach ($User as $vo){
        dump($vo->name);
}
```

更多的查询操作请参考后面章节的内容。

更新记录

了解了查询记录后，更新操作就显得非常简单了。

```
$User->find(1); // 查找主键为 1 的数据

$User->name = 'TOPThink'; // 修改数据对象

$User->save(); // 保存当前数据对象
```

```
// 还可以使用下面的方式更新

$User->score = '(score+1)'; // 对用户的积分加 1

$User->save();
```

如果不是使用数据对象的方式来保存，可以传入要保存的数据和条件

```
$data['id'] = 1;
$data['name'] = 'TopThink';
$User->save($data);
```

除了 save 方法外，你还可以使用 setField 方法来更新特定字段的值，例如：

```
$User->setField('name','TopThink','id=1');
```

同样可以支持对字段的操作

```
$User->setField('score','(score+1)','id=1');

// 或者改成下面的

$User->setInc('score','id=1');
```

删除记录

```
$User->find(2);

$User->delete(); // 删除查找到的记录

$User->delete('5,6'); // 删除主键为 5、6 的数据

$User->deleteAll(); // 删除查询出来的所有数据
```

2.9 丰富的查询语言

ThinkPHP 大多数情况使用的都是对象查询，因为充分利用了 ORM 查询语言，了解查询条件的定义对使用对象查询非常有帮助，对于复杂的查询，或者从安全方面考虑，通常我们可以使用 HashMap 对象或者索引数组来传递查询条件。查询条件可以用于 find、findAll 等所有有查询条件的方法，下面是几种查询条件的定义：

普通查询

```
$condition = new HashMap();
```

```
// 查询 name 为 thinkphp 的记录
$condition->put('name','thinkphp');
// 使用数组作为查询条件
$condition = Array();
$condition['name'] = 'thinkphp';
```

使用 Map 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。

条件查询

在查询条件里面，如果仅仅使用

$map->put('name','thinkphp');

查询条件应该是 name = 'thinkphp'

如果需要进行其它方式的条件判断，可以使用

```
$map->put('name',array('like','thinkphp%'));
这样，查询条件就变成 name like 'thinkphp%'
$map->put('id',array('gt',100));
查询条件 id > 100
$map->put('id',array('in','1,3,8'));
// 或者使用数组范围
$map->put('id',array('in',array(1,3,8)));
```

上面表示的查询条件都是 id in (1,3,8)

支持的查询表达式有

EQ|NEQ|GT|EGT|LT|ELT|LIKE|BETWEEN|IN|NOT IN

区间查询

ThinkPHP 支持对某个字段的区间查询，例如：

```
$map->put('id',array(1,10)); // id >=1 and id<=10
$map->put('id',array('10','3','or')); //id >= 10 or id <=3
```

```
$map->put('id',array(array('neq',6),array('gt',3),'and')); // id != 6 and id > 3
```

组合查询

如果进行多字段查询，那么字段之间的逻辑关系是 逻辑与 AND，但是用下面的规则可以更改默认的逻辑判断，例如下面的查询条件：

```
$map->put('id',array('neq',1));
$map->put('name','ok');
// 现在的条件是 id !=1 and name like '%ok%'
$map->put('_logic','or');
// 现在的条件变为 id !=1 or name like '%ok%'
```

多字段查询

ThinkPHP 还支持直接对进行多字段查询的方法，可以简化查询表达式和完成最复杂的查询方法，例如：

```
$map->put('id,name,title',array(array('neq',1),array('like','aaa'),array('like','bbb'),'or'));
```

查询条件是

```
( id != 1) OR ( name like 'aaa') OR ( title like '%bbb%')
```

如果结合上面的几种方式，我们可以写出下面更加复杂的查询条件

```
$map->put('id',array('NOT IN','1,6,9'));
$map->put('name,email',array(array('like','thinkphp'),array('like','liu21st'),'or'));
```

以上查询条件变成：

```
( id NOT IN(1,6,9) ) AND ( ( name like 'aaa') OR ( title like '%bbb%') )
```

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。

```
// 获取用户数
$userCount = $User->count();
// 获取用户的最大积分
```

```
$maxScore = $User->max('score');
// 获取积分大于 0 的用户的最小积分
$minScore = $User->min('score','score>0');
// 获取用户的平均积分
$avgScore = $User->avg('score');
// 统计用户的总成绩
$sumScore = $User->sum('score');
```

定位查询

ThinkPHP 支持定位查询，可以使用 getN 方法直接返回查询结果中的某个位置的记录。例如：

```
// 返回符合条件的第 2 条记录
$User->getN(2,'score>80','score desc');
还可以获取最后第二条记录
$User->getN(-2,'score>80','score desc');
如果要查询第一条记录，还可以使用
$User->first('score>80','score desc');
// 获取最后一条记录
$User->last('score>80','score desc');
// 获取积分最高的前 5 条记录
$User->top(5,'','score desc');
```

动态查询

借助 PHP5 语言的特性，ThinkPHP 实现了动态查询。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号条件的记录。

```
$user = $User->getByName('liu21st');
```

上面的查询会转化为$User->getBy('name','liu21st')的查询语言来执行

```
$user = $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 findAll 方法进行查询。

ThinkPHP 还提供了另外一种动态查询方式，就是获取符合条件的前 N 条记录

例如，我们需要获取当前用户中积分大于 0，积分最高的前 5 位用户

```
$User->top5('score>0','*','score desc');
```

而在另外一个频道，我们需要获取点击最多的前 10 位主播

```
$Master->top10('','*','visit desc');
```

原生 SQL 查询

ThinkPHP 支持原生的 SQL 查询，在某些特殊的情况下可以满足应用的需要。

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，所以永远是返回的数据集对象或者惰性数据查询对象。而且可以支持查询缓存、延迟加载和事务锁（悲观锁）。

SQL 查询使用 query 方法

```
$list = $User->query('select id,name from think_user');
```

2.10 分布式数据库支持

ThinkPHP 的模型支持主从式数据库的连接，配置 DB_DEPLOY_TYPE 为 1 可以采用分布式数据库支持。

如果采用分布式数据库，定义数据库配置信息的方式如下：

```
// 在项目配置文件里面定义
Return array(
'DB_TYPE'=> 'mysql', // 分布式数据库的类型必须相同
'DB_HOST'=> '192.168.0.1,192.168.0.2',
'DB_NAME'=>'thinkphp', // 如果相同可以不用定义多个
'DB_USER'=>'user1,user2',
'DB_PWD'=>'pwd1,pwd2',
'DB_PORT'=>'3306',
'DB_PREFIX'=>'think_',
…… 其它项目配置参数
```

```
);
```

连接的数据库个数取决于 DB_HOST 定义的数量，所以即使是两个相同的 IP 也需要重复定义，但是其他的参数如果存在相同的可以不用重复定义，例如：

```
'DB_PORT'=>'3306,3306' 和 'DB_PORT'=>'3306' 等效
'DB_USER'=>'user1',
'DB_PWD'=>'pwd1',
和
'DB_USER'=>'user1,user1',
'DB_PWD'=>'pwd1,pwd1',
```

等效。

还可以设置分布式数据库的读写是否分离，默认的情况下读写不分离，也就是每台服务器都可以进行读写操作，对于主从式数据库而言，需要设置读写分离，通过下面的设置就可以：

```
'DB_RW_SEPARATE'=>true,
```

在读写分离的情况下，第一个数据库配置是主服务器的配置信息，负责写入数据，其它的都是从数据库的配置信息，负责读取数据，数量不限制。每次连接从服务器并且进行读取操作的时候，系统会随机进行在从服务器中选择。

注意事项

主从数据库的数据同步工作不在框架实现，需要数据库考虑自身的同步或者复制机制。

2.11 多数据库支持

分布式数据库的配置信息是定义在配置文件里面的，所以一般情况下是无法更改的。另外使用分布式数据库有个不足，就是无法同时连接多个不同类型的数据库。

多数据库支持

如果你的应用需要在特殊的时候连接多个数据库，那么可以尝试使用 ThinkPHP 的多数据库连接特性：

包括相同类型的数据库和不同类型的数据库。

注意：所谓的相同类型数据库的定义是指和项目配置文件或者模型的数据库连接的数据库类型相同。

我们首先需要在模型类里面增加需要的数据库连接，例如：

我们在 UserModel 类增加多个数据库连接

首先定义额外的数据库连接信息

```
$myConnect1 = array(
'dbms'     => 'mysql',
'username' => 'username',
'password' => 'password',
'hostname' => 'localhost',
'hostport' => '3306',
'database' => 'dbname'
);
```

或者使用下面的定义

```
$myConnect1 = 'mysql://username:passwd@localhost:3306/DbName';
```

定义之后就可以进行动态的增加和切换数据库了。

```
$User->D("User");
// 增加数据库连接 第二个参数表示连接的序号
// 注意内置的数据库连接序号是 0,所以额外的数据库连接序号应该从 1 开始
$User->addConnect($myConnect1,1);
// 可以同时增加多个数据库连接 myConnect2 和 myConnect3 的定义方式同 myConnect1
$User->addConnect($myConnect1,1);
$User->addConnect($myConnect2,2);
$User->addConnect($myConnect3,3);
```

这样在 UserModel 里面就同时存在了 4 个数据库（加上项目配置里面定义的）连接。那么我们如何使用这些不同的数据库连接呢？ThinkPHP 采用了灵活的切换机制，由应用来控制不同的数据库连接。例如，我们需要在其中一个应用里面用到$myConnect2 这个数据库连接，那么用下面的方法切换即可：

```
$User->switchConnect($myConnect2);
```

switchConnect 方法会智能识别该连接是否是相同类型的连接

如果需要删除之前动态添加的连接，可以使用 delConnect 方法，例如：

```
// 删除连接序号为 2 的数据库连接
$User->delConnect(2);
```

2.12 模型自动验证和处理

模型数据验证

系统内置了数据对象的自动验证功能 ,而大多数情况下面 ,数据对象是由表单提交的$_POST 数据创建。

需要使用系统的自动验证功能，只需要在 Model 类里面定义$_validate 属性，是由多个验证因子组成的数组，支持的验证因子格式：

```
array(验证字段,验证规则,错误提示,验证条件,附加规则,验证时间)
```

验证字段就是定义需要验证的表单字段 ,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。

验证规则 要进行验证的规则，需要结合附加规则

提示信息 用于验证失败后的提示信息定义

验证因子中上面三个参数必须定义，下面为可选参数。

验证条件

EXISTS_TO_VAILIDATE 或者 0 存在字段就验证 （默认）

MUST_TO_VALIDATE 或者 1 必须验证

VALUE_TO_VAILIDATE 或者 2 值不为空的时候验证

附加规则 配合验证规则使用，包括：

function 使用函数验证，前面定义的验证规则是一个函数名

callback 使用方法验证，前面定义的验证规则是一个当前 Model 类的方法

confirm 验证表单中的两个字段是否相同，前面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

in 验证是否在某个范围内，前面定义的验证规则必须是一个数组

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

regex 使用正则进行验证，表示前面定义的验证规则是一个正则表达式（默认）

如果采用正则进行验证，会调用系统内置的验证类进行验证操作，该验证类位于 ORG.Text.Validation，

通过正则的方式对数据进行验证，并定义了一些常用的验证规则。包括：

require 字段必须

email 邮箱

url URL 地址

currency 货币

number 数字

这些验证规则可以直接使用。

验证时间:

all 全部情况下验证（默认）

add 新增数据时候验证

edit 编辑数据时候验证

示例：

```
var $_validate          =          array(
```

```
array('verify','require','验证码必须！'), //所有情况下用正则进行验证
array(name,'','帐号名称已经存在！',0,'unique','add'), // 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
);
```

当使用系统的 create 方法创建数据对象的时候会自动进行数据验证操作，代码示例：

```
$User = D("User");
$vo = $User->create();
if (!$vo){
// 如果创建失败 表示验证没有通过 输出错误提示信息
$this->error($User->getError());
}
```

数据处理

在 Model 类定义 $_auto 属性，可以完成数据自动处理功能，用来处理默认值和其他系统写入字段。

$_auto 属性是由多个填充因子组成的数组，填充因子定义格式：

```
array(填充字段,填充内容,填充条件,附加规则)
```

填充字段就是需要进行处理的表单字段 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。

填充条件包括：

ADD 新增数据的时候处理（默认方式）

UPDATE 更新数据的时候处理

ALL 所有情况下都进行处理

附加规则包括：

function 使用函数

callback 回调方法

field 用其它字段填充

string 字符串（默认方式）

示例：

```
var $_auto = array (
array('status','1','ADD'),  // 默认把 status 字段设置为 1
array('password','md5','ADD','function') // 对 password 字段在新增的时候使
md5 函数处理
array('name','getName','ADD','callback') // 对 name 字段在新增的时候回
getName 方法
array('mTime','time','UPDATE','function'), // 对 mTime 字段在编辑的时候写入
当前时间戳
);
```

PS：该自动填充可能会覆盖表单提交项目。其目的是为了防止表单非法提交字段。

使用 Model 类的 create 方法创建数据对象的时候会自动进行表单数据处理

2.13 可配置静态缓存

ThinkPHP 内置了静态缓存的功能，并且支持静态缓存的规则定义。

要使用静态缓存功能，需要开启 HTML_CACHE_ON 参数，并且在项目配置目录下面增加静态缓存规则文件 htmls.php，规则的定义方式如下：

```
Return Array(
'ActionName'=>array('静态规则','静态缓存有效期','附加规则'),
'ModuleName:ActionName'=>array('静态规则','静态缓存有效期','附加规则'),
'*'=>array('静态规则','静态缓存有效期','附加规则'),
…更多操作的静态规则
)
```

静态缓存文件的根目录在 HTML_PATH 定义的路径下面，并且只有定义了静态规则的操作才会进行静态缓存，注意，静态规则的定义有三种方式，

第一种是定义全局的操作静态规则，例如定义所有的 read 操作的静态规则为

```
'read'=>array('{id}','60')
```

其中，{id} 表示取$_GET['id'] 为静态缓存文件名，第二个参数表示缓存 60 秒

第二种是定义某个模块的操作的静态规则，例如，我们需要定义 Blog 模块的 read 操作进行静态缓存

```
'Blog:read'=>array('{id}',-1)
```

第三种方式是定义全局的静态缓存规则，这个属于特殊情况下的使用，任何模块的操作都适用，例如

```
'*'=>array('{$_SERVER.REQUEST_URI|md5}')
```

根据当前的 URL 进行缓存

静态规则的写法可以包括以下情况

使用系统变量 包括 _GET _REQUEST _SERVER _SESSION _COOKIE

格式：{$_×××|function}

例如：{$_GET.name} {$_SERVER. REQUEST_URI}

框架变量 框架特定的变量

例如：{:module} {:action} 表示当前模块名和操作名

使用_GET 变量

{var|function}

也就是说 {id} 其实等效于 {$_GET.id}

直接使用函数

{|function}

例如：{|time}

支持混合定义，例如我们可以定义一个静态规则为：

```
'{id},{name|md5}'
```

在{}之外的字符作为字符串对待，如果包含有"/"，会自动创建目录。

例如，定义下面的静态规则：

```
{:module}/{:action}_{id}
```

则会在静态目录下面创建模块名称的子目录，然后写入操作名_id.shtml 文件。

静态有效时间如果不定义，则会获取配置参数 HTML_CACHE_TIME 的设置值

附加规则通常用于对静态规则进行函数运算，例如

```
'read'=>array('Think{id},{name}','60','md5')
```

翻译后的静态规则是 md5('Think'.$_GET['id'].','.$_GET['name']);

页面静态化后读取的规则有两种方式，通过 HTML_READ_TYPE 设置：

一种是直接读取缓存文件输出（ readfile 方式 HTML_READ_TYPE 为 0） 这是系统默认的方式，属于隐

含静态化，用户看到的 URL 地址是没有变化的。

另外一种方式是重定向到静态文件的方式(HTML_READ_TYPE 为 1),这种方式下面 ,用户可以看到 URL 的地址属于静态页面地址，比较直观。

2.14 多元化缓存机制

ThinkPHP 在数据缓存方面包括 SQL 查询缓存、数据对象缓存、Action 缓存、视图缓存、静态页面缓存以及浏览器缓存等多种机制，采用了包括文件方式、共享内存方式和数据库方式在内的多种方式进行缓存，通过插件方式还可以增加以后需要的缓存类，让应用开发可以选择更加适合自己的缓存方式，从而有效地提高应用执行效率。

ThinkPHP 把各种缓存方式都抽象成统一的缓存类来调用，而且 ThinkPHP 把所有的缓存机制统一成一个 S 方法来进行操作，所以在使用不同的缓存方式的时候并不需要关注具体的缓存细节。

那么如何操作缓存呢？很简单，使用内置的 S 方法，例如：

```
// 使用 data 标识缓存$Data 数据
S('data',$Data);
// 缓存$Data 数据 3600 秒
S('data',$Data,3600);
// 获取缓存数据
$Data = S('data');
```

系统默认的缓存方式是采用 File 方式缓存，我们可以在项目配置文件里面定义其他的缓存方式，例如修改默认的缓存方式为 Xcache（当然，你的环境需要支持 Xcache）

```
'DATA_CACHE_TYPE'=>'Xcache'
```

通过上面的定义，相同的代码就会使用 Xcache 方式来缓存了，而事实上，代码并没有任何改变。

```
// 使用 data 标识缓存$Data 数据  有效期为默认的设置
S('data',$Data);
// 缓存$Data 数据 3600 秒
```

```
S('data',$Data,3600);
// 获取缓存数据
$Data = S('data');
```

当然，我们还可以在 S 方法里面显式的指定缓存方式，例如

```
S('data',$Data,3600,'File');
// 或者动态切换缓存方式
C('DATA_CACHE_TYPE','Xcache');
S('data',$Data,3600);
$data = S('data');
// 操作完成后切换会默认的缓存方式
C('DATA_CACHE_TYPE','File');
```

对于 File 方式缓存下的缓存目录下面因为缓存数据过多而导致存在大量的文件问题，ThinkPHP 也给出了解决方案，可以启用哈希子目录缓存的方式，只需要设置

```
'DATA_CACHE_SUBDIR'=>true
```

就可以根据缓存标识的哈希自动创建子目录来缓存。

相关的缓存配置参数还包括：

```
'DATA_CACHE_TIME'=>-1,              // 数据缓存有效期
'DATA_CACHE_COMPRESS'=>false,       // 数据缓存是否压缩缓存
'DATA_CACHE_CHECK'        =>false,  // 数据缓存是否校验缓存
'DATA_CACHE_TYPE'        =>'File',  // 数据缓存类型 支
持 File Db Apc Memcache Shmop Sqlite Xcache Apachenote Eaccelerator
'DATA_CACHE_SUBDIR'=>false, // 使用子目录缓存（自动根据缓存标识的哈希创建子目录）
```

2.15 内置模板引擎

ThinkPHP 内置了一个基于 XML 的性能卓越的模板引擎 ThinkTemplate ,这是一个专门为 ThinkPHP 服务的模板引擎。ThinkTemplate 是一个使用了 XML 标签库技术的编译型模板引擎，支持两种类型的模板标签，使用了动态编译和缓存技术，而且支持自定义标签库。其特点包括：

1、支持 XML 标签库和普通标签的混合定义；

2、编译一次，下次直接运行而无需重新编译；

3、模板文件更新后，自动更新模板缓存；

4、自动加载当前操作的模板缓存文件，无需指定模板文件名称；

5、支持编码转换和 Content-Type 更换；

6、模板变量输出前缀支持，避免变量名称冲突；

7、模板常量替换，无需设置模板变量；

8、支持变量组合调节器和格式化功能；

9、支持替换其它模板引擎插件使用；

10、支持获取模板页面内容

如果在 ThinkPHP 框架中使用的话，无需创建 ThinkTemplate 对象，Action 类会自动创建，只需要赋值并输出就行了。

```
$this->assign('vo',$vo);
$this->display();
```

ThinkPHP 内置模板引擎的的模板标签有两种类型：第一种是普通标签，类似于 Smarty 的模板标签，在功能方面作了部分简化，增强了显示功能，弱化了逻辑控制功能；第二种是 XML 标签库形式，有效地借鉴了 Java 的标签库技术，在控制功能和各方面都比较强大，而且允许自定义标签库，是新版 ThinkPHP 系统引入和推荐的模板标签技术。两种标签方式的结合使用，可以让您的模板定义功能相当强大。

ThinkPHP 架构的设计中模板和程序完全分离，一套模板文件就是一个目录，模板是标准 html 文件（可以配置成其它后缀，如.shtml，.xml 等），可以单独预览。

由于使用了模板动态缓存技术 在您第一次运行某个模块的某个操作时候 对应的模板文件就会被缓存，下次读取的时候，无论是模板文件修改或者是缓存文件被删除，系统都会重新生成缓存文件。你还可以设置模板缓存的有效时间间隔，如每隔 10 分钟重新读取模板文件。模板动态缓存只是让您免去每次重复编译模板的时间。

ThinkTemplate 提供了一定程度的功能，虽然卓越但是需要一个熟悉和掌握的过程，如果你已经熟悉了另外一种模版引擎的使用，而目前的项目又不允许你花更多的时间来学习内置的模版引擎，没有关系，ThinkPHP 框架允许你使用第三方的模版引擎。目前官方已经提供了 Smarty 模版引擎的插件，已经有人给 ThinkPHP 开发了 TemplateLite、EaseTempalte 和 DzTemplate 模版引擎插件。而且对于自己熟悉的模版引擎来说，非常容易扩展类似的插件。然后在项目配置文件里面配置使用何种模板引擎就可以了。

内置模板引擎也支持直接书写 PHP 代码，还可以设置 TMPL_ENGINE_TYPE 为 php，则使用纯 PHP 代码作为模板引擎支持。

2.16 系统基类库

这里对 ThinkPHP 的基类库概念做个介绍

系统基类库

ThinkPHP 框架通过基类库的概念把所有系统类库都集成在一起，包括 ThinkPHP 的核心类库。

目前的基类库分成 Think 核心类库、ORG 扩展类库，在这主要介绍的是核心基类库，核心基类库的作用

是完成框架的通用性开发而必须的基础类和常用工具类等，包含有：

Think.Core 核心类库包

Think.Db 数据库类库包

Think.Util 系统工具类库包

Think.Template 内置模板引擎类库包

Think.Exception 异常处理类库包

函数库

ThinkPHP 内置了一个系统公共函数库，提供了一些系统需要的函数，系统函数库位于 ThinkPHP 的 Common 目录下面，名称为 functions.php。

每个项目可以定义自己的函数库，位于项目的 Common 目录下面的 common.php 文件。

如果项目在 Common 目录下面有定义自己的 common.php 文件，框架会在初始化的时候自动导入，而无需自己导入。

匹配导入

Import 方法是 ThinkPHP 内建的类库和文件导入方法，提供了方便和灵活的类似于命名空间的文件导入机制。例如：

```
Import("Think.util.ListIterator");
Import("App.Model.UserModel");
```

import 方法不仅可以导入系统基类库文件，还可以导入项目类库文件。并且可以自动识别是基类库和项目类库，通常来说基类库用于放置一些通用的类库，而项目类库主要是项目相关的一些类库文件。

Import 方法具有缓存和检测机制，相同的文件不会重复导入，如果发现导入了不同的位置下面的同名类库文件，系统会提示冲突，例如：

```
Import("Think.Util.Array");
Import("App.Util.Array");
```

上面的情况导入会产生引入两个同名的 Array.class.php 类，即使实际上的类名可能不存在冲突，但是按照 ThinkPHP 的规范，类名和文件名是一致的，所以系统会抛出类名冲突的异常，并终止执行。

除了正常的导入操作为，还支持模式匹配导入和多文件导入。例如：

导入 Think.Util 目录下面的所有类文件

```
Import("Think.Util.*");
Import("App.Model.*");
```

注意：使用子目录引入的方式，如果目录下面文件较多会给系统带来较大的目录遍历开销。

对于 Import 方法，系统会自动识别导入类库文件的位置，ThinkPHP 的约定是 Think、ORG、Com 包的导入以系统基类库为相对起始目录，否则就认为是项目应用类库为起始目录，如果是其它情况的导入，需要指定 baseUrl 参数，也就是 import 方法的第二个参数。例如，要导入当前文件所在目录下面的 RBAC/AccessDecisionManager.class.php 文件：

```
Import("RBAC. AccessDecisionManager",dirname(__FILE__));
```

智能导入

Import 方法具有智能的模式匹配导入机制，例如下面的例子使用了更高级的匹配模式导入：

```
Import("Think.*.Array");
Import("ORG.Util.Array*");
Import("ORG.Util.*Map");
Import("App.Util.?Tool");
```

ThinkPHP 的类文件命名规则是以.class.php 作为后缀，所以默认的导入只会匹配.class.php 为后缀的文件，如果你使用的第三方类库文件没有使用.class.php 作为类文件命名规范，那么需要指定后缀，例如，导入 Com.Zend.Search 目录下面的所有类文件,但不包括子目录，并且这些类库文件的后缀是.php，我们就需要指定 import 方法的第三个参数 ext：

```
Import("Com.Zend.Search.*",'','.php');
```

为了方便类似的导入机制，系统给第三方类库专门指定了一个 vendor 方法，该方法默认的导入类文件的后缀是.php，并且起始路径是 ThinkPHP 系统目录下面的 Vendor 目录。其它使用方法相同。

导入第三方类库

我们知道 ThinkPHP 的基类库都是以.class.php 为后缀的，这是系统内置的一个约定，当然也可以通过 import 的参数来控制，为了更加方便引入其他框架和系统的类库，系统增加了导入第三方类库的功能，第三方类库统一放置在系统的 Vendor 目录下面，并且使用 vendor 方法导入，其参数和 import 方法是一致的，只是默认的值有针对变化。

例如，我们把 Zend 的 Filter\Dir.php 放到 Vendor 目录下面，这个时候 Dir 文件的路径就是 Vendor\Zend\Filter\Dir.php，我们使用 vendor 方法导入就是：

```
Vendor('Zend.Filter.Dir');
```

类库扩展

系统基类库可以很方便的进行扩展，你可以在 ORG 类库目录下面添加自己需要的类库（ThinkPHP 建议所有的类库文件用 class.php 作为后缀，并且文件名和类名相同），你甚至还可以创建属于自己企业的类库，只需要在 ThinkPHP\Lib\目录下面创建 Com 目录，然后在里面增加相应的类库就可以方便的使用 import 方法导入了。

例如，我们在 ThinkPHP\Lib\Com\下面创建了 Sina 目录，并且放了 Util\UnitTest.class.php 类库文件，可以使用下面的方式导入

```
import('Com.Sina.Util.UnitTest');
```

如果你使用的是第三方的类库包，而且类名和后缀和 Tp 的默认规则不符合，那我们建议你放到 Vendor 目录下面，使用 vendor 方法来导入。

2.17 自动编码转换

ThinkPHP 框架的文件全部采用 UTF-8 编码格式，UTF-8 的支持和自动输出编码转换的功能让页面表现更加灵活。您可以配置输出的页面编码格式，如 gb2312 等（默认采用 UTF-8 输出）。系统根据配置文件

中设置的编码格式自动对页面进行编码转换，支持 iconv 和 mb_string 两种方式，为了提高效率，如果系统的模板编码设置和输出编码设置项相同，则不会进行编码转换。ThinkPHP 可以设置模板编码、输出编码和数据库编码，并自动完成转换工作，让你的应用不再受编码的苦恼。

如果你的项目编码都是 UTF-8 格式的，你可能无需了解 ThinkPHP 的自动编码转换的细节，如果你的项目里面存在个别的非 UTF-8 编码的情况，就需要了解 TP 的自动编码转换机制。首先让我们来了解下 TP 中有哪些编码是可以配置的，下面是惯例配置里面的一些编码设置：

代码: 复制内容到剪贴板

```
'TEMPLATE_CHARSET'=> 'utf-8',         // 模板文件编码

'OUTPUT_CHARSET' => 'utf-8',         // 默认输出编码

'DB_CHARSET'     => 'utf8',         // 数据库编码默认采用 utf8
```

所以，如果以上参数都配置成 utf-8 的话，系统就无需进行编码转换工作，可以提高一定的效率。但是需要注意的是文件编码在使用 UTF-8 的时候，千万要注意 BOOM 的问题，因为这个隐藏的输出会导致 Cookie、Session 以及 Header 方法失效。关于 BOOM 的问题，可以 Google 下。如果使用 Editplus 的话，建议在参数选择->文件里面设置文件编码为 utf-8，并且总是删除签名。

上面的编码都可以配置成其他的编码格式，系统会完成自动的编码转换。不过，数据库编码的设置需要数据库的支持，有些数据库或者版本并不支持编码设置。

模板文件编码是指模板文件的文件编码 输出编码是指浏览器输出的页面编码，注意这两个不要相混淆。如果你的模板文件编码是 utf-8，而输出编码的设置是 gb2312，数据库的编码是 utf8(这里要注意的是数据库编码是没有 utf-8 的，而应该是 utf8)，那么系统的自动编码转换过程是：

```
从数据库中查询数据(utf8)->读取模板文件 utf-8 并渲染->转换成输出编码 gb2312 输出
```

另外一个需要注意的地方是，设置了输出编码后，模板文件里面最好添加 meta 头信息，例如：

```
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
```

如果你的模板文件编码和输出编码不一致，假设模板文件编码是 utf-8，而输出编码是 gb2312，但是为了不影响模板文件的预览，你可以在模板文件里面定义

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
```

框架模板引擎在编译的时候会自动把 utf-8 转换成输出编码 gb2312。

系统的编码转换方法使用自带的 auto_charset 方法，需要 iconv 或者 mb_string 类库的支持。

2.18 多语言支持

ThinkPHP 内置多语言支持，如果你的应用涉及到国际化的支持，那么可以定义相关的语言包文件。任何字符串形式的输出，都可以定义语言常量。可以为项目定义不同的语言文件，框架的系统语言包目录在系统框架的 Lang 目录下面，每个语言都对应一个语言包文件，系统默认只有简体中文语言包文件 zh-cn.php ，如果要增加繁体中文 zh-tw 或者英文 en，只要增加相应的文件。

语言包的使用由系统自动判断当前用户的浏览器支持语言来定位，如果找不到相关的语言包文件，会使用默认的语言。如果浏览器支持多种语言，那么取第一种支持语言。

ThinkPHP 的多语言支持已经相当完善了，可以满足应用的多语言需求。这里是指的是模板多语言支持，数据的多语言转换（翻译）不在这个范畴之内。ThinkPHP 具备语言包定义、自动识别、动态定义语言参数的功能。并且可以自动识别用户浏览器的语言，从而选择相应的语言包（如果有定义）。例如：

Throw_exception（'新增用户失败！'）；

我们在语言包里面增加了 ADD_USER_ERROR 语言配置变量后，在程序中的写法就要改为：

Throw_exception（L('ADD_USER_ERROR')）；

项目语言包文件位于项目的 Lang 目录下面，并且按照语言类别分子目录存放，在执行的时候系统会自动加载，无需手动加载。语言包文件可以按照模块来定义，每个模块单独定义语言包文件，文件名和模块名称相同，例如：

Lang/zh-cn/user.php 表示给 User 模块定义简体中文语言包文件

Lang/zh-tw/user.php 表示给 User 模块定义繁体中文语言包文件

语言子目录采用浏览器的语言命名(全部小写)定义，例如 English (United States) 可以使用 en-us 作为目录名。如果项目比较小，整个项目只有一个语言包文件，那可以定义 common.php 文件，而无需按照模块分开定义。

语言文件定义

ThinkPHP 语言文件定义采用返回数组方式：

```
return array(
    'lan_define'=>'欢迎使用 ThinkPHP'
);
```

要在程序里面设置语言定义的值，使用下面的方式：

```
L('define2','语言定义');
$value = L('define2');
```

如果要在模板中输出语言变量不需要在 Action 中赋值，可以直接使用模板引擎特殊标签来直接输出语言定义的值：

```
{$Think.lang.define2}
```

上面的语言包是指项目的语言包，如果在提示信息的时候使用了框架底层的提示，那么还需要定义系统的语言包，系统语言包目录位于 ThinkPHP 目录下面的 Lang 目录。

多语言的切换

ThinkPHP 的多语言支持是自动的，系统会根据用户的浏览器语言设置自动获取相关的语言包，如果找到匹配当前的语言包文件，就会自动加载进来，因此不需要你手动加载语言包文件在项目中。

如果你需要测试语言包的话，可以使用 l 参数(可以通过项目配置项 VAR_LANGUAGE 设置)来手动切换，进行效果测试。例如：

http:/ /ServerIP/AppName/Module/Action/?l=zh-tw　//切换到繁体中文

http:/ /ServerIP/AppName/Module/Action?l=en-us //切换到英文

语言包切换后会自动保存 Cookie 记录，所以浏览其他页面的时候仍然可以保持上次选择的语言。

2.19 RBAC 权限控制

企业级的应用是离不开安全保护的，ThinkPHP 以基于 Spring 的 Acegi 安全系统作为参考原型，并做了简化，以适合目前的 ThinkPHP 结构，提供了一个多层的、可定制的安全体系来为应用开发提供安全控制。安全体系中主要有：

安全拦截器

认证管理器

决策访问管理器

运行身份管理器

安全拦截器

安全拦截器就好比一道道门，在系统的安全防护系统中可能存在很多不同的安全控制环节，一旦某个环节你未通过安全体系认证，那么安全拦截器就会实施拦截。

认证管理器

防护体系的第一道门就是认证管理器，认证管理器负责决定你是谁，一般它通过验证你的主体（通常是一个用户名）和你的凭证（通常是一个密码），或者更多的资料来做到。更简单的说，认证管理器验证你的身份是否在安全防护体系授权范围之内。

访问决策管理

虽然通过了认证管理器的身份验证，但是并不代表你可以在系统里面肆意妄为，因为你还需要通过访问决策管理这道门。访问决策管理器对用户进行授权，通过考虑你的身份认证信息和与受保护资源关联的安全属性决定是是否可以进入系统的某个模块，和进行某项操作。

例如，安全规则规定只有主管才允许访问某个模块，而你并没有被授予主管权限，那么安全拦截器会拦截你的访问操作。

决策访问管理器不能单独运行，必须首先依赖认证管理器进行身份确认，因此，在加载访问决策过滤器的时候已经包含了认证管理器和决策访问管理器。

为了满足应用的不同需要，ThinkPHP 在进行访问决策管理的时候采用两种模式 :登录模式和即时模式。

登录模式，系统在用户登录的时候读取改用户所具备的授权信息到 Session，下次不再重新获取授权信息。也就是说即使管理员对该用户进行了权限修改，用户也必须在下次登录后才能生效。

即时模式就是为了解决上面的问题，在每次访问系统的模块或者操作时候，进行即使验证该用户是否具有该模块和操作的授权，从更高程度上保障了系统的安全。

运行身份管理器

运行身份管理器的用处在大多数应用系统中是有限的，例如某个操作和模块需要多个身份的安全需求，运行身份管理器可以用另一个身份替换你目前的身份，从而允许你访问应用系统内部更深处的受保护对象。这一层安全体系目前的 RBAC 中尚未实现。

要启用 RBAC 请在项目配置文件中设置

USER_AUTH_ON 为 True

并设置认证类型 USER_AUTH_TYPE

1 普通认证（认证一次）

2 高级认证（实时认证）

不设置默认为 1

例如，下面的设置可供参考

```
return array(
'USER_AUTH_ON'=>true,
'USER_AUTH_TYPE'    => 1,
'USER_AUTH_KEY'    => 'authId',
'USER_AUTH_PROVIDER' => 'DaoAuthentictionProvider',
'USER_AUTH_GATEWAY' => '/Public/login',
'NOT_AUTH_MODULE'   => 'Public',
'REQUIRE_AUTH_MODULE'=> '',
);
```

认证识别号 USER_AUTH_KEY 是用于检查用户是否经过身份认证的标识，一旦用户经过系统认证，系统

会把该用户编号保存在$_SESSION[C('USER_AUTH_KEY')]中

为了满足应用系统的需要，RBAC 插件中可以设置

REQUIRE_AUTH_MODULE 需要认证的模块

NOT_AUTH_MODULE 无需认证的模块

多个模块之间用逗号分割

如果某个模块需要认证，但是用户还没有经过身份认证，就会跳转到

USER_AUTH_GATEWAY 认证网关，例如 /Public/login

验证地址就是： 项目入口文件 URL 地址/Public/login

假设认证网关的验证操作地址是/Public/CheckLogin，可以在 public 模块的 checkLogin 操作中采用如下

方式进行认证：

```
<?php
// 生成认证 Map 条件

// 这里使用用户名、密码和状态的方式进行认证
$map = array();
$map["name"] = $_POST['name'];
$map["password"] = $_POST['password'];
$map["status"] = 1;
```

```
$authInfo = RBAC: : authenticate($map);
if(false === $authInfo) {
$this->error('登录失败，请检查用户名和密码是否有误！');
}else {
// 设置认证识别号
$_SESSION[C('USER_AUTH_KEY')] = $authInfo['id'];
//获取并保存用户访问权限列表
RBAC: :saveAccessList();
// 登录成功，页面跳转

$this->success('登录成功！');
}
?>
```

RBAC 的委托认证方法

```
authenticate($map,$model='User',$provider=USER_AUTH_PROVIDER)
```

方法是静态方法，支持三个参数，其中第一个认证条件$map 是必须的，可以灵活地控制需要认证的字段。

第二个参数是进行认证的模型类，默认是 UserModel 类

第三个参数是委托方式 由 USER_AUTH_PROVIDER 设置委托认证管理器的委托方式，目前支持的是 DaoAuthentictionProvider 通过数据库进行认证。

在应用系统的开发过程中，只需要设置相关的配置项和添加上面的认证方法，其他的认证和决策访问就由 RBAC 组件的 AccessDecision 方法自动完成了。

系统会在执行某个模块的操作时候，首先判断该模块是否需要认证，如果需要认证并且已经登录，就会获取当前用户的权限列表判断是否具有当前模块的当前操作权限，并进行相应的提示。

接下来就是在框架总后台设置相关项目的模块和操作权限了。

关于如何授权请参考示例中心提供的 RBAC 示例

首先，在节点管理添加相关项目、模块和操作，作为权限管理的节点。

如果需要设置公共的操作，可以使用 Public 模块，所有属于 Public 模块的操作对所有模块都有效。

添加完成项目管理节点后，就在权限管理里面对某个用户组设置相关权限（包括项目权限、模块权限和操作权限）

以后需要授权就把用户添加到某个权限组就可以了，同一个用户可以属于多个权限组。

授权和认证功能涉及到四个数据表，DB_PREFIX 为配置文件中设置的数据库前缀

DB_PREFIX_group 权限组表

DB_PREFIX_groupuser 组-用户关联表

DB_PREFIX_access 访问权限表

DB_PREFIX_node 权限节点表

所谓的授权操作其实就是往 DB_PREFIX_groupuser 表和 DB_PREFIX_access 里面写入数据

至于数据表的字段官方给出的示例仅供参考，可以通过修改 ORG.RBAC.AccessDecisionManager 类来完成项目的需要