ThinkPHP®
中文WEB应用开发框架

完全开发手册

ThinkPHP 3.2.3
中文WEB应用开发框架

ThinkPHP文档小组

看云

序言

> 手册阅读须知：本手册仅针对ThinkPHP3.2.3版本，尽管3.2版本大多数功能都通用，但我们还是建议你把手册里面的特性使用在3.2.3版本（可以使用左右键（ <-- 和 --> ）翻页阅读）

版权申明

捐赠我们

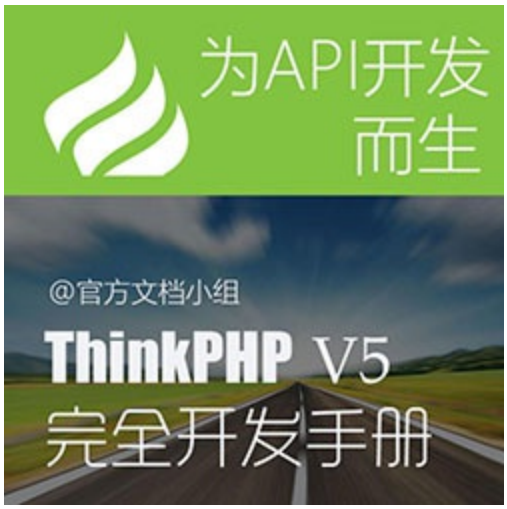
ThinkPHP一直在致力于简化企业和个人的WEB应用开发，您的帮助是对我们最大的支持和动力！

我们的团队10年来一直在坚持不懈地努力，并坚持开源和免费提供使用，帮助开发人员更加方便的进行WEB应用的快速开发，如果您对我们的成果表示认同并且觉得对你有所帮助我们愿意接受来自各方面的捐赠^_^。

用手机扫描进行支付宝捐赠 （ 查看捐赠列表 ）

更多关注

ThinkPHP V5.0RC4版本发布了！



基础

ThinkPHP是一个快速、简单的基于MVC和面向对象的轻量级PHP开发框架，遵循Apache2开源协议发布，从诞生以来一直秉承简洁实用的设计原则，在保持出色的性能和至简的代码的同时，尤其注重开发体验和易用性，并且拥有众多的原创功能和特性，为WEB应用开发提供了强有力的支持。

3.2版本则在原来的基础上进行一些架构的调整，引入了命名空间支持和模块化的完善，为大型应用和模块化开发提供了更多的便利。

3.2.3 主要更新

- 数据库驱动完全用PDO重写；
- 支持通用insertAll方法；
- 改进参数绑定机制；
- 主从分布式数据库连接改进；
- 对Mongo的支持更加完善；
- 模型类的诸多增强和改进；
- 增加聚合模型扩展；
- 支持复合主键；
- 多表操作的支持完善；
- 模型的CURD操作支持仅获取SQL语句而不执行；
- 增加using/index/fetchSql/strict/token连贯操作方法；
- 模型类的setInc和setDec方法支持延迟写入；
- I函数增加变量修饰符和正则检测支持；
- 支持全局变量过滤和Action参数绑定的变量过滤；
- 修正可能的SQL注入漏洞；
- 支持全局路由定义；
- 增加插件控制器支持；
- 增加对全局和模块的模板路径的灵活设置；
- 日志目录分模块存放；
- 增加memcache Session驱动；
- 改进session函数的数组操作；

获取ThinkPHP

获取ThinkPHP的方式很多，官方网站（http://thinkphp.cn）是最好的下载和文档获取来源。

官网提供了稳定版本的下载：http://thinkphp.cn/down/framework.html

> 官网下载版本提供了完整版和核心版两个版本，核心版本只保留了核心类库和必须的文件，去掉了所有的扩展类库和驱动，支持标准模式和SAE模式。

如果你希望保持最新的更新，可以通过github获取当前最新的版本（完整版）。

Git获取地址列表（你可以选择一个最快的地址）：

> Github： `https://github.com/liu21st/thinkphp`
> Oschina： `http://git.oschina.net/liu21st/thinkphp.git`
> Code： `https://code.csdn.net/topthink2011/ThinkPHP`
> Coding： `https://coding.net/u/liu21st/p/thinkphp/git`

3.2版本也支持composer安装，确保你的电脑已经安装了composer，然后在你的web根目录下面执行 `composer create-project topthink/thinkphp your-project-name`

（注意：目前国内的速度很慢 建议直接下载官网版本）

ThinkPHP无需任何安装，直接拷贝到你的电脑或者服务器的WEB运行目录下面即可。

环境要求

框架本身没有什么特别模块要求，具体的应用系统运行环境要求视开发所涉及的模块。ThinkPHP底层运行的内存消耗极低，而本身的文件大小也是轻量级的，因此不会出现空间和内存占用的瓶颈。

PHP版本要求

- PHP5.3以上版本（注意：PHP5.3dev版本和PHP6均不支持）

支持的服务器和数据库环境

- 支持Windows/Unix服务器环境
- 可运行于包括Apache、IIS和nginx在内的多种WEB服务器和模式
- 支持Mysql、MsSQL、PgSQL、Sqlite、Oracle、Ibase、Mongo等多种数据库和连接

> 对于刚刚接触PHP或者ThinkPHP的新手，我们推荐使用集成开发环境WAMPServer（wampserver是一个集成了Apache、PHP和MySQL的开发套件，而且支持不同PHP版本、MySQL版本和Apache版本的切换）来使用ThinkPHP进行本地开发和测试。

目录结构

下载框架后，解压缩到web目录下面，可以看到初始的目录结构如下：

```
www  WEB部署目录（或者子目录）
├─index.php       入口文件
├─README.md       README文件
├─Application     应用目录
├─Public          资源文件目录
└─ThinkPHP        框架目录
```

开发人员可以在这个基础之上灵活调整。其中，`Application`和`Public`目录下面都是空的。

> README.md文件仅用于说明，实际部署的时候可以删除。
> 上面的目录结构和名称是可以改变的，这取决于你的入口文件和配置参数。
> Application目录默认是空的，但是第一次访问入口文件会自动生成，参考后面的入口文件部分。

其中框架目录ThinkPHP的结构如下：

```
├─ThinkPHP 框架系统目录（可以部署在非web目录下面）
│  ├─Common       核心公共函数目录
│  ├─Conf         核心配置目录
│  ├─Lang         核心语言包目录
│  ├─Library      框架类库目录
│  │  ├─Think     核心Think类库包目录
│  │  ├─Behavior  行为类库目录
│  │  ├─Org       Org类库包目录
│  │  ├─Vendor    第三方类库目录
│  │  ├─ ...      更多类库目录
│  ├─Mode         框架应用模式目录
│  ├─Tpl          系统模板目录
│  ├─LICENSE.txt  框架授权协议文件
│  ├─logo.png     框架LOGO文件
│  ├─README.txt   框架README文件
│  └─ThinkPHP.php 框架入口文件
```

> 上述应用的目录结构只是默认设置，事实上，在实际部署应用的时候，我们建议除了应用入口文件和`Public`资源目录外，其他文件都放到非WEB目录下面，具有更好的安全性。

入口文件

ThinkPHP采用单一入口模式进行项目部署和访问，无论完成什么功能，一个应用都有一个统一（但不一定

是唯一）的入口。

应该说，所有应用都是从入口文件开始的，并且不同应用的入口文件是类似的。

入口文件定义

入口文件主要完成：

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

默认情况下，框架已经自带了一个应用入口文件（以及默认的目录结构），内容如下：

```
define('APP_PATH','./Application/');
require './ThinkPHP/ThinkPHP.php';
```

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

```
define('APP_PATH','./Apps/');
require './ThinkPHP/ThinkPHP.php';
```

> 注意：APP_PATH的定义支持相对路径和绝对路径，但必须以“/”结束

如果你调整了框架核心目录的位置或者目录名，只需要这样修改：

```
define('APP_PATH','./Application/');
require './Think/ThinkPHP.php';
```

也可以单独定义一个THINK_PATH常量用于引入：

```
define('APP_PATH','./Application/');
define('THINK_PATH',realpath('../Think').'/');
require THINK_PATH.'ThinkPHP.php';
```

> 和APP_PATH一样THINK_PATH路径定义也必须以“/”结尾。
> 给THINK_PATH和APP_PATH定义绝对路径会提高系统的加载效率。

入口文件中的其他定义

一般不建议在入口文件中做过多的操作，但可以重新定义一些系统常量，入口文件中支持定义（建议）的

一些系统常量包括：

常量	描述
THINK_PATH	框架目录
APP_PATH	应用目录
RUNTIME_PATH	应用运行时目录（可写）
APP_DEBUG	应用调试模式 （默认为false）
STORAGE_TYPE	存储类型（默认为File）
APP_MODE	应用模式（默认为common）

> 注意：所有路径常量都必须以“/”结尾

例如，我们可以在入口文件中重新定义相关目录并且开启调试模式：

```
// 定义应用目录
define('APP_PATH','./Apps/');
// 定义运行时目录
define('RUNTIME_PATH','./Runtime/');
// 开启调试模式
define('APP_DEBUG',True);
// 更名框架目录名称，并载入框架入口文件
require './Think/ThinkPHP.php';
```

这样最终的应用目录结构如下：

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

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

自动生成

自动创建目录

在第一次访问应用入口文件的时候，会显示如图所示的默认的欢迎页面，并自动生成了一个默认的应用模

块Home。

欢迎使用 **ThinkPHP**！

接下来再看原来空的 Application 目录下面，已经自动生成了公共模块 Common 、默认的 Home 模块和 Runtime 运行时目录的目录结构：

```
Application
├─Common       应用公共模块
│  ├─Common    应用公共函数目录
│  └─Conf      应用公共配置文件目录
├─Home         默认生成的Home模块
│  ├─Conf      模块配置文件目录
│  ├─Common    模块函数公共目录
│  ├─Controller 模块控制器目录
│  ├─Model     模块模型目录
│  └─View      模块视图文件目录
├─Runtime      运行时目录
│  ├─Cache     模版缓存目录
│  ├─Data      数据目录
│  ├─Logs      日志目录
│  └─Temp      缓存目录
```

> 如果你不是Windows环境下面的话，需要对应用目录 Application 设置可写权限才能自动生成。
> 如果不是调试模式的话，会在Runtime目录下面生成 common~runtime.php 文件（应用编译缓存文件）。

目录安全文件

在自动生成目录结构的同时，在各个目录下面我们还看到了index.html文件，这是ThinkPHP自动生成的目录安全文件。

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：

```
define('DIR_SECURE_FILENAME', 'default.html');
define('APP_PATH','./Application/');
require './ThinkPHP/ThinkPHP.php';
```

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

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

模块

下载后的框架自带了一个应用目录结构，并且带了一个默认的应用入口文件，方便部署和测试，默认的应用目录是Application（实际部署过程中可以随意设置），应用目录只有一个，因为大多数情况下，我们都可以通过多模块化以及多入口的设计来解决应用的扩展需求。

模块设计

新版采用模块化的设计架构，下面是一个应用目录下面的模块目录结构，每个模块可以方便的卸载和部署，并且支持公共模块。

```
Application      默认应用目录（可以设置）
├─Common         公共模块（不能直接访问）
├─Home           前台模块
├─Admin          后台模块
├─...            其他更多模块
├─Runtime        默认运行时目录（可以设置）
```

每个模块是相对独立的，其目录结构如下：

```
├─Module         模块目录
│  ├─Conf        配置文件目录
│  ├─Common      公共函数目录
│  ├─Controller  控制器目录
│  ├─Model       模型目录
│  ├─Logic       逻辑目录（可选）
│  ├─Service     Service目录（可选）
│  ... 更多分层目录可选
│  └─View        视图目录
```

> 由于采用多层的MVC机制，除了Conf和Common目录外，每个模块下面的目录结构可以根据需要灵活设置和添加，所以并不拘泥于上面展现的目录

控制器

我们可以在自动生成的Application/Home/Controller目录下面找到一个 IndexController.class.php 文件，这就是默认的Index控制器文件。

控制器类的命名方式是：控制器名（驼峰法，首字母大写）+Controller

控制器文件的命名方式是：类名+class.php（类文件后缀）

默认的欢迎页面其实就是访问的Home模块下面的Index控制器类的index操作方法 我们修改默认的index操作方法如下：

```
namespace Home\Controller;
use Think\Controller;
class IndexController extends Controller {
    public function index(){
        echo 'hello,world!';
    }
}
```

再次运行应用入口文件，浏览器会显示： hello,world! 。

我们再来看下控制器类，IndexController控制器类的开头是命名空间定义：

```
namespace Home\Controller;
```

这是系统的规范要求，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，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，而且之前不能有任何输出，否则会出错

```
use Think\Controller;
```

表示引入 Think\Controller 类库便于直接使用。 所以，

```
namespace Home\Controller;
use Think\Controller;
class IndexController extends Controller
```

等同于使用：

```
namespace Home\Controller;
class IndexController extends \Think\Controller
```

开发规范

命名规范

使用ThinkPHP开发的过程中应该尽量遵循下列命名规范：

- 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 ；
- 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）；
- 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；
- 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）；
- 函数的命名使用小写字母和下划线的方式，例如 get_client_ip ；
- 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；
- 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；
- 以双下划线“__”打头的函数或方法作为魔法方法，例如 __call 和 __autoload ；
- 常量以大写字母和下划线命名，例如 HAS_ONE 和 MANY_TO_MANY ；
- 配置参数以大写字母和下划线命名，例如 HTML_CACHE_ON ；
- 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_ ；
- 对变量的命名没有强制的规范，可以根据团队规范来进行；
- ThinkPHP的模板文件默认是以 .html 为后缀（可以通过配置修改）；
- 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。

特例：在ThinkPHP里面，有一个函数命名的特例，就是单字母大写函数，这类函数通常是某些操作的快捷定义，或者有特殊的作用。例如：A、D、S、L 方法等等，他们有着特殊的含义，后面会有所了解。

> 由于ThinkPHP默认全部使用UTF-8编码，所以请确保你的程序文件采用UTF-8编码格式保存，并且去掉BOM信息头（去掉BOM头信息有很多方式，不同的编辑器都有设置方法，也可以用工具进行统一检测和处理），否则可能导致很多意想不到的问题。

开发建议

在使用ThinkPHP进行开发的过程中，我们给出如下建议，会让你的开发变得更轻松：

> - 遵循框架的命名规范和目录规范；
> - 开发过程中尽量开启调试模式，及早发现问题；
> - 多看看日志文件，查找隐患问题；
> - 养成使用I函数获取输入变量的好习惯；
> - 更新或者环境改变后遇到问题首要问题是清空Runtime目录；

配置

ThinkPHP提供了灵活的全局配置功能，采用最有效率的PHP返回数组方式定义，支持惯例配置、公共配置、模块配置、调试配置和动态配置。

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，你还可以增加动态配置文件。

系统的配置参数是通过静态变量全局存取的，存取方式简单高效。

配置格式

PHP数组定义

ThinkPHP框架中默认所有配置文件的定义格式均采用返回PHP数组的方式，格式为：

```
//项目配置文件
return array(
    'DEFAULT_MODULE'     => 'Index', //默认模块
    'URL_MODEL'          => '2', //URL模式
    'SESSION_AUTO_START' => true, //是否开启session
    //更多配置参数
    //...
);
```

配置参数不区分大小写（因为无论大小写定义都会转换成小写），所以下面的配置等效：

```
//项目配置文件
return array(
    'default_module'     => 'Index', //默认模块
    'url_model'          => '2', //URL模式
    'session_auto_start' => true, //是否开启session
    //更多配置参数
    //...
);
```

但是我们建议保持大写定义配置参数的规范。

还可以在配置文件中可以使用二维数组来配置更多的信息，例如：

```
//项目配置文件
return array(
    'DEFAULT_MODULE'     => 'Index', //默认模块
    'URL_MODEL'          => '2', //URL模式
    'SESSION_AUTO_START' => true, //是否开启session
    'USER_CONFIG'        => array(
        'USER_AUTH' => true,
        'USER_TYPE' => 2,
    ),
    //更多配置参数
    //...
);
```

> 需要注意的是，二级参数配置区分大小写，也就说读取确保和定义一致。

其他配置格式支持

也可以采用 yaml/json/xml/ini 以及自定义格式的配置文件支持。

我们可以在应用入口文件中定义应用的配置文件的后缀，例如：

```
define('CONF_EXT','.ini');
```

定义后，应用的配置文件（包括模块的配置文件）后缀都统一采用.ini。

> 无论是什么格式的配置文件，最终都会解析成数组格式。
> 该配置不会影响框架内部的配置文件加载。

ini格式配置示例：

```
DEFAULT_MODULE=Index ;默认模块
URL_MODEL=2 ;URL模式
SESSION_AUTO_START=on ;是否开启session
```

xml格式配置示例：

```
<config>
<default_module>Index</default_module>
<url_model>2</url_model>
<session_auto_start>1</session_auto_start>
</config>
```

yaml格式配置示例：

```
default_module:Index #默认模块
url_model:2 #URL模式
session_auto_start:True #是否开启session
```

json格式配置示例：

```
{
"default_module":"Index",
"url_model":2,
"session_auto_start":True
}
```

除了 yaml/json/xml/ini 格式之外，我们还可以自定义配置格式，定义如下：

```
define('CONF_EXT','.test'); // 配置自定义配置格式（后缀）
define('CONF_PARSE','parse_test'); // 对应的解析函数
```

假设我们的自定义配置格式是类似 var1=val1&var2=val2 之类的字符串，那么parse_test定义如下：

```
function parse_test($str){
    parse_str($str,$config);
    return (array)$config;
}
```

> CONF_PARSE定义的解析函数返回值必须是一个PHP索引数组。

配置加载

在ThinkPHP中，一般来说应用的配置文件是自动加载的，加载的顺序是：

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

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

下面说明下不同的配置文件的区别和位置：

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。

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

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

模式配置（可选）

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（后面会有描述）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）。

> 模式配置文件是可选的

调试配置（可选）

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）

状态配置（可选）

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）。

> 状态配置文件是可选的

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）。

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

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

读取配置

无论何种配置文件，定义了配置文件之后，都统一使用系统提供的C方法（可以借助Config单词来帮助记忆）来读取已有的配置。

用法：

C('参数名称')

例如，读取当前的URL模式配置参数：

```
$model = C('URL_MODEL');
// 由于配置参数不区分大小写，因此下面的写法是等效的
// $model = C('url_model');
```

但是建议使用大写方式的规范。

> 注意：配置参数名称中不能含有 “.” 和特殊字符，允许字母、数字和下划线。

如果`url_model`尚未存在设置，则返回NULL。

支持在读取的时候设置默认值，例如：

```
// 如果my_config尚未设置的话，则返回default_config字符串
C('my_config',null,'default_config');
```

C方法也可以用于读取二维配置：

```
//获取用户配置中的用户类型设置
C('USER_CONFIG.USER_TYPE');
```

因为配置参数是全局有效的，因此C方法可以在任何地方读取任何配置，即使某个设置参数已经生效过期了。

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。

设置格式：

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);
```

扩展配置

扩展配置可以支持自动加载额外的自定义配置文件，并且配置格式和项目配置一样。

设置扩展配置的方式如下（多个文件用逗号分隔）：

```
// 加载扩展配置文件
'LOAD_EXT_CONFIG' => 'user,db',
```

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

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

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

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

```
<?php
//用户配置文件
return array(
    'USER_TYPE'      => 2, //用户类型
    'USER_AUTH_ID'   => 10, //用户认证ID
    'USER_AUTH_TYPE' => 2, //用户认证模式
);
```

那么，最终获取用户参数的方式是：

```
C('USER_AUTH_ID');
```

如果配置文件改成：

```
// 加载扩展配置文件
'LOAD_EXT_CONFIG' => array('USER'=>'user','DB'=>'db'),
```

则最终获取用户参数的方式改成：

```
C('USER.USER_AUTH_ID');
```

批量配置

C配置方法支持批量配置，例如：

```
$config = array('WEB_SITE_TITLE'=>'ThinkPHP','WEB_SITE_DESCRIPTION'=>'开源PHP框架');
C($config);
```

$config数组中的配置参数会合并到现有的全局配置中。

我们可以通过这种方式读取数据库中的配置参数，例如：

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

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

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

架构

模块化设计

一个完整的ThinkPHP应用基于模块/控制器/操作设计，并且，如果有需要的话，可以支持多入口文件和多级控制器。

ThinkPHP新版采用模块化的架构设计思想，对目录结构规范做了调整，可以支持多模块应用的创建，让应用的扩展更加方便。

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

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

ThinkPHP的应用可以支持切换到命令行访问，如果切换到命令行模式下面的访问规则是：

```
>php.exe index.php(或其它应用入口文件） 模块/控制器/操作/[参数名/参数值...]
```

解释下其中的几个概念：

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。

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

模块设计

新版采用模块化的设计架构，下面是一个应用目录下面的模块目录结构，每个模块可以方便的卸载和部署，并且支持公共模块。

```
Application      默认应用目录（可以设置）
├─Common         公共模块（不能直接访问）
├─Home           前台模块
├─Admin          后台模块
├─...            其他更多模块
├─Runtime        默认运行时目录（可以设置）
```

> 默认情况下，只要应用目录下面存在模块目录，该模块就可以访问，只有当你希望禁止某些模块或者仅允许模块访问的时候才需要进行模块列表的相关设置。

每个模块是相对独立的，其目录结构如下：

```
├─Module         模块目录
│  ├─Conf        配置文件目录
│  ├─Common      公共函数目录
│  ├─Controller  控制器目录
│  ├─Model       模型目录
│  ├─Logic       逻辑目录（可选）
│  ├─Service     Service目录（可选）
│  ... 更多分层目录可选
│  └─View        视图目录
```

> 由于采用多层的MVC机制，除了Conf和Common目录外，每个模块下面的目录结构可以根据需要灵活设置和添加，所以并不拘泥于上面展现的目录

公共模块

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

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

```
define('COMMON_PATH','./Common/');
define('APP_PATH','./Application/');
require './ThinkPHP/ThinkPHP.php';
```

其应用目录结构变成：

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

> 定义之后，Application目录下面就不再需要Common目录了。

自动生成模块目录

可以支持自动生成默认模块之外的模块目录以及批量生成控制器和模型类。

例如，如果我们需要生成一个Admin模块用于后台应用，在应用入口文件中定义如下：

```
// 绑定Admin模块到当前入口文件
define('BIND_MODULE','Admin');
define('APP_PATH','./Application/');
require './ThinkPHP/ThinkPHP.php';
```

然后访问URL地址

```
http://serverName/index.php
```

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

```
// 绑定Admin模块到当前入口文件
define('BIND_MODULE','Admin');
define('BUILD_CONTROLLER_LIST','Index,User,Menu');
define('APP_PATH','./Application/');
require './ThinkPHP/ThinkPHP.php';
```

访问后会自动生成三个指定的控制器类：

```
Admin\Controller\IndexController
Admin\Controller\UserController
Admin\Controller\MenuController
```

> 注意：默认生成的控制器类都是继承 `Think\Controller` ，如果需要继承其他的公共类需要另外调整。如果在应用的公共配置文件中设置关闭了 `APP_USE_NAMESPACE` 的话，生成的控制器类则不会采用命名空间定义。

同样，也可以定义 BUILD_MODEL_LIST 支持生成多个模型类：

```
// 绑定Admin模块到当前入口文件
define('BIND_MODULE','Admin');
define('BUILD_MODEL_LIST','User,Menu');
define('APP_PATH','./Application/');
require './ThinkPHP/ThinkPHP.php';
```

访问会自动生成模型类：

```
Admin\Model\UserModel
Admin\Model\MenuModel
```

> 注意：默认生成的模型类都是继承 Think\Model ，如果需要继承公共的模型类需要另外调整。 如果在应用的公共配置文件中设置关闭了 APP_USE_NAMESPACE 的话，生成的模型类则不会采用命名空间定义。

还可以自己手动调用 Think\Build 类的方法来生成控制器类和模型类，例如：

```
// 生成Admin模块的Role控制器类
// 默认类库为Admin\Controller\RoleController
// 如果已经存在则不会重新生成
\Think\Build::buildController('Admin','Role');
// 生成Admin模块的Role模型类
// 默认类库为Admin\Model\RoleModel
// 如果已经存在则不会重新生成
\Think\Build::buildModel('Admin','Role');
```

更多的方法可以参考Think\Build类库。

禁止访问模块

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

```
// 设置禁止访问的模块列表
'MODULE_DENY_LIST'      =>  array('Common','Runtime','Api'),
```

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

设置访问列表

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

```
'MODULE_ALLOW_LIST'    =>    array('Home','Admin','User'),
'DEFAULT_MODULE'       =>    'Home',
```

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

单模块设计

如果你的应用够简单，那么也许仅仅用一个模块就可以完成，那么可以直接设置：

```
// 关闭多模块访问
'MULTI_MODULE'          =>  false,
'DEFAULT_MODULE'        =>  'Home',
```

一旦关闭多模块访问后，就只能访问默认模块（这里设置的是Home）。

> 单模块设计后公共模块依然有效

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 `admin.php` 入口文件，并绑定Admin模块：

```
// 绑定Home模块到当前入口文件
define('BIND_MODULE','Admin');
define('APP_PATH','./Application/');
require './ThinkPHP/ThinkPHP.php';
```

> 如果你更改了系统默认的变量设置，则需要做对应的模块绑定的变量调整。

绑定模块后，原来的访问地址

```
http://serverName/index.php/Admin/Index/index
```

就变成

```
http://serverName/admin.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：

```
define('BIND_MODULE', 'Home'); // 绑定Home模块到当前入口文件
define('BIND_CONTROLLER','Index'); // 绑定Index控制器到当前入口文件
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
```

> 不同的入口文件还可以用于绑定不同的应用模式，参考应用模式部分。

URL模式

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

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

这是3.2版本的标准URL格式。

> 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。

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。

URL模式

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。

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

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

PATHINFO模式下面，URL是可定制的，例如，通过下面的配置：

```
// 更改PATHINFO参数分隔符
'URL_PATHINFO_DEPR'=>'-',
```

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

REWRITE模式

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`

> 更多环境的URL重写支持参考部署部分的URL重写。

兼容模式

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

可以更改兼容模式变量的名称定义，例如：

```
'VAR_PATHINFO'          =>  'path'
```

PATHINFO参数分隔符对兼容模式依然有效，例如：

```
// 更改PATHINFO参数分隔符
'URL_PATHINFO_DEPR'=>'-',
```

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

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

例如，我们在Apache下面的话，.htaccess文件改成如下内容：

```
<IfModule mod_rewrite.c>
 RewriteEngine on
 RewriteCond %{REQUEST_FILENAME} !-d
 RewriteCond %{REQUEST_FILENAME} !-f
 RewriteRule ^(.*)$ index.php?s=/$1 [QSA,PT,L]
</IfModule>
```

就可以和REWRITE模式一样访问下面的URL地址访问了：
`http://localhost/home/user/login/var/value`

多层MVC

ThinkPHP基于MVC（Model-View-Controller，模型-视图-控制器）模式，并且均支持多层（multi-Layer）设计。

模型（Model）层

默认的模型层由Model类构成，但是随着项目的增大和业务体系的复杂化，单一的模型层很难解决要求，ThinkPHP支持多层Model，设计思路很简单，不同的模型层仍然都继承自系统的Model类，但是在目录结构和命名规范上做了区分。

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

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

而这三个模型操作类统一都继承Model类即可，例如：

数据层：Home/Model/UserModel.class.php

```
namespace Home\Model;
use Think\Model;
class UserModel extends Model{
}
```

逻辑层：Home/Logic/UserLogic.class.php

```
namespace Home\Logic;
use Think\Model;
class UserLogic extends Model{
}
```

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

```
namespace Home\Service;
use Think\Model;
class UserService extends Model{
}
```

这样区分不同的模型层之后对用户数据的操作就非常清晰，在调用的时候，我们也可以用内置的D方法很方便的调用：

```
D('User') //实例化UserModel
D('User','Logic') //实例化UserLogic
D('User','Service') //实例化UserService
```

默认的模型层是Model，我们也可以更改设置，例如：

```
'DEFAULT_M_LAYER'      =>  'Logic', // 更改默认的模型层名称为Logic
```

更改之后，实例化的时候需要改成：

```
D('User') //实例化UserLogic
D('User','Model') //实例化UserModel
D('User','Service') //实例化UserService
```

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

视图（View）层

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

```
View/default/User/add.html
View/blue/User/add.html
```

复杂一点的多层视图还可以更进一步，采用不同的视图目录来完成，例如：

```
view 普通视图层目录
mobile 手机端访问视图层目录
```

这样做的好处是每个不同的视图层都可以支持不同的模板主题功能。

默认的视图层是View目录，我们可以调整设置如下：

```
'DEFAULT_V_LAYER'      =>  'Mobile', // 默认的视图层名称更改为Mobile
```

> 非默认视图层目录的模板获取需要使用T函数，后面会讲到。

控制器（Controller）层

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

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

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

```
namespace Home\Controller;
use Think\Controller;
class UserController extends Controller{
}
```

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

```
namespace Home\Event;
use Think\Controller;
class UserEvent extends Controller{
}
```

UserController负责外部交互响应，通过URL请求响应，例如 `http://serverName/User/index` ,而

UserEvent负责内部的事件响应，并且只能在内部调用：

```
A('User','Event');
```

默认的访问控制器层是Controller，我们可以调整设置如下：

```
'DEFAULT_C_LAYER'      =>  'Event', // 默认的控制器层名称改为Event
```

所以是和外部隔离的。

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

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

CBD模式

ThinkPHP引入了全新的CBD（核心Core+行为Behavior+驱动Driver）架构模式，从底层开始，框架就采用核心+行为+驱动的架构体系，核心保留了最关键的部分，并在重要位置设置了标签用以标记，其他功能都采用行为扩展和驱动的方式组合，开发人员可以根据自己的需要，对某个标签位置进行行为扩展或者替换，就可以方便的定制框架底层，也可以在应用层添加自己的标签位置和添加应用行为。而标签位置类似于AOP概念中的“切面”，行为都是围绕这个“切面”来进行编程。

Core（核心）

ThinkPHP的核心部分包括核心函数库、惯例配置、核心类库（包括基础类和内置驱动及核心行为），这些是ThinkPHP必不可少的部分。

```
ThinkPHP/Common/functions.php // 核心函数库
ThinkPHP/Conf/convention.php  // 惯例配置文件
ThinkPHP/Conf/debug.php  // 惯例调试配置文件
ThinkPHP/Mode/common.php  // 普通模式定义文件
ThinkPHP/Library/Think // 核心类库包
ThinkPHP/Library/Behavior // 系统行为类库
ThinkPHP/Library/Think/App.class.php // 核心应用类
ThinkPHP/Library/Think/Cache.class.php // 核心缓存类
ThinkPHP/Library/Think/Controller.class.php // 基础控制器类
ThinkPHP/Library/Think/Db.class.php // 数据库操作类
ThinkPHP/Library/Think/Dispatcher.class.php // URL解析调度类
ThinkPHP/Library/Think/Exception.class.php // 系统基础异常类
ThinkPHP/Library/Think/Hook.class.php // 系统钩子类
ThinkPHP/Library/Think/Log.class.php // 系统日志记录类
ThinkPHP/Library/Think/Model.class.php // 系统基础模型类
ThinkPHP/Library/Think/Route.class.php // 系统路由类
ThinkPHP/Library/Think/Storage.class.php // 系统存储类
ThinkPHP/Library/Think/Template.class.php // 内置模板引擎类
ThinkPHP/Library/Think/Think.class.php // 系统引导类
ThinkPHP/Library/Think/View.class.php // 系统视图类
```

Behavior目录下面是系统内置的一些行为类库，内置驱动则分布在各个不同的驱动目录下面（参考下面的驱动部分）。

Driver（驱动）

3.2版本在架构设计上更加强化了驱动的设计，替代了之前的引擎和模式扩展，并且改进了行为的设计，使得框架整体更加灵活，并且由于在需要写入数据的功能类库中都采用了驱动化的设计思想，所以使得新的框架能够轻松满足分布式部署的需求，对云平台的支持可以更简单的实现了。因此，在新版的扩展里面，已经取消了引擎扩展和模式扩展，改成配置不同的应用模式即可。

驱动包括

```
ThinkPHP/Library/Think/Cache/Driver // 缓存驱动类库
ThinkPHP/Library/Think/Db/Driver // 数据库驱动类库
ThinkPHP/Library/Think/Log/Driver // 日志记录驱动类库
ThinkPHP/Library/Think/Session/Driver // Session驱动类库
ThinkPHP/Library/Think/Storage/Driver // 存储驱动类库
ThinkPHP/Library/Think/Template/Driver // 第三方模板引擎驱动类库
ThinkPHP/Library/Think/Template/TagLib // 内置模板引擎标签库扩展类库
```

Behavior（行为）

行为（Behavior）是ThinkPHP扩展机制中比较关键的一项扩展，行为既可以独立调用，也可以绑定到某个标签（位）中进行侦听。这里的行为指的是一个比较抽象的概念，你可以想象成在应用执行过程中的一个动作或者处理，在框架的执行流程中，各个位置都可以有行为产生，例如路由检测是一个行为，静态缓存是一个行为，用户权限检测也是行为，大到业务逻辑，小到浏览器检测、多语言检测等等都可以当做是

一个行为，甚至说你希望给你的网站用户的第一次访问弹出Hello，world！这些都可以看成是一种行为，行为的存在让你无需改动框架和应用，而在外围通过扩展或者配置来改变或者增加一些功能。

而不同的行为之间也具有位置共同性，比如，有些行为的作用位置都是在应用执行前，有些行为都是在模板输出之后，我们把这些行为发生作用的位置称之为标签（位），也可以称之为钩子，当应用程序运行到这个标签的时候，就会被拦截下来，统一执行相关的行为，类似于AOP编程中的“切面”的概念，给某一个标签绑定相关行为就成了一种类AOP编程的思想。

系统标签位

系统核心提供的标签位置包括（按照执行顺序排列）：

- app_init 应用初始化标签位
- module_check 模块检测标签位（3.2.1版本新增）
- path_info PATH_INFO检测标签位
- app_begin 应用开始标签位
- action_name 操作方法名标签位
- action_begin 控制器开始标签位
- view_begin 视图输出开始标签位
- view_template 视图模板解析标签位
- view_parse 视图解析标签位
- template_filter 模板解析过滤标签位
- view_filter 视图输出过滤标签位
- view_end 视图输出结束标签位
- action_end 控制器结束标签位
- app_end 应用结束标签位

在每个标签位置，可以配置多个行为，行为的执行顺序按照定义的顺序依次执行。除非前面的行为里面中断执行了（某些行为可能需要中断执行，例如检测机器人或者非法执行行为），否则会继续下一个行为的执行。

除了这些系统内置标签之外，开发人员还可以在应用中添加自己的应用标签，在任何需要拦截的位置添加如下代码即可：

```
// 添加my_tag 标签侦听
\Think\Hook::listen('my_tag');
```

方法第一个参数是要侦听的标签位，除此之外还可以传入并且只接受一个参数，如果需要传入多个参数，请使用数组。

```
// 添加my_tag 标签侦听
\Think\Hook::listen('my_tag',$params);
```

该参数为引用传值，所以只能传入变量，因此下面的传值是错误的：

```
// 添加my_tag 标签侦听
\Think\Hook::listen('my_tag','param');
```

核心行为

系统的很多核心功能也是采用行为扩展组装的，对于满足项目日益纷繁复杂的需求和定制底层框架提供了更多的方便和可能性。

核心行为位于 `ThinkPHP/Behavior/` 目录下面，框架核心内置的行为包括如下：

行为名称	说明	对应标签位置
BuildLite	生成Lite文件（3.2.1版本新增）	app_init
ParseTemplate	模板文件解析，并支持第三方模板引擎驱动	view_parse
ShowPageTrace	页面Trace功能行为，完成页面Trace功能	view_end
ShowRuntime	运行时间显示行为，完成运行时间显示	view_filter
TokenBuild	令牌生成行为，完成表单令牌的自动生成	view_filter
ReadHtmlCache	读取静态缓存行为	app_init
WriteHtmlCache	生成静态缓存行为	view_filter

行为定义

自定义的扩展行为可以放在核心或者应用目录，只要遵循命名空间的定义规则即可。

行为类的命名采用：

> 行为名称（驼峰法，首字母大写）+Behavior

行为类的定义方式如下：

```
namespace Home\Behavior;
class TestBehavior {
    // 行为扩展的执行入口必须是run
    public function run(&$params){
        if(C('TEST_PARAM')) {
            echo 'RUNTEST BEHAVIOR '.$params;
        }
    }
}
```

行为类必须定义执行入口方法 `run` ,由于行为的调用机制影响，run方法不需要任何返回值，所有返回都通过引用返回。

> run方法的参数只允许一个，但可以传入数组。

行为绑定

行为定义完成后，就需要绑定到某个标签位置才能生效，否则是不会执行的。

我们需要在应用的行为定义文件 tags.php 文件中进行行为和标签的位置定义，格式如下：

```
return array(
    '标签名称1'=>array('行为名1','行为名2',...),
    '标签名称2'=>array('行为名1','行为名2',...),
 );
```

标签名称包括我们前面列出的系统标签和应用中自己定义的标签名称，比如你需要在app_init标签位置定义一个 CheckLangBehavior 行为类的话，可以使用：

```
return array(
  'app_init'=>array('Home\Behavior\CheckLangBehavior'),
);
```

可以给一个标签位定义多个行为，行为的执行顺序就是定义的先后顺序，例如：

```
return array(
  'app_init'=>array(
    'Home\Behavior\CheckLangBehavior',
    'Home\Behavior\CronRunBehavior'
   ),
);
```

默认情况下tags.php中定义的行为会并入系统行为一起执行，也就是说如果系统的行为定义中app_init标签中已经定义了其他行为，则会首先执行系统行为扩展中定义的行为，然后再执行项目行为中定义的行为。例如： 系统行为定义文件中定义了：

```
'app_begin'  =>  array(
    'Behavior\ReadHtmlCacheBehavior', // 读取静态缓存
),
```

而应用行为定义文件有定义：

```
'app_begin'  =>  array(
    'Home\Behavior\CheckModuleBehavior',
    'Home\Behavior\CheckLangBehavior',
),
```

则最终执行到app_begin标签（位）的时候，会依次执行：

```
Library\Behavior\ReadHtmlCacheBehavior
Home\Behavior\CheckModuleBehavior
Home\Behavior\CheckLangBehavior
```

三个行为（除非中间某个行为有中止执行的操作）。

如果希望应用的行为配置文件中的定义覆盖系统的行为定义，可以改为为如下方式：

```
'app_begin'  =>  array(
    'Home\Behavior\CheckModuleBehavior',
    'Home\Behavior\CheckLangBehavior',
    '_overlay'    =>    true,
),
```

则最终执行到app_begin标签（位）的时候，会依次执行下面两个行为：

```
Home\Behavior\CheckModuleBehavior
Home\Behavior\CheckLangBehavior
```

应用行为的定义没有限制，你可以把一个行为绑定到多个标签位置执行，例如：

```
return array(
    'app_begin'=>array('Home\Behavior\TestBehavior'), // 在app_begin 标签位添加Test行为
    'app_end'=>array('Home\Behavior\TestBehavior'), // 在app_end 标签位添加Test行为
 );
```

单独执行

行为的调用不一定要放到标签才能调用，如果需要的话，我们可以在控制器中或者其他地方直接调用行为。例如，我们可以把用户权限检测封装成一个行为类，例如：

```
namespace Home\Behavior;
use Think\Behavior;
class AuthCheckBehavior extends Behavior {

    // 行为扩展的执行入口必须是run
    public function run(&$return){
        if(C('USER_AUTH_ON')) {
            // 进行权限认证逻辑 如果认证通过 $return = true;
            // 否则用halt输出错误信息
        }
    }
}
```

定义了AuthCheck行为后，我们可以在控制器的_initialize方法中直接用下面的方式调用：

```
B('Home\Behavior\AuthCheck');
```

命名空间

3.2版本全面采用命名空间方式定义和加载类库文件，有效的解决多个模块之间的冲突问题，并且实现了更加高效的类库自动加载机制。

> 命名空间的概念必须了解，否则会成为学习3.2版本开发的重大障碍。
> 如果不清楚什么是命名空间，可以参考PHP手册：PHP命名空间

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

```
namespace Org\Util;
class File {
}
```

其所在的路径是 `ThinkPHP/Library/Org/Util/File.class.php` ，因此，如果我们实例化该类的话：

```
$class = new \Org\Util\File();
```

系统会自动加载 `ThinkPHP/Library/Org/Util/File.class.php` 文件。

根命名空间

根命名空间是一个关键的概念，以上面的 `Org\Util\File` 类为例， `Org` 就是一个根命名空间，其对应的初始命名空间目录就是系统的类库目录（ `ThinkPHP/Library` ），Library目录下面的子目录会自动识别为根命名空间，这些命名空间无需注册即可使用。

例如，我们在Library目录下面新增一个My根命名空间目录，然后定义一个Test类如下：

```
namespace My;
class Test {
    public function sayHello(){
        echo 'hello';
    }
}
```

Test类保存在 `ThinkPHP/Library/My/Test.class.php` ，我们就可以直接实例化和调用：

```
$Test = new \My\Test();
$Test->sayHello();
```

模块中的类库命名空间的根都是以模块名命名，例如：

```
namespace Home\Model;
class UserModel extends \Think\Model {
}
```

其类文件位于 `Application/Home/Model/UserModel.class.php` 。

```
namespace Admin\Event;
class UserEvent {
}
```

其类文件位于 `Application/Admin/Event/UserEvent.class.php` 。

> 特别注意：如果你需要在3.2版本中实例化PHP内置的类库或者第三方的没有使用命名空间定义的类，需要采用下面的方式：

```
// 必须从根命名空间调用系统内置的类库或者第三方没有使用命名空间的类库
$class =    new \stdClass();
$sxml  =    new \SimpleXmlElement($xmlstr);
```

自动加载

在3.2中，基本上无需手动加载类库文件，你可以很方便的完成自动加载。

命名空间自动加载

系统可以通过类的命名空间自动定位到类库文件，例如：

我们定义了一个类 `\Org\Util\Auth` 类：

```
namespace Org\Util;
class Auth {
}
```

保存到 `ThinkPHP/Library/Org/Util/Auth.class.php` 。

接下来，我们就可以直接实例化了。

```
new \Org\Util\Auth();
```

在实例化 \Org\Util\Auth 类的时候，系统会自动加载 ThinkPHP/Library/Org/Util/Auth.class.php 文件。

框架的Library目录下面的命名空间都可以自动识别和定位，例如：

```
├─Library      框架类库目录
│  ├─Think     核心Think类库包目录
│  ├─Org       Org类库包目录
│  ├─ ...      更多类库目录
```

Library目录下面的子目录都是一个根命名空间，也就是说以Think、Org为根命名空间的类都可以自动加载：

```
new \Think\Cache\Driver\File();
new \Org\Util\Auth();
new \Org\Io\File();
```

都可以自动加载对应的类库文件。

> 你可以在Library目录下面任意增加新的目录，就会自动注册成为一个新的根命名空间。

注册新的命名空间

除了Library目录下面的命名空间之外，我们还可以注册其他的根命名空间，例如：

```
'AUTOLOAD_NAMESPACE' => array(
    'My'      => THINK_PATH.'My',
    'One'     => THINK_PATH.'One',
)
```

配置了上面的 AUTOLOAD_NAMESPACE 后，如果我们实例化下面的类库

```
new \My\Net\IpLocation();
new \One\Util\Log();
```

会自动加载对应的类库文件

```
ThinkPHP/My/Net/IpLocation.class.php
ThinkPHP/One/Util/Log.class.php
```

如果命名空间不在Library目录下面，并且没有定义对应的 AUTOLOAD_NAMESPACE 参数的话，则会当

作模块的命名空间进行自动加载，例如：

```
new \Home\Model\UserModel();
new \Home\Event\UserEvent();
```

由于ThinkPHP/Library目录下面不存在Home目录，也没在 AUTOLOAD_NAMESPACE 参数定义Home命名空间，所以就把Home当成模块命名空间来识别，所以会自动加载：

```
Application/Home/Model/UserModel.class.php
Application/Home/Event/UserEvent.class.php
```

> 注意：命名空间的大小写需要和目录名的大小写对应，否则可能会自动加载失败。

类库映射

遵循我们上面的命名空间定义规范的话，基本上可以完成类库的自动加载了，但是如果定义了较多的命名空间的话，效率会有所下降，所以，我们可以给常用的类库定义类库映射。

命名类库映射相当于给类文件定义了一个别名，效率会比命名空间定位更高效，例如：

```
Think\Think::addMap('Think\Log',THINK_PATH.'Think\Log.php');
Think\Think::addMap('Org\Util\Array',THINK_PATH.'Org\Util\Array.php');
```

> 注意：添加类库映射的时候不需要写类库开头的"\"

也可以利用addMap方法批量导入类库映射定义，例如：

```
$map = array('Think\Log'=>THINK_PATH.'Think\Log.php','Org\Util\Array'=>THINK_PATH.'Org\Util\Ar
ray.php');
Think\Think::addMap($map);
```

当然，比较方便的方式是我们可以在模块配置目录下面创建alias.php文件用于定义类库映射，该文件会自动加载，定义方式如下：

```
return array(
    'Think\Log'         =>    THINK_PATH.'Think\Log.php',
    'Org\Util\Array'    =>    THINK_PATH.'Org\Util\Array.php'
);
```

自动加载的优先级

在实际的应用类库加载过程中，往往会涉及到自动加载的优先级问题，以 Test\MyClass 类为例，自动加

载的优先顺序如下：

1. 判断是否有注册了Test\MyClass类库映射，如果有则自动加载类库映射定义的文件；
2. 判断是否存在Library/Test目录，有则以该目录为初始目录加载；
3. 判断是否有注册Test根命名空间，有则以注册的目录为初始目录加载；
4. 如果以上都不成立，则以Test为模块目录进行初始目录加载；

然后以上面获取到的初始目录加载命名空间对应路径的文件；

手动加载第三方类库

如果要加载第三方类库，包括不符合命名规范和后缀的类库，以及没有使用命名空间或者命名空间和路径不一致的类库，或者你就是想手动加载类库文件，我们都可以通过手动导入的方式加载。

我们可以使用import方法导入任何类库，用法如下：

```
// 导入Org类库包 Library/Org/Util/Date.class.php类库
import("Org.Util.Date");
// 导入Home模块下面的 Application/Home/Util/UserUtil.class.php类库
import("Home.Util.UserUtil");
// 导入当前模块下面的类库
import("@.Util.Array");
// 导入Vendor类库包 Library/Vendor/Zend/Server.class.php
import('Vendor.Zend.Server');
```

对于import方法，系统会自动识别导入类库文件的位置，ThinkPHP可以自动识别的类库包包括Think、Org、Com、Behavior和Vendor包，以及Library目录下面的子目录，如果你在Library目录下面创建了一个Test子目录，并且创建了一个UserTest.class.php类库，那么可以这样导入：

```
import('Test.UserTest');
```

其他的就认为是应用类库导入。

注意，如果你的类库没有使用命名空间定义的话，实例化的时候需要加上根命名空间，例如：

```
import('Test.UserTest');
$test = new \UserTest();
```

按照系统的规则，import方法是无法导入具有点号的类库文件的，因为点号会直接转化成斜线，例如我们定义了一个名称为User.Info.class.php 的文件的话，采用：

```
import("Org.User.Info");
```

方式加载的话就会出现错误，导致加载的文件不是Org/User.Info.class.php 文件，而是

Org/User/Info.class.php 文件，这种情况下，我们可以使用：

```
import("Org.User#Info");
```

来导入。

大多数情况下，import方法都能够自动识别导入类库文件的位置，如果是特殊情况的导入，需要指定import方法的第二个参数作为起始导入路径。例如，要导入当前文件所在目录下面的RBAC/AccessDecisionManager.class.php 文件，可以使用：

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

如果你要导入的类库文件名的后缀不是class.php而是php，那么可以使用import方法的第三个参数指定后缀：

```
import("RBAC.AccessDecisionManager",dirname(__FILE__),".php");
```

> 注意：在Unix或者Linux主机下面是区别大小写的，所以在使用import方法的时候要注意目录名和类库名称的大小写，否则会导入失败。

如果你的第三方类库都放在Vendor目录下面，并且都以.php为类文件后缀，也没用采用命名空间的话，那么可以使用系统内置的Vendor函数简化导入。 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 方法导入只需要使用：

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

就可以导入Dir类库了。

Vendor方法也可以支持和import方法一样的基础路径和文件名后缀参数，例如：

```
Vendor('Zend.Filter.Dir',dirname(__FILE__),'.class.php');
```

应用模式

ThinkPHP支持应用模式定义，每个应用模式有自己的定义文件，用于配置当前模式需要加载的核心文件和配置文件，以及别名定义、行为扩展定义等等。除了模式定义外，应用自身也可以独立定义自己的模式文件。

如果应用模式涉及到不同的存储类型，例如采用分布式存储等，就需要另外设置存储类型（STORAGE_TYPE）。不同的存储类型由Think\Storage类及相关驱动进行支持。

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

```
// 定义存储类型和应用模式为SAE（用于支持SAE平台）
define('STORAGE_TYPE','sae');
define('APP_MODE','sae');
define('APP_PATH','./Application/');
require './ThinkPHP/ThinkPHP.php';
```

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

> 以上代码只是示范，其实SAE平台不需要显式定义，因为系统会自动识别SAE平台而采用sae模式。
> 每个入口文件仅能定义一个应用模式，所以，如果需要对相同的应用模块设置不同的应用模式访问，就可以通过增加入口文件的方式来解决。

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

```
// 应用配置文件
Application/Common/Conf/config_sae.php
// 模块配置文件
Application/Home/Conf/config_sae.php
```

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

项目编译

应用编译缓存

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

项目编译机制对运行没有任何影响，预编译机制只会执行一次，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。

编译缓存文件默认生成在应用目录的Runtime目录下面，我们可以在Application/Runtime目录下面看到有一个 common~runtime.php 文件，这个就是普通模式的编译缓存文件。如果你当前运行在其他的应用模式下面，那么编译缓存文件就是： 应用模式名~runtime.php

例如，如果你当前用的是SAE模式，那么生成的编译缓存文件则会变成 sae~runtime.php 。

> 普通模式的编译缓存的内容包括：系统函数库、系统基础核心类库、核心行为类库、项目函数文件，当然这些是可以改变的。

运行Lite文件

运行Lite文件的作用是替换框架的入口文件或者替换应用入口文件，提高运行效率。因为默认生成的文件名为lite.php，并且是运行时动态生成，因此称之为运行Lite文件。

Lite文件的特点包括：

- 运行时动态生成；
- 常量定义为针对当前环境；
- 支持定义需要编译的文件列表；
- 支持生成Lite文件的名称；

如何生成Lite文件，请参考部署部分的替换入口。

系统流程

ThinkPHP框架开发的应用的标准执行流程如下：

1. 用户URL请求
2. 调用应用入口文件（通常是网站的index.php）
3. 载入框架入口文件（ThinkPHP.php）
4. 记录初始运行时间和内存开销
5. 系统常量判断及定义
6. 载入框架引导类（Think\Think）并执行Think::start方法进行应用初始化
7. 设置错误处理机制和自动加载机制
8. 调用Think\Storage类进行存储初始化（由STORAGE_TYPE常量定义存储类型）
9. 部署模式下如果存在应用编译缓存文件则直接加载（直接跳转到步骤22）
10. 读取应用模式（由APP_MODE常量定义）的定义文件（以下以普通模式为例说明）
11. 加载当前应用模式定义的核心文件（普通模式是 ThinkPHP/Mode/common.php）
12. 加载惯例配置文件（普通模式是 ThinkPHP/Conf/convention.php）
13. 加载应用配置文件（普通模式是 Application/Common/Conf/config.php）
14. 加载系统别名定义
15. 判断并读取应用别名定义文件（普通模式是 Application/Common/Conf/alias.php）
16. 加载系统行为定义
17. 判断并读取应用行为定义文件（普通模式是 Application/Common/Conf/tags.php）

18. 加载框架底层语言包（普通模式是 ThinkPHP/Lang/zh-cn.php）
19. 如果是部署模式则生成应用编译缓存文件
20. 加载调试模式系统配置文件（ThinkPHP/Conf/debug.php）
21. 判断并读取应用的调试配置文件（默认是 Application/Common/Conf/debug.php）
22. 判断应用状态并读取状态配置文件（如果APP_STATUS常量定义不为空的话）
23. 检测应用目录结构并自动生成（如果CHECK_APP_DIR配置开启并且RUNTIME_PATH目录不存在的情况下）
24. 调用Think\App类的run方法启动应用
25. 应用初始化（app_init）标签位侦听并执行绑定行为
26. 判断并加载动态配置和函数文件
27. 调用Think\Dispatcher::dispatch方法进行URL请求调度
28. 自动识别兼容URL模式和命令行模式下面的$_SERVER['PATH_INFO']参数
29. 检测域名部署以及完成模块和控制器的绑定操作（APP_SUB_DOMAIN_DEPLOY参数开启）
30. 分析URL地址中的PATH_INFO信息
31. 获取请求的模块信息
32. 检测模块是否存在和允许访问
33. 判断并加载模块配置文件、别名定义、行为定义及函数文件
34. 判断并加载模块的动态配置和函数文件
35. 模块的URL模式判断
36. 模块的路由检测（URL_ROUTER_ON开启）
37. PATH_INFO处理（path_info）标签位侦听并执行绑定行为
38. URL后缀检测（URL_DENY_SUFFIX以及URL_HTML_SUFFIX处理）
39. 获取当前控制器和操作，以及URL其他参数
40. URL请求调度完成（url_dispatch）标签位侦听并执行绑定行为
41. 应用开始（app_begin）标签位侦听并执行绑定行为
42. 调用SESSION_OPTIONS配置参数进行Session初始化（如果不是命令行模式）
43. 根据请求执行控制器方法
44. 如果控制器不存在则检测空控制器是否存在
45. 控制器开始（action_begin）标签位侦听并执行绑定行为
46. 默认调用系统的ReadHtmlCache行为读取静态缓存（HTML_CACHE_ON参数开启）
47. 判断并调用控制器的_initialize初始化方法
48. 判断操作方法是否存在，如果不存在则检测是否定义空操作方法
49. 判断前置操作方法是否定义，有的话执行
50. Action参数绑定检测，自动匹配操作方法的参数
51. 如果有模版渲染（调用控制器display方法）
52. 视图开始（view_begin）标签位侦听并执行绑定行为
53. 调用Think\View的fetch方法解析并获取模版内容
54. 自动识别当前主题以及定位模版文件
55. 视图解析（view_parse）标签位侦听并执行绑定行为

56. 默认调用内置ParseTemplate行为解析模版（普通模式下面）
57. 模版引擎解析模版内容后生成模版缓存
58. 模版过滤替换（template_filter）标签位侦听并执行绑定行为
59. 默认调用系统的ContentReplace行为进行模版替换
60. 输出内容过滤（view_filter）标签位侦听并执行绑定行为
61. 默认调用系统的WriteHtmlCache行为写入静态缓存（HTML_CACHE_ON参数开启）
62. 调用Think\View类的render方法输出渲染内容
63. 视图结束（view_end）标签位侦听并执行绑定行为
64. 判断后置操作方法是否定义，有的话执行
65. 控制器结束（action_end）标签位侦听并执行绑定行为
66. 应用结束（app_end）标签位侦听并执行绑定行为
67. 执行系统的ShowPageTrace行为（SHOW_PAGE_TRACE参数开启并且不是AJAX请求）
68. 日志信息存储写入

如果你绑定了更多的应用行为的话，流程可能会更加复杂。

> 如果是部署模式下面的第二次请求的话，上面的流程中的步骤10~21是可以省略的。

路由

利用路由功能，可以让你的URL地址更加简洁和优雅。ThinkPHP支持对模块的URL地址进行路由操作（路由功能是针对PATHINFO模式或者兼容URL而设计的，暂时不支持普通URL模式）。

ThinkPHP的路由功能包括：

- 正则路由
- 规则路由
- 静态路由（URL映射）
- 闭包支持

路由定义

启用路由

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

```
// 开启路由
'URL_ROUTER_ON'   => true,
```

> 路由功能可以针对模块，也可以针对全局，针对模块的路由则需要在模块配置文件中开启和设置路由，如果是针对全局的路由，则是在公共模块的配置文件中开启和设置（后面我们以模块路由定义为例）。

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

```
'URL_ROUTE_RULES'=>array(
    'news/:year/:month/:day' => array('News/archive', 'status=1'),
    'news/:id'               => 'News/read',
    'news/read/:id'          => '/news/:1',
),
```

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

路由定义

路由规则的定义格式为： '路由表达式'=>'路由地址和传入参数'

或者：array('路由表达式','路由地址','传入参数')

> 模块路由和全局路由配置的区别在于，全局路由的路由地址必须包含模块。

路由表达式

路由表达式包括规则路由和正则路由的定义表达式，只能使用字符串。

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

> 详细的规则路由和正则路由表达式的定义方法参考后面的章节。

路由地址

路由地址（可以支持传入额外参数）表示前面的路由表达式需要路由到的地址（包括内部地址和外部地址），并且允许隐式传入URL里面没有的一些参数，这里允许使用字符串或者数组方式定义，特殊情况下还可以采用闭包函数定义路由功能，支持下面6种方式定义：

定义方式	定义格式
方式1：路由到内部地址（字符串）	'[控制器/操作]?额外参数1=值1&额外参数2=值2...'
方式2：路由到内部地址（数组）参数采用字符串方式	array('[控制器/操作]','额外参数1=值1&额外参数2=值2...')
方式3：路由到内部地址（数组）参数采用数组方式	array('[控制器/操作]',array('额外参数1'=>'值1','额外参数2'=>'值2'...)[,路由参数])
方式4：路由到外部地址（字符串）301重定向	'外部地址'
方式5：路由到外部地址（数组）可以指定重定向代码	array('外部地址','重定向代码'[,路由参数])
方式6：闭包函数	function($name){ echo 'Hello,'.$name;}

如果你定义的是全局路由（在公共模块的配置文件中定义），那么路由地址的定义格式中需要增加模块名，例如：

```
'blog/:id'=>'Home/blog/read' // 表示路由到Home模块的blog控制器的read操作方法
```

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

```
'blog/:id'=>'/blog/read/id/:1'
```

和

```
'blog/:id'=>'blog/read'
```

虽然都是路由到同一个地址，但是前者采用的是301重定向的方式路由跳转，这种方式的好处是URL可以比较随意（包括可以在URL里面传入更多的非标准格式的参数），而后者只是支持模块和操作地址。

举个例子，如果我们希望 `avatar/123` 重定向到 `/member/avatar/id/123_small` 的话，只能使用：

```
'avatar/:id'=>'/member/avatar/id/:1_small'
```

路由地址采用重定向地址的话，如果要引用动态变量，也是采用 `:1、:2` 的方式。

采用重定向到外部地址通常对网站改版后的URL迁移过程非常有用，例如：

```
'blog/:id'=>'http://blog.thinkphp.cn/read/:1'
```

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

默认情况下，外部地址的重定向采用301重定向，如果希望采用其它的，可以使用：

```
'blog/:id'=>array('http://blog.thinkphp.cn/read/:1',302);
```

在路由跳转的时候支持额外传入参数对（额外参数指的是不在URL里面的参数，隐式传入需要的操作中，有时候能够起到一定的安全防护作用，后面我们会提到），支持 `额外参数1=值1&额外参数2=值2` 或者 `array('额外参数1'=>'值1','额外参数2'=>'值2'...)` 这样的写法，可以参考不同的定义方式选择。例如：

```
'blog/:id'=>'blog/read?status=1&app_id=5',
'blog/:id'=>array('blog/read?status=1&app_id=5'),
'blog/:id'=>array('blog/read','status=1&app_id=5'),
'blog/:id'=>array('blog/read',array('status'=>1,'app_id'=>5)),
```

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

路由参数

当路由地址采用数组方式定义的时候，还可以传入额外的路由参数。

> 这些参数的作用是限制前面定义的路由规则的生效条件。

限制URL后缀

例如：

```
'blog/:id'=>array('blog/read','status=1&app_id=5',array('ext'=>'html')),
```

就可以限制html后缀访问该路由规则才能生效。

限制请求类型

例如：

```
'blog/:id'=>array('blog/read','status=1&app_id=5',array('method'=>'get')),
```

就限制了只有GET请求该路由规则才能生效。

自定义检测

支持自定义检测，例如： 例如：

```
'blog/:id'=>array('blog/read','status=1&app_id=5',array('callback'=>'checkFun')),
```

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

规则路由

规则路由是一种比较容易理解的路由定义方式，采用ThinkPHP设计的规则表达式来定义。

规则表达式

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

```
'my'          => 'Member/myinfo', // 静态地址路由
'blog/:id'    => 'Blog/read', // 静态地址和动态地址结合
'new/:year/:month/:day'=>'News/read', // 静态地址和动态地址结合
':user/:blog_id' =>'Blog/read',// 全动态地址
```

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

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

数字约束

支持对变量的类型检测，但仅仅支持数字类型的约束定义，例如

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

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

> 目前不支持长度约束，需要的话采用正则定义解决

函数支持

可以支持对路由变量的函数过滤，例如：

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

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

> 注意：不支持对变量使用多次函数处理和函数额外参数传入。

可选定义

支持对路由参数的可选定义，例如：

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

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

以上定义路由规则后，下面的URL访问地址都可以被正确的路由匹配：

```
http://serverName/index.php/Home/blog/2013
http://serverName/index.php/Home/blog/2013/12
```

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

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

规则排除

非数字变量支持简单的排除功能，主要是起到避免解析混淆的作用，例如：

```
'news/:cate^add-edit-delete'=>'News/category'
```

因为规则定义的局限性，恰巧我们的路由规则里面的news和实际的news模块是相同的命名，而`:cate`并不能自动区分当前URL里面的动态参数是实际的操作名还是路由变量，所以为了避免混淆，我们需要对路由变量cate进行一些排除以帮助我们进行更精确的路由匹配，格式`^add-edit-delete`表示，匹配除了add edit 和delete之外的所有字符串，我们建议更好的方式还是改进你的路由规则，避免路由规则和模块同名的情况存在，例如

```
'new/:cate'=>'News/category'
```

就可以更简单的定义路由规则了。

完全匹配

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

```
'new/:cate$'=> 'News/category'
```

`http://serverName/index.php/Home/new/info`

会匹配成功，而

`http://serverName/index.php/Home/new/info/2`

则不会匹配成功。

如果是采用

```
'new/:cate'=> 'News/category'
```

方式定义的话，则两种方式的URL访问都可以匹配成功。

> 完全匹配的路由规则中如果使用可选参数的话将会无效。

正则路由

正则路由也就是采用正则表达式定义路由的一种方式，依靠强大的正则表达式，能够定义更灵活的路由规则。

路由表达式支持的正则定义必须以“/”开头，否则就视为规则表达式。也就是说如果采用：

```
'#^blog\/(\d+)$#' => 'Blog/read/id/:1'
```

方式定义的正则表达式不会被支持，而会被认为是规则表达式进行解析，从而无法正确匹配。

下面是一种正确的正则路由定义：

```
'/^new\/(\d{4})\/(\d{2})$/' => 'News/achive?year=:1&month=:2',
```

> 对于正则表达式中的每个变量（即正则规则中的子模式）部分，如果需要在后面的路由地址中引用，可以采用:1、:2这样的方式，序号就是子模式的序号。

正则定义也支持函数过滤处理，例如：

```
'/^new\/(\d{4})\/(\d{2})$/' => 'News/achive?year=:1|format_year&month=:2',
```

其中 year=:1|format_year 就表示对匹配到的变量进行format_year函数处理（假设format_year是一个用户自定义函数）。

更多的关于如何定义正则表达式就不在本文的描述范畴了。

静态路由

静态路由其实属于规则路由的静态简化版（又称为URL映射），路由定义中不包含动态参数，静态路由不需要遍历路由规则而是直接定位，因此效率较高，但作用也有限。

如果我们定义了下面的静态路由

```
'URL_ROUTER_ON'   => true,
'URL_MAP_RULES'=>array(
   'new/top' => 'news/index?type=top'
)
```

> 注意：为了不影响动态路由的遍历效率，静态路由采用URL_MAP_RULES定义和动态路由区分开来

定义之后，如果我们访问： `http://serverName/Home/new/top`

其实是访问： `http://serverName/Home/news/index/type/top`

静态路由是完整匹配，所以如果访问： `http://serverName/Home/new/top/var/test`

尽管前面也有 new/top ，但并不会被匹配到 news/index/type/top 。

静态路由定义不受URL后缀影响，例如： http://serverName/Home/new/top.html 也可以正常访问。

> 静态路由的路由地址 只支持字符串，格式： [控制器/操作?]参数1=值1&参数2=值2

闭包支持

闭包定义

我们可以使用闭包的方式定义一些特殊需求的路由，而不需要执行控制器的操作方法了，例如：

```
'URL_ROUTE_RULES'=>array(
    'test'        =>
        function(){
            echo 'just test';
        },
    'hello/:name' =>
        function($name){
            echo 'Hello,'.$name;
        }
)
```

参数传递

闭包定义的参数传递在规则路由和正则路由的两种情况下有所区别。

规则路由

规则路由的参数传递比较简单：

```
'hello/:name' =>
    function($name){
        echo 'Hello,'.$name;
    }
```

规则路由中定义的动态变量的名称 就是闭包函数中的参数名称，不分次序。 因此，如果我们访问的URL地址是： http://serverName/Home/hello/thinkphp

则浏览器输出的结果是： Hello,thinkphp

如果多个参数可以使用：

```
'blog/:year/:month' =>
    function($year,$month){
        echo 'year='.$year.'&month='.$month;
    }
```

正则路由

如果是正则路由的话，闭包函数中的参数就以正则中出现的参数次序来传递，例如：

```
'/^new\/(\d{4})\/(\d{2})$/' =>
    function($year,$month){
        echo 'year='.$year.'&month='.$month;
    }
```

如果我们访问： `http://serverName/Home/new/2013/03` 浏览器输出结果是：
`year=2013&month=03`

继续执行

默认的情况下，使用闭包定义路由的话，一旦匹配到路由规则，执行完闭包方法之后，就会中止后续执行。如果希望闭包函数执行后，后续的程序继续执行，可以在闭包函数中使用布尔类型的返回值，例如：

```
'hello/:name' =>
    function($name){
        echo 'Hello,'.$name.'<br/>';
        $_SERVER['PATH_INFO'] = 'blog/read/name/'.$name;
        return false;
    }
```

该路由定义中的闭包函数首先执行了一段输出代码，然后重新设置了 `$_SERVER['PATH_INFO']` 变量，交给后续的程序继续执行，因为返回值是false，所以会继续执行控制器和操作的检测，从而会执行Blog控制器的read操作方法。

假设blog控制器中的read操作方法代码如下：

```
public function read($name){
    echo 'read,'.$name.'!<br/>';
}
```

如果我们访问的URL地址是： `http://serverName/Home/hello/thinkphp`

则浏览器输出的结果是：

```
Hello,thinkphp
read,thinkphp!
```

实例说明

我们已经了解了如何定义路由规则，下面我们来举个例子加深印象。

假设我们定义了News控制器如下（代码实现仅供参考）：

```
namespace Home\Controller;
use Think\Controller;
class NewsController extends Controller{
    public function read(){
        $New = M('New');
        if(isset($_GET['id'])) {
            // 根据id查询结果
            $data = $New->find($_GET['id']);
        }elseif(isset($_GET['name'])){
            // 根据name查询结果
            $data = $New->getByName($_GET['name']);
        }
        $this->data = $data;
        $this->display();
    }

    public function archive(){
        $New = M('New');
        $year  =  $_GET['year'];
        $month  =  $_GET['month'];
        $begin_time = strtotime($year . $month . "01");
        $end_time = strtotime("+1 month", $begin_time);
        $map['create_time'] =  array(array('gt',$begin_time),array('lt',$end_time));
        $map['status']  =  1;
        $list = $New->where($map)->select();
        $this->list =  $list;
        $this->display();
    }
}
```

定义路由规则如下：

```
'URL_ROUTER_ON'   => true, //开启路由
'URL_ROUTE_RULES' => array( //定义路由规则
    'new/:id\d'    => 'News/read',
    'new/:name'    => 'News/read',
    'new/:year\d/:month\d'  => 'News/archive',
),
```

然后，我们访问： http://serverName/index.php/Home/new/8

会匹配到第一个路由规则，实际执行的效果等效于访问：
http://serverName/index.php/Home/News/read/id/8

当访问： http://serverName/index.php/Home/new/hello

会匹配到第二个路由规则，实际执行的效果等效于访问：
http://serverName/index.php/Home/News/read/name/hello

那么如果访问： http://serverName/index.php/Home/new/2012/03

是否会匹配第三个路由规则呢？我们期望的实际执行的效果能够等效于访问：
http://serverName/index.php/Home/News/archive/year/2012/month/03

事实上却没有，因为 http://serverName/index.php/Home/new/2012/ 这个URL在进行路由匹配过程中已经优先匹配到了第一个路由规则了，把2012当成id的值传入了，这种情况属于路由规则的冲突，解决办法有两个：

1、调整定义顺序

路由定义改成：

```
'URL_ROUTE_RULES' => array( //定义路由规则
    'new/:year\d/:month\d'  => 'News/archive',
    'new/:id\d'                    => 'News/read',
    'new/:name'    => 'News/read',
),
```

接下来，当我们再次访问： http://serverName/index.php/Home/new/2012/03

的时候，达到了预期的访问效果。所以如果存在可能规则冲突的情况，尽量把规则复杂的规则定义放到前面，确保最复杂的规则可以优先匹配到。但是如果路由规则定义多了之后，仍然很容易混淆，所以需要寻找更好的解决办法。

2、利用完全匹配功能

现在我们来利用路由的完全匹配定义功能，把路由定义改成：

```
'URL_ROUTE_RULES' => array( //定义路由规则
    'new/:id\d$'                    => 'News/read',
    'new/:name$'    => 'News/read',
    'new/:year\d/:month\d$'  => 'News/archive',
),
```

在规则最后加上$符号之后，表示完整匹配当前的路由规则，就可以避免规则定义的冲突了。对于规则路由来说，简单的理解就是URL里面的参数数量或者类型约束要完全一致。 所以，如果我们访问 `http://serverName/index.php/Home/new/2012/03/01`

的话，是不会匹配成功任何一条路由的。

3、利用正则路由

当然，解决问题的办法总是不止一种，对于复杂的情况，我们不要忘了使用正则路由规则定义，在你找不到解决方案的时候，正则路由总能帮到你。 要实现上面的同样路由功能的话，还可以用下面的规则定义：

```
'URL_ROUTE_RULES' => array( //定义路由规则
    '/^new\/(\d+)$/'        => 'News/read?id=:1',
    '/^new\/(\w+)$/'        => 'News/read?name=:1',
    '/^new\/(\d{4})\/(\d{2})$/' => 'News/achive?year=:1&month=:2',
),
```

控制器

控制器定义

控制器和操作

一般来说，ThinkPHP的控制器是一个类，而操作则是控制器类的一个公共方法。

下面就是一个典型的控制器类的定义：

```
<?php
namespace Home\Controller;
use Think\Controller;
class IndexController extends Controller {
    public function hello(){
        echo 'hello,thinkphp!';
    }
}
```

`Home\IndexController` 类就代表了Home模块下的Index控制器，而hello操作就是 `Home\IndexController` 类的hello（公共）方法。

当访问 `http://serverName/index.php/Home/Index/hello` 后会输出：

```
hello,thinkphp!
```

> 注意：如果你设置了操作方法绑定到类，则操作方法对应了一个类（参考操作绑定到类）。

定义控制器

控制器通常需要继承系统的Controller类或者其子类，例如，下面定义了一个 `\Home\Controller\IndexController` 控制器类：

```
<?php
namespace Home\Controller;
use Think\Controller;
class IndexController extends Controller {
    public function hello(){
        echo 'hello';
    }

    public function test(){
        echo 'test';
    }
}
```

控制器的名称采用驼峰法命名（首字母大写），控制器文件位于`Home/Controller/IndexController.class.php`。

IndexController控制器类的hello和test方法就是操作方法，访问下面的URL地址：

```
http://serverName/Home/Index/hello
http://serverName/Home/Index/test
```

会分别输出：

```
hello
// 和
test
```

操作方法的定义必须是公共方法，否则会报操作错误，所以，下面的操作定义只能访问hello操作，而不能访问test操作。

```
<?php
namespace Home\Controller;
use Think\Controller;
class IndexController extends Controller {
    public function hello(){
        echo 'hello';
    }

    protected function test(){
        echo 'test';
    }
}
```

注意：定义控制器方法的时候，尽量避免和系统的保留方法相冲突（`除非你非常明确自己在做什么`），这些保留方法名包括但不限于：

```
display
get
show
fetch
theme
assign
error
success
```

因为操作方法就是控制器的一个方法，所以遇到有和系统的关键字冲突的方法可能就不能定义了，这个时候我们可以设置操作方法的后缀来解决，例如：

```
'ACTION_SUFFIX'         =>  'Action', // 操作方法后缀
```

设置操作方法的后缀为Action，这样，控制器的操作方法定义调整为：

```
<?php
namespace Home\Controller;
use Think\Controller;
class IndexController extends Controller {
    public function listAction(){
        echo 'list';
    }

    public function helloAction(){
        echo 'hello';
    }

    public function testAction(){
        echo 'test';
    }
}
```

> 操作方法的后缀设置只是影响控制器类的定义，对URL访问没有影响。

多层控制器

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

```
├─Controller 访问控制器
│  ├─UserController.class.php
│  ├─BlogController.class.php
│  ...
├─Event 事件控制器
│  ├─UserEvent.class.php
│  ├─BlogEvent.class.php
│  ...
```

> 访问控制器的名称是通过DEFAULT_C_LAYER设置的，默认是Controller。

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

> 多层控制器的划分可以根据项目的需要自由分层。

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

```
<?php
namespace Home\Event;
class UserEvent {
    public function login(){
        echo 'login event';
    }

    public function logout(){
        echo 'logout event';
    }
}
```

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

多级控制器

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

```
'CONTROLLER_LEVEL'    =>  2,
```

控制器文件的位置放置如下：

```
├─Controller 访问控制器
│  ├─User User分级（组）
│  │  ├─UserTypeController.class.php
│  │  ├─UserAuthController.class.php
│  ...
│  ├─Admin Admin分级（组）
│  │  ├─UserController.class.php
│  │  ├─ConfigController.class.php
│  ...
```

多级控制器中的命名空间需要这样定义：

```
<?php
namespace Home\Controller\Admin;
use Think\Controller;
class IndexController extends Controller {
    public function hello(){
        echo 'hello';
    }

    public function test(){
        echo 'test';
    }
}
```

然后就可以通过URL地址访问：

```
http://serverName/Home/User/UserType
http://serverName/Home/Admin/User
```

> 如果希望简化URL地址中的模块地址，可以参考 模块部署

实例化控制器

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

如果你需要跨控制器调用的话，则可以单独实例化：

```
// 实例化Home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\BlogController();
```

系统为上面的控制器实例化提供了一个快捷调用方法A，上面的代码可以简化为：

```
// 假设当前模块是Home模块
$User = A('User');
$Blog = A('Admin/Blog');
```

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

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

上面的代码等效于：

```
// 实例化Home模块的User事件控制器
$User = new \Home\Event\UserEvent();
// 实例化Admin模块的Blog事件控制器
$Blog = new \Admin\Event\BlogEvent();
```

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。

> 其他的分层控制器层和内部调用控制器的情况下前置和后置操作是无效的。

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：

```
<?php
namespace Home\Controller;
use Think\Controller;
class IndexController extends Controller{
    //前置操作方法
    public function _before_index(){
        echo 'before<br/>';
    }
    public function index(){
        echo 'index<br/>';
    }
    //后置操作方法
    public function _after_index(){
        echo 'after<br/>';
    }
}
```

如果我们访问 http://serverName/index.php/Home/Index/index

结果会输出

```
before
index
after
```

前置和后置操作的注意事项如下：

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。
2. 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控制器类的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。

Action参数绑定

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

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

要启用参数绑定功能，首先确保你开启了 URL_PARAMS_BIND 设置：

```
'URL_PARAMS_BIND'       =>  true, // URL变量绑定到操作方法作为参数
```

参数绑定有两种方式：按照变量名绑定和按照变量顺序绑定。

按变量名绑定

默认的参数绑定方式是按照变量名进行绑定，例如，我们给Blog控制器定义了两个操作方法read和archive方法，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，那么我们可以如下定义：

```
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='.$year.'&month='.$month;
    }
}
```

> 注意这里的操作方法并没有具体的业务逻辑，只是简单的示范。

URL的访问地址分别是：

```
http://serverName/index.php/Home/Blog/read/id/5
http://serverName/index.php/Home/Blog/archive/year/2013/month/11
```

两个URL地址中的id参数和year和month参数会自动和read操作方法以及archive操作方法的同名参数绑定。

> 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

输出的结果依次是：

```
id=5
year=2013&month=11
```

按照变量名进行参数绑定的参数必须和URL中传入的变量名称一致，但是参数顺序不需要一致。也就是说

```
http://serverName/index.php/Home/Blog/archive/month/11/year/2013
```

和上面的访问结果是一致的，URL中的参数顺序和操作方法中的参数顺序都可以随意调整，关键是确保参数名称一致即可。

如果使用下面的URL地址进行访问，参数绑定仍然有效：

```
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
```

如果用户访问的URL地址是（至于为什么会这么访问暂且不提）：

```
http://serverName/index.php/Home/Blog/read/
```

那么会抛出下面的异常提示： `参数错误:id`

报错的原因很简单，因为在执行read操作方法的时候，id参数是必须传入参数的，但是方法无法从URL地址中获取正确的id参数信息。由于我们不能相信用户的任何输入，因此建议你给read方法的id参数添加默认值，例如：

```
    public function read($id=0){
        echo 'id='.$id;
    }
```

这样，当我们访问 `http://serverName/index.php/Home/Blog/read/` 的时候 就会输出

```
id=0
```

当我们访问 `http://serverName/index.php/Home/Blog/archive/` 的时候，输出：

```
year=2013&month=01
```

> 始终给操作方法的参数定义默认值是一个避免报错的好办法

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` 为1：

```
'URL_PARAMS_BIND_TYPE'  =>  1, // 设置参数绑定按照变量顺序绑定
```

操作方法的定义不需要改变，URL的访问地址分别改成：

```
http://serverName/index.php/Home/Blog/read/5
http://serverName/index.php/Home/Blog/archive/2013/11
```

输出的结果依次是：

```
id=5
year=2013&month=11
```

这个时候如果改成

```
http://serverName/index.php/Home/Blog/archive/11/2013
```

输出的结果就变成了：

```
year=11&month=2013
```

显然就有问题了，所以不能随意调整参数在URL中的传递顺序，要确保和你的操作方法定义顺序一致。

> 可以看到，这种参数绑定的效果有点类似于简单的规则路由。

按变量顺序绑定的方式目前仅对PATHINFO地址有效，所以下面的URL访问参数绑定会失效：

```
http://serverName/index.php?c=Blog&a=read&id=5
http://serverName/index.php?c=Blog&a=archive&year=2013&month=11
```

但是，兼容模式URL地址访问依然有效：

```
http://serverName/index.php?s=/Home/Blog/read/5
http://serverName/index.php?s=/Home/Blog/archive/2013/11
```

如果你的操作方法定义都不带任何参数或者不希望使用该功能的话，可以关闭参数绑定功能：

```
'URL_PARAMS_BIND'      =>  false
```

伪静态

URL伪静态通常是为了满足更好的SEO效果，ThinkPHP支持伪静态URL设置，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，而不会影响当前操作的正常执行。例如，我们设置

```
'URL_HTML_SUFFIX'=>'shtml'
```

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

后者更具有静态页面的URL特征，但是具有和前面的URL相同的执行效果，并且不会影响原来参数的使用。

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，

```
'URL_HTML_SUFFIX'=>''
```

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

例如：

```
http://serverName/Home/blog/3.html
http://serverName/Home/blog/3.shtml
http://serverName/Home/blog/3.xml
http://serverName/Home/blog/3.pdf
```

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

如果希望支持多个伪静态后缀，可以直接设置如下：

```
// 多个伪静态后缀设置 用|分割
'URL_HTML_SUFFIX' => 'html|shtml|xml'
```

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

可以设置禁止访问的URL后缀，例如：

```
'URL_DENY_SUFFIX' => 'pdf|ico|png|gif|jpg', // URL禁止访问的后缀设置
```

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

> 注意：
> URL_DENY_SUFFIX 的优先级比 URL_HTML_SUFFIX 要高。

URL大小写

系统默认的规范是根据URL里面的模块名、控制器名来定位到具体的控制器类的，从而执行控制器类的操作方法。

以URL访问 http://serverName/index.php/Home/Index/index 为例，其实访问的控制器类文件是：

```
Application/Home/Controller/IndexController.class.php
```

如果是Windows环境，无论大小写如何都能定位到 IndexController.class.php 文件，所以下面的访问都是有效的：

```
http://serverName/index.php/Home/Index/index
http://serverName/index.php/Home/index/index
http://serverName/index.php/home/index/index
```

如果在Linux环境下面，一旦大小写不一致，就会发生URL里面使用小写模块名不能找到模块类的情况。例如在Linux环境下面，我们访问 `http://serverName/index.php/home/index/index` 其实请求的控制器文件是

```
Application/home/Controller/indexController.class.php
```

因为，我们定义的控制器类是IndexController而不是indexController（参考ThinkPHP的命名规范），由于Linux的文件特性，其实是不存在indexController控制器文件的，就会出现Index控制器不存在的错误，这样的问题会造成用户体验的下降。

但是系统本身提供了一个不区分URL大小写的解决方案，可以通过配置简单实现。

只要在项目配置中，增加：

```
'URL_CASE_INSENSITIVE' =>true
```

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

```
http://serverName/index.php/Home/Index/index
// 将等效于
http://serverName/index.php/home/index/index
```

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

```
// 正确的访问地址
http://serverName/index.php/home/user_type/index
// 错误的访问地址（linux环境下）
http://serverName/index.php/home/usertype/index
```

利用系统提供的U方法（后面一章URL生成会告诉你如何生成）可以为你自动生成相关的URL地址。

如果设置

```
'URL_CASE_INSENSITIVE' =>false
```

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

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

> URL不区分大小写。

URL生成

为了配合所使用的URL模式，我们需要能够动态的根据当前的URL设置生成对应的URL地址，为此，ThinkPHP内置提供了U方法，用于URL的动态生成，可以确保项目在移植过程中不受环境的影响。

定义规则

U方法的定义规则如下（方括号内参数根据实际应用决定）：

U('地址表达式',['参数'],['伪静态后缀'],['显示域名'])

地址表达式

地址表达式的格式定义如下：

```
[模块/控制器/操作#锚点@域名]?参数1=值1&参数2=值2...
```

如果不定义模块的话 就表示当前模块名称，下面是一些简单的例子：

```
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
```

参数

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

```
U('Blog/cate',array('cate_id'=>1,'status'=>1))
U('Blog/cate','cate_id=1&status=1')
U('Blog/cate?cate_id=1&status=1')
```

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

但是不允许使用下面的定义方式来传参数

```
U('Blog/cate/cate_id/1/status/1');
```

伪静态后缀

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

```
U('Blog/cate','cate_id=1&status=1','xml');
```

自动识别

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

```
U ( 'Blog/read?id=1' ) ;
```

这个定义为例。

如果当前URL设置为普通模式的话，最后生成的URL地址是：

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

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

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

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

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

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

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

> 如果开启了`URL_CASE_INSENSITIVE`，则会统一生成小写的URL地址。

生成路由地址

U方法还可以支持路由，如果我们定义了一个路由规则为：

```
 'news/:id\d'=>'News/read'
```

那么可以使用

```
U ( '/news/1' ) ;
```

最终生成的URL地址是：

```
http://serverName/index.php/Home/news/1
```

> 注意：如果你是在模板文件中直接使用U方法的话，需要采用 {:U('参数1', '参数2'...)} 的方式，具体参考模板的使用函数内容。

域名支持

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

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

@后面传入需要指定的域名即可。

系统会自动判断当前是否SSL协议，生成 `https://` 。

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

锚点支持

U函数可以直接生成URL地址中的锚点，例如：

```
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
```

AJAX返回

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

ajaxReturn方法调用示例：

```
$data = 'ok';
$this->ajaxReturn($data);
```

支持返回数组数据：

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

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

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

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

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

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

```
'DEFAULT_JSONP_HANDLER' =>  'myJsonpReturn', // 默认JSONP格式返回的处理方法
```

或者直接在页面中用callback参数来指定。

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

跳转和重定向

页面跳转

在应用开发中，经常会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，并且自动跳转到另外一个目标页面。系统的 `\Think\Controller` 类内置了两个跳转方法success和error，用于页面跳转提示，而且可以支持ajax提交。

使用方法很简单，举例如下：

```
$User = M('User'); //实例化User对象
$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/index');
} else {
    //错误页面的默认跳转页面是返回前一页，通常不需要设置
    $this->error('新增失败');
}
```

success和error方法的第一个参数表示提示信息，第二个参数表示跳转地址，第三个参数是跳转时间（单位为秒），例如：

```
// 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',
```

模板文件可以使用模板标签，并且可以使用下面的模板变量：

变量	含义
$message	页面提示信息
$error	页面错误提示信息
$waitSecond	跳转等待时间 单位为秒

变量	含义
$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 如果是success是1 error 是0
$data['url']    =   $jumpUrl; // 成功或者错误的跳转地址
```

重定向

Controller类的redirect方法可以实现页面的重定向功能。

redirect方法的参数用法和U函数的用法一致（参考URL生成部分），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。

输入变量

在Web开发过程中，我们经常需要获取系统变量或者用户提交的数据，这些变量数据错综复杂，而且一不小心就容易引起安全隐患，但是如果利用好ThinkPHP提供的变量获取功能，就可以轻松的获取和驾驭变量了。

获取变量

虽然你仍然可以在开发过程中使用传统方式获取各种系统变量，例如：

```
$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
```

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

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

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

变量类型是指请求方式或者输入类型，包括：

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

> 注意：变量类型不区分大小写，变量名则严格区分大小写。
> 默认值和过滤方法均属于可选参数。

我们以GET变量类型为例，说明下I方法的使用：

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

支持默认值：

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

采用方法过滤：

```
// 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');
```

支持直接获取整个变量类型，例如：

```
// 获取整个$_GET 数组
I('get.');
```

用同样的方式，我们可以获取post或者其他输入类型的变量，例如：

```
I('post.name','','htmlspecialchars'); // 采用htmlspecialchars方法对$_POST['name'] 进行过滤，如果不存在则返回空字符串
I('session.user_id',0); // 获取$_SESSION['user_id'] 如果不存在则默认为0
I('cookie.'); // 获取整个 $_COOKIE 数组
I('server.REQUEST_METHOD'); // 获取 $_SERVER['REQUEST_METHOD']
```

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

```
echo I('param.id');
```

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

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

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

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

那么我们可以通过

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

data类型变量可以用于获取不支持的变量类型的读取，例如：

```
I('data.file1','','',$_FILES);
```

变量过滤

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

```
// 系统默认的变量过滤机制
'DEFAULT_FILTER'        => 'htmlspecialchars'
```

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

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

同样，该参数也可以设置支持多个过滤，例如：

```
'DEFAULT_FILTER'        => 'strip_tags,htmlspecialchars'
```

设置后，我们在使用：

```
// 等同于 htmlspecialchars(strip_tags($_GET['name']))
I('get.name');
```

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

```
// 等同于 strip_tags($_GET['name'])
echo I('get.name','','strip_tags');
```

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

```
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');
```

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

```
int
boolean
float
validate_regexp
validate_url
validate_email
validate_ip
string
stripped
encoded
special_chars
unsafe_raw
email
url
number_int
number_float
magic_quotes
callback
```

还可以支持进行正则匹配过滤，例如：

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

如果正则匹配不通过的话，则返回默认值。

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

```
// 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。

变量修饰符

最新版本的I函数支持对变量使用修饰符功能，可以更方便的通过类型过滤变量。

用法如下：

```
I('变量类型.变量名/修饰符')
```

例如：

```
I('get.id/d'); // 强制变量转换为整型
I('post.name/s'); // 强制转换变量为字符串类型
I('post.ids/a'); // 强制变量转换为数组类型
```

可以使用的修饰符包括：

修饰符	作用
s	强制转换为字符串类型
d	强制转换为整型类型
b	强制转换为布尔类型
a	强制转换为数组类型
f	强制转换为浮点类型

请求类型

判断请求类型

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

系统内置了一些常量用于判断请求类型，包括：

常量	说明
IS_GET	判断是否是GET方式提交
IS_POST	判断是否是POST方式提交
IS_PUT	判断是否是PUT方式提交
IS_DELETE	判断是否是DELETE方式提交
IS_AJAX	判断是否是AJAX提交
REQUEST_METHOD	当前提交类型

使用举例如下：

```
class UserController extends Controller{
    public function update(){
        if (IS_POST){
            $User = M('User');
            $User->create();
            $User->save();
            $this->success('保存完成');
        }else{
            $this->error('非法请求');
        }
    }
}
```

> 个别情况下，你可能需要在表单里面添加一个隐藏域，告诉后台属于ajax方式提交，默认的隐藏域名称是ajax（可以通过VAR_AJAX_SUBMIT配置），如果是JQUERY类库的话，则无需添加任何隐藏域即可自动判断。

空操作

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

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

```
<?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：

```
当前城市:beijing
当前城市:shanghai
当前城市:shenzhen
```

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

空控制器

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

现在我们把前面的需求进一步，把URL由原来的

```
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。

我们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
        $cityName = CONTROLLER_NAME;
        $this->city($cityName);
    }
    //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/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
```

> 空控制器和空操作还可以同时使用，用以完成更加复杂的操作。

插件控制器

插件控制器可以更加方便的在控制器以外扩展你的功能，当URL中传入插件控制器变量的时候，会自动定位到插件控制器中的操作方法。

插件控制器的变量由参数 VAR_ADDON 进行设置，默认为addon，例如我们在URL中传入：`http://serverName/Home/info/index/addon/SystemInfo`

由于传入了addon参数，因此这里的Info控制器并非原来的`Home/Controller/InfoController.class.php`

而是调用SystemInfo插件的InfoController控制器了，文件位于`Addon/SystemInfo/Controller/InfoController.class.php`。

插件控制器本身的定义和普通的访问控制器一样，例如：

```
namespace Addon\SystemInfo\Controller;
class InfoController extends \Think\Controller{
    public function index(){
        echo 'Addon SystemInfo';
    }
}
```

这样，我们在访问 http://serverName/Home/info/index/addon/SystemInfo 的时候 就会输出 Addon SystemInfo

如果我们的插件目录不是Addon，而是Plugin，那么需要在配置文件中定义：

```
'VAR_ADDON'    =>    'plugin'
```

然后访问URL地址就变成了 http://serverName/Home/info/index/plugin/SystemInfo

操作绑定到类

定义

ThinkPHP3.2版本提供了把每个操作方法定位到一个类的功能，可以让你的开发工作更细化，可以设置参数ACTION_BIND_CLASS，例如：

```
'ACTION_BIND_CLASS'    =>    True,
```

设置后，我们的控制器定义有所改变，以URL访问为 http://serverName/Home/Index/index 为例，原来的控制器文件定义位置为：

```
Application/Home/Controller/IndexController.class.php
```

控制器类的定义如下：

```
namespace Home\Controller;
use Think\Controller;
class IndexController extends Controller{
    public function index(){
        echo '执行Index控制器的index操作';
    }
}
```

可以看到，实际上我们调用的是 `Home\Controller\IndexController` 类的index方法。

设置后，控制器文件位置改为：

```
Application/Home/Controller/Index/index.class.php
```

控制器类的定义如下：

```
namespace Home\Controller\Index;
use Think\Controller;
class index extends Controller{
    public function run(){
        echo '执行Index控制器的index操作';
    }
}
```

现在，我们调用的其实是 `Home\Controller\Index\index` 类的run方法。

> run方法依旧可以支持传入参数和进行Action参数绑定操作，但不再支持A方法实例化和R方法远程调用，我们建议R方法不要进行当前访问控制器的远程调用。

前置和后置操作

当设置操作方法绑定到类后，前置和后置操作的定义有所改变，只需要在类里面定义 _before_run 和 _after_run 方法即可，例如：

```
namespace Home\Controller\Index;
use Think\Controller;
class index extends Controller{
    public function _before_run(){
        echo 'before_'.ACTION_NAME;
    }

    public function run(){
        echo '执行Index控制器的index操作';
    }

    public function _after_run(){
        echo 'after_'.ACTION_NAME;
    }
}
```

空控制器

操作方法绑定到类后，一样可以支持空控制器，我们可以创建 `Application/Home/Controller/_empty` 目录，即表示如果找不到当前的控制器的话，会到_empty控制器目录下面定位操作方法。

例如，我们访问了URL地址 http://serverName/Home/Test/index ,但并不存在 Application/Home/Controller/Test 目录，但是有定义 Application/Home/Controller/_empty 目录。

并且我们有定义：

```
Application/Home/Controller/_empty/index.class.php
```

控制器定义如下：

```
namespace Home\Controller\_empty;
use Think\Controller;
class index extends Controller{
    public function run(){
        echo '执行'CONTROLLER_NAME.'控制器的'.ACTION_NAME.'操作';
    }
}
```

访问 http://serverName/Home/Test/index 后 输出结果显示：

```
执行Test控制器的index操作
```

空操作

操作绑定到类后，我们依然可以实现空操作方法，我们只要定义一个 Home\Controller\Index_empty 类，就可以支持Index控制器的空操作访问，例如： 控制器定义如下：

```
namespace Home\Controller\Index;
use Think\Controller;
class _empty extends Controller{
    public function run(){
        echo '执行Index控制器的'.ACTION_NAME.'操作';
    }
}
```

当我们访问 http://serverName/Home/Index/test 后 输出结果显示：

```
执行Index控制器的test操作
```

模型

在ThinkPHP中基础的模型类就是 Think\Model 类，该类完成了基本的CURD、ActiveRecord模式、连贯操作和统计查询，一些高级特性被封装到另外的模型扩展中。

基础模型类的设计非常灵活，甚至可以无需进行任何模型定义，就可以进行相关数据表的ORM和CURD操作，只有在需要封装单独的业务逻辑的时候，模型类才是必须被定义的。

模型定义

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。

模型类通常需要继承系统的\Think\Model类或其子类，下面是一个Home\Model\UserModel类的定义：

```
namespace Home\Model;
use Think\Model;
class UserModel extends Model {
}
```

模型类的作用大多数情况是操作数据表的，如果按照系统的规范来命名模型类的话，大多数情况下是可以自动对应数据表。

模型类的命名规则是除去表前缀的数据表名称，采用驼峰法命名，并且首字母大写，然后加上模型层的名称（默认定义是Model），例如：

模型名	约定对应数据表（假设数据库的前缀定义是 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。

数据表定义

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。

举个例子来加深理解，例如，在数据库里面有一个 think_categories 表，而我们定义的模型类名称是 CategoryModel ，按照系统的约定，这个模型的名称是Category，对应的数据表名称应该是 think_category （全部小写），但是现在的数据表名称是 think_categories ，因此我们就需要设置 tableName 属性来改变默认的规则（假设我们已经在配置文件里面定义了 DB_PREFIX 为 think_）。

```
namespace Home\Model;
use Think\Model;
class CategoryModel extends Model {
    protected $tableName = 'categories';
}
```

注意这个属性的定义不需要加表的前缀 think_

如果我们需要CategoryModel模型对应操作的数据表是 top_category ，那么我们只需要设置数据表前缀即可：

```
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 。

而对于另外一种特殊情况，我们需要操作的数据表是 top_categories ，这个时候我们就需要定义

`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';
}
```

> 系统的规则下，tableName会转换为小写定义，但是trueTableName定义的数据表名称是保持原样。因此，如果你的数据表名称需要区分大小写的情况，那么可以通过设置trueTableName定义来解决。

模型实例化

在ThinkPHP中，可以无需进行任何模型定义。只有在需要封装单独的业务逻辑的时候，模型类才是必须被定义的，因此ThinkPHP在模型上有很多的灵活和方便性，让你无需因为表太多而烦恼。

根据不同的模型定义，我们有几种实例化模型的方法，根据需要采用不同的方式：

直接实例化

可以和实例化其他类库一样实例化模型类，例如：

```
$User = new \Home\Model\UserModel();
$Info = new \Admin\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，分别是：

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

> 数据表前缀传入空字符串表示取当前配置的表前缀，如果当前数据表没有前缀，则传入null即可。

数据库连接信息参数支持三种格式：

1、字符串定义

字符串定义采用DSN格式定义，格式定义规范为：

数据库类型://用户名:密码@数据库主机名或者IP:数据库端口/数据库名#字符集

例如：

```
new \Home\Model\NewModel('blog','think_','mysql://root:1234@localhost/demo');
```

2、数组定义

可以传入数组格式的数据库连接信息，例如：

```
$connection = array(
    'db_type'    =>   'mysql',
    'db_host'    =>   '127.0.0.1',
    'db_user'    =>   'root',
    'db_pwd'     =>   '12345',
    'db_port'    =>    3306,
    'db_name'    =>    'demo',
    'db_charset' =>    'utf8',
);
new \Home\Model\NewModel('new','think_',$connection);
```

如果需要的话，还可以传入更多的连接参数，包括数据的部署模式和调试模式设定，例如：

```
$connection = array(
    'db_type'    =>   'mysql',
    'db_host'    =>   '192.168.1.2,192.168.1.3',
    'db_user'    =>   'root',
    'db_pwd'     =>   '12345',
    'db_port'    =>    3306,
    'db_name'    =>    'demo',
    'db_charset' =>    'utf8',
    'db_deploy_type'=>    1,
    'db_rw_separate'=>    true,
    'db_debug'    =>    true,
);
// 分布式数据库部署 并且采用读写分离 开启数据库调试模式
new \Home\Model\NewModel('new','think_',$connection);
```

> 注意，如果设置了db_debug参数，那么数据库调试模式就不再受APP_DEBUG常量影响。

3、配置定义

我们可以事先在配置文件中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，例如：

```
//数据库配置1
'DB_CONFIG1' => array(
    'db_type'  => 'mysql',
    'db_user'  => 'root',
    'db_pwd'   => '1234',
    'db_host'  => 'localhost',
    'db_port'  => '3306',
    'db_name'  => 'thinkphp'
),
//数据库配置2
'DB_CONFIG2' => 'mysql://root:1234@localhost:3306/thinkphp',
```

在配置文件中定义数据库连接信息的时候也支持字符串和数组格式，格式和上面实例化传入的参数一样。

然后，我们就可以这样实例化模型类传入连接信息：

```
new \Home\Model\NewModel('new','think_','DB_CONFIG1');
new \Home\Model\BlogModel('blog','think_','DB_CONFIG2');
```

事实上，当我们实例化的时候没有传入任何的数据库连接信息的时候，系统其实默认会获取配置文件中的相关配置参数，包括：

```
'DB_TYPE'      =>  '',     // 数据库类型
'DB_HOST'      =>  '',     // 服务器地址
'DB_NAME'      =>  '',     // 数据库名
'DB_USER'      =>  '',     // 用户名
'DB_PWD'       =>  '',     // 密码
'DB_PORT'      =>  '',     // 端口
'DB_PREFIX'    =>  '',     // 数据库表前缀
'DB_DSN'       =>  '',     // 数据库连接DSN 用于PDO方式
'DB_CHARSET'   =>  'utf8', // 数据库的编码 默认为utf8
```

如果应用配置文件中有配置上述数据库连接信息的话，实例化模型将会变得非常简单。

D方法实例化

上面实例化的时候我们需要传入完整的类名，系统提供了一个快捷方法D用于数据模型的实例化操作。

要实例化自定义模型类，可以使用下面的方式：

```
<?php
//实例化模型
$User = D('User');
// 相当于 $User = new \Home\Model\UserModel();
// 执行具体的数据操作
$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方法的参数就是模型的名称，并且和模型类的大小写定义是一致的，例如：

参数	实例化的模型文件（假设当前模块为Home）
User	对应的模型类文件的 \Home\Model\UserModel.class.php
UserType	对应的模型类文件的 \Home\Model\UserTypeModel.class.php

> 如果在Linux环境下面，一定要注意D方法实例化的时候的模型名称的大小写。

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。

M方法实例化模型

D方法实例化模型类的时候通常是实例化某个具体的模型类，如果你仅仅是对数据表进行基本的CURD操作的话，使用M方法实例化的话，由于不需要加载具体的模型类，所以性能会更高。

例如：

```
// 使用M方法实例化
$User = M('User');
// 和用法 $User = new \Think\Model('User'); 等效
// 执行其他的数据操作
$User->select();
```

M方法也可以支持跨库操作，例如：

```
// 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);
```

具体的参数含义可以参考前面的介绍。

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');
```

> 如果你的模型类有自己的业务逻辑，M方法是无法支持的，就算是你已经定义了具体的模型类，M方法实例化的时候是会直接忽略。

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');
```

> 实例化空模型类后还可以用table方法切换到具体的数据表进行操作

我们在实例化的过程中，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。

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.数据表前缀+模型名（小写）.php

例如：

```
demo.think_user.php    // User模型生成的字段缓存文件
demo.top_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）。

可以通过设置 DB_FIELDS_CACHE 参数来关闭字段自动缓存，如果在开发的时候经常变动数据库的结构，而不希望进行数据表的字段缓存，可以在项目配置文件中增加如下配置：

```
// 关闭字段缓存
'DB_FIELDS_CACHE'=>false
```

> 注意：调试模式下面由于考虑到数据结构可能会经常变动，所以默认是关闭字段缓存的。

如果需要显式获取当前数据表的字段信息，可以使用模型类的getDbFields方法来获取当前数据对象的全部字段信息，例如：

```
$User   = M('User');
$fields = $User->getDbFields();
```

如果你在部署模式下面修改了数据表的字段信息，可能需要清空 Data/_fields 目录下面的缓存文件，让系统重新获取更新的数据表字段信息，否则会发生新增的字段无法写入数据库的问题。

如果不希望依赖字段缓存或者想提高性能，也可以在模型类里面手动定义数据表字段的名称，可以避免IO加载的效率开销，例如：

```
namespace Home\Model;
use Think\Model;
class UserModel extends Model {
    protected $fields = array('id', 'username', 'email', 'age');
    protected $pk     = 'id';
}
```

pk 属性定义当前数据表的主键名，默认值就是id，因此如果是id的话可以无需定义。

如果你的数据表使用了复合主键，可以这样定义：

```
namespace Home\Model;
use Think\Model;
class ScoreModel extends Model {
    protected $fields = array('user_id', 'lession_id','score');
    protected $pk     = array('user_id','lession_id');
}
```

除了可以设置数据表的字段之外，我们还可以定义字段的类型，用于某些验证环节。例如：

```
namespace Home\Model;
use Think\Model;
class UserModel extends Model {
    protected $fields = array('id', 'username', 'email', 'age',
        '_type'=>array('id'=>'bigint','username'=>'varchar','email'=>'varchar','age'=>'int')
    );
}
```

连接数据库

ThinkPHP内置了抽象数据库访问层，把不同的数据库操作封装起来，我们只需要使用公共的Db类进行操

作，而无需针对不同的数据库写不同的代码和底层实现，Db类会自动调用相应的数据库驱动来处理。目前包含了Mysql、SqlServer、PgSQL、Sqlite、Oracle、Ibase、Mongo等数据库的支持，并且采用PDO方式。

如果应用需要使用数据库，必须配置数据库连接信息，数据库的配置文件有多种定义方式。

一、全局配置定义

常用的配置方式是在应用配置文件或者模块配置文件中添加下面的配置参数：

```
//数据库配置信息
'DB_TYPE'   => 'mysql', // 数据库类型
'DB_HOST'   => '127.0.0.1', // 服务器地址
'DB_NAME'   => 'thinkphp', // 数据库名
'DB_USER'   => 'root', // 用户名
'DB_PWD'    => '123456', // 密码
'DB_PORT'   => 3306, // 端口
'DB_PARAMS' =>  array(), // 数据库连接参数
'DB_PREFIX' => 'think_', // 数据库表前缀
'DB_CHARSET'=> 'utf8', // 字符集
'DB_DEBUG'  =>  TRUE, // 数据库调试模式 开启后可以记录SQL日志
```

数据库的类型由DB_TYPE参数设置。

下面是目前支持的数据库设置：

DB_TYPE设置	数据库类型
mysql	mysql
pgsql	pgsql
sqlite	sqlite
sqlsrv	sqlserver
oracle	oracle
firebird	ibase
mongo	mongo

配置文件定义的数据库连接信息一般是系统默认采用的，因为一般一个应用的数据库访问配置是相同的。该方法系统在连接数据库的时候会自动获取，无需手动连接。

可以对每个模块定义不同的数据库连接信息，如果开启了调试模式的话，还可以在不同的应用状态的配置文件里面定义独立的数据库配置信息。

长连接

如果需要使用长连接，可以采用下面的方式定义：

```
'DB_PARAMS' => array(PDO::ATTR_PERSISTENT => true),
```

> 你可以在DB_PARAMS里面配置任何PDO支持的连接参数。

二、模型类定义

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：

```
//在模型里单独设置数据库连接信息
namespace Home\Model;
use Think\Model;
class UserModel extends Model{
    protected $connection = array(
        'db_type'  => 'mysql',
        'db_user'  => 'root',
        'db_pwd'   => '1234',
        'db_host'  => 'localhost',
        'db_port'  => '3306',
        'db_name'  => 'thinkphp',
        'db_charset' =>    'utf8',
        'db_params' =>  array(), // 非必须
    );
}
```

也可以采用字符串方式定义，定义格式为：

数据库类型://用户名:密码@数据库地址:数据库端口/数据库名#字符集

例如：

```
//在模型里单独设置数据库连接信息
namespace Home\Model;
use Think\Model;
class UserModel extends Model{
    //或者使用字符串定义
    protected $connection = 'mysql://root:1234@localhost:3306/thinkphp#utf8';
}
```

> 注意：字符串方式可能无法定义某些参数，例如前缀和连接参数。

如果我们已经在配置文件中配置了额外的数据库连接信息，例如：

```
//数据库配置1
'DB_CONFIG1' => array(
    'db_type'  => 'mysql',
    'db_user'  => 'root',
    'db_pwd'   => '1234',
    'db_host'  => 'localhost',
    'db_port'  => '3306',
    'db_name'  => 'thinkphp',
    'db_charset'=>    'utf8',
),
//数据库配置2
'DB_CONFIG2' => 'mysql://root:1234@localhost:3306/thinkphp#utf8';
```

那么，我们可以把模型类的属性定义改为：

```
//在模型里单独设置数据库连接信息
namespace Home\Model;
use Think\Model;
class UserModel extends Model{
    //调用配置文件中的数据库配置1
    protected $connection = 'DB_CONFIG1';
}
```

```
//在模型里单独设置数据库连接信息
namespace Home\Model;
use Think\Model;
class InfoModel extends Model{
    //调用配置文件中的数据库配置1
    protected $connection = 'DB_CONFIG2';
}
```

三、实例化定义

除了在模型定义的时候指定数据库连接信息外，我们还可以在实例化的时候指定数据库连接信息，例如：如果采用的是M方法实例化模型的话，也可以支持传入不同的数据库连接信息，例如：

```
$User = M('User','other_','mysql://root:1234@localhost/demo#utf8');
```

表示实例化User模型，连接的是demo数据库的other_user表，采用的连接信息是第三个参数配置的。如果我们在项目配置文件中已经配置了 DB_CONFIG2 的话，也可以采用：

```
$User = M('User','other_','DB_CONFIG2');
```

> 需要注意的是，ThinkPHP的数据库连接是惰性的，所以并不是在实例化的时候就连接数据库，而是在有实际的数据操作的时候才会去连接数据库（额外的情况是，在系统第一次实例化模型的时候，会自动

> 连接数据库获取相关模型类对应的数据表的字段信息）。

切换数据库

除了在预先定义数据库连接和实例化的时候指定数据库连接外，我们还可以在模型操作过程中动态的切换数据库，支持切换到相同和不同的数据库类型。用法很简单， 只需要调用Model类的db方法，用法：

Model->db("数据库编号","数据库配置");

数据库编号用数字格式，对于已经调用过的数据库连接，是不需要再传入数据库连接信息的，系统会自动记录。对于默认的数据库连接，内部的数据库编号是0，因此为了避免冲突，请不要再次定义数据库编号为0的数据库配置。

数据库配置的定义方式和模型定义connection属性一样，支持数组、字符串以及调用配置参数三种格式。

Db方法调用后返回当前的模型实例，直接可以继续进行模型的其他操作，所以该方法可以在查询的过程中动态切换，例如：

```
$this->db(1,"mysql://root:123456@localhost:3306/test")->query("查询SQL");
```

该方法添加了一个编号为1的数据库连接，并自动切换到当前的数据库连接。

当第二次切换到相同的数据库的时候，就不需要传入数据库连接信息了，可以直接使用：

```
$this->db(1)->query("查询SQL");
```

如果需要切换到默认的数据库连接，只需要调用：

```
$this->db(0);
```

如果我们已经在项目配置中定义了其他的数据库连接信息，例如：

```
//数据库配置1
'DB_CONFIG1' = array(
    'db_type'  => 'mysql',
    'db_user'  => 'root',
    'db_pwd'   => '1234',
    'db_host'  => 'localhost',
    'db_port'  => '3306',
    'db_name'  => 'thinkphp'
),
//数据库配置2
'DB_CONFIG2' => 'mysql://root:1234@localhost:3306/thinkphp';
```

我们就可以直接在db方法中调用配置进行连接了：

```
$this->db(1,"DB_CONFIG1")->query("查询SQL");
$this->db(2,"DB_CONFIG2")->query("查询SQL");
```

如果切换数据库之后，数据表和当前不一致的话，可以使用table方法指定要操作的数据表：

```
$this->db(1)->table("top_user")->find();
```

分布式数据库支持

ThinkPHP内置了分布式数据库的支持，包括主从式数据库的读写分离，但是分布式数据库必须是相同的数据库类型。

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：

```
//分布式数据库配置定义
'DB_DEPLOY_TYPE'=> 1, // 设置分布式数据库支持
'DB_TYPE'       => 'mysql', //分布式数据库类型必须相同
'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_',
```

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，默认第一个数据库配置是主服务器的配置信息，负责写入数据，如果设置了 DB_MASTER_NUM 参数，则可以支持多个主服务器写入。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。

还可以设置 DB_SLAVE_NO 指定某个服务器进行读操作。

> 3.2.3版本开始，如果从数据库连接错误，会自动切换到主数据库连接。

调用模型的CURD操作的话，系统会自动判断当前执行的方法的读操作还是写操作，如果你用的是原生SQL，那么需要注意系统的默认规则： 写操作必须用模型的execute方法，读操作必须用模型的query方法，否则会发生主从读写错乱的情况。

> 注意：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。

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。

使用也比较简单， 假如我们现在要查询一个User表的满足状态为1的前10条记录，并希望按照用户的创建时间排序 ，代码如下：

```
$User->where('status=1')->order('create_time')->limit(10)->select();
```

这里的 where 、 order 和 limit 方法就被称之为连贯操作方法，除了select方法必须放到最后一个外（因为select方法并不是连贯操作方法），连贯操作的方法调用顺序没有先后，例如，下面的代码和上面的等效：

```
$User->order('create_time')->limit(10)->where('status=1')->select();
```

如果不习惯使用连贯操作的话，还支持直接使用参数进行查询的方式。例如上面的代码可以改写为：

```
$User->select(array('order'=>'create_time','where'=>'status=1','limit'=>'10'));
```

使用数组参数方式的话，索引的名称就是连贯操作的方法名称。其实不仅仅是查询方法可以使用连贯操作，包括所有的CURD方法都可以使用，例如：

```
$User->where('id=1')->field('id,name,email')->find();
$User->where('status=1 and id=1')->delete();
```

连贯操作通常只有一个参数，并且仅在当此查询或者操作有效，完成后会自动清空连贯操作的所有传值（有个别特殊的连贯操作有多个参数，并且会记录当前的传值）。简而言之，连贯操作的结果不会带入以后的查询。

系统支持的连贯操作方法有：

连贯操作	作用	支持的参数类型
where*	用于查询或者更新条件的定义	字符串、数组和对象
table	用于定义要操作的数据表名称	字符串和数组
alias	用于给当前数据表定义别名	字符串
data	用于新增或者更新数据之前的数据对象赋值	数组和对象
field	用于定义要查询的字段（支持字段排除）	字符串和数组
order	用于对结果排序	字符串和数组
limit	用于限制查询结果数量	字符串和数字
page	用于查询分页（内部会转换成limit）	字符串和数字
group	用于对查询的group支持	字符串
having	用于对查询的having支持	字符串
join*	用于对查询的join支持	字符串和数组
union*	用于对查询的union支持	字符串、数组和对象
distinct	用于查询的distinct支持	布尔值
lock	用于数据库的锁机制	布尔值
cache	用于查询缓存	支持多个参数
relation	用于关联查询（需要关联模型支持）	字符串
result	用于返回数据转换	字符串

连贯操作	作用	支持的参数类型
validate	用于数据自动验证	数组
auto	用于数据自动完成	数组
filter	用于数据过滤	字符串
scope*	用于命名范围	字符串、数组
bind*	用于数据绑定操作	数组或多个参数
token	用于令牌验证	布尔值
comment	用于SQL注释	字符串
index	用于数据集的强制索引（3.2.3新增）	字符串
strict	用于数据入库的严格检测（3.2.3新增）	布尔值

> 所有的连贯操作都返回当前的模型实例对象（this），其中带*标识的表示支持多次调用。

WHERE

where方法的用法是ThinkPHP查询语言的精髓，也是ThinkPHP ORM的重要组成部分和亮点所在，可以完成包括普通查询、表达式查询、快捷查询、区间查询、组合查询在内的查询操作。where方法的参数支持字符串和数组，虽然也可以使用对象但并不建议。

字符串条件

使用字符串条件直接查询和操作，例如：

```
$User = M("User"); // 实例化User对象
$User->where('type=1 AND status=1')->select();
```

最后生成的SQL语句是

```
SELECT * FROM think_user WHERE type=1 AND status=1
```

使用字符串条件的时候，建议配合预处理机制，确保更加安全，例如：

```
$Model->where("id=%d and username='%s' and xx='%f'",array($id,$username,$xx))->select();
```

或者使用：

```
$Model->where("id=%d and username='%s' and xx='%f'",$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。

数组条件

数组条件的where用法是ThinkPHP推荐的用法。

普通查询

最简单的数组查询方式如下：

```
$User = M("User"); // 实例化User对象
$map['name'] = 'thinkphp';
$map['status'] = 1;
// 把查询条件传入查询方法
$User->where($map)->select();
```

最后生成的SQL语句是

```
SELECT * FROM think_user WHERE `name`='thinkphp' AND status=1
```

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['字段1']  = array('表达式','查询条件1');
$map['字段2']  = array('表达式','查询条件2');
$Model->where($map)->select(); // 也支持
```

表达式不分大小写，支持的查询表达式有下面几种，分别表示的含义是：

表达式	含义
EQ	等于（=）
NEQ	不等于（<>）
GT	大于（>）
EGT	大于等于（>=）
LT	小于（<）

表达式	含义
ELT	小于等于（<=）
LIKE	模糊查询
[NOT] BETWEEN	（不在）区间查询
[NOT] IN	（不在）IN 查询
EXP	表达式查询，支持SQL语法

多次调用

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。

更多的查询用法，可以参考查询语言部分。

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。

用法

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：

1. 切换操作的数据表；
2. 对多表进行操作；

例如：

```
$Model->table('think_user')->where('status>1')->select();
```

也可以在table方法中指定数据库，例如：

```
$Model->table('db_name.think_user')->where('status>1')->select();
```

table方法指定的数据表需要完整的表名，但可以采用下面的方式简化数据表前缀的传入，例如：

```
$Model->table('__USER__')->where('status>1')->select();
```

会自动获取当前模型对应的数据表前缀来生成 `think_user` 数据表名称。

需要注意的是table方法不会改变数据库的连接，所以你要确保当前连接的用户有权限操作相应的数据库和数据表。 切换数据表后，系统会自动重新获取切换后的数据表的字段缓存信息。

如果需要对多表进行操作，可以这样使用：

```
$Model->field('user.name,role.title')
->table('think_user user,think_role role')
->limit(10)->select();
```

为了尽量避免和mysql的关键字冲突，可以建议使用数组方式定义，例如：

```
$Model->field('user.name,role.title')
->table(array('think_user'=>'user','think_role'=>'role'))
->limit(10)->select();
```

使用数组方式定义的优势是可以避免因为表名和关键字冲突而出错的情况。

> 一般情况下，无需调用table方法，默认会自动获取当前模型对应或者定义的数据表。

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。

示例：

```
$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
```

DATA

data方法也是模型类的连贯操作方法之一，用于设置当前要操作的数据对象的值。

写操作

通常情况下我们都是通过create方法或者赋值的方式生成数据对象，然后写入数据库，例如：

```
$Model = D('User');
$Model->create();
// 这里略过具体的自动生成和验证判断
$Model->add();
```

又或者直接对数据对象赋值，例如：

```
$Model = M('User');
$Model->name = '流年';
$Model->email = 'thinkphp@qq.com';
$Model->add();
```

那么data方法则是直接生成要操作的数据对象，例如：

```
$Model = M('User');
$data['name'] = '流年';
$data['email'] = 'thinkphp@qq.com';
$Model->data($data)->add();
```

> 注意：如果我们同时使用create方法和data创建数据对象的话，则最后调用的方法有效。

data方法支持数组、对象和字符串，对象方式如下：

```
$Model = M('User');
$obj = new \stdClass;
$obj->name = '流年';
$obj->email = 'thinkphp@qq.com';
$Model->data($obj)->add();
```

字符串方式用法如下：

```
$Model = M('User');
$data = 'name=流年&email=thinkphp@qq.com';
$Model->data($data)->add();
```

也可以直接在add方法中传入数据对象来新增数据，例如：

```
$Model = M('User');
$data['name'] = '流年';
$data['email'] = 'thinkphp@qq.com';
$Model->add($data);
```

但是这种方式data参数只能使用数组。

当然data方法也可以用于更新数据，例如：

```
$Model = M('User');
$data['id'] = 8;
$data['name'] = '流年';
$data['email'] = 'thinkphp@qq.com';
$Model->data($data)->save();
```

当然我们也可以直接这样用：

```
$Model = M('User');
$data['id'] = 8;
$data['name'] = '流年';
$data['email'] = 'thinkphp@qq.com';
$Model->save($data);
```

同样，此时data参数只能传入数组。

在调用save方法更新数据的时候 会自动判断当前的数据对象里面是否有主键值存在，如果有的话会自动作为更新条件。也就是说，下面的用法和上面等效：

```
$Model = M('User');
$data['name'] = '流年';
$data['email'] = 'thinkphp@qq.com';
$Model->data($data)->where('id=8')->save();
```

读操作

除了写操作外，data方法还可以用于读取当前的数据对象，例如：

```
$User = M('User');
$map['name'] = '流年';
$User->where($map)->find();
// 读取当前数据对象
$data = $User->data();
```

FIELD

field方法属于模型的连贯操作方法之一，主要目的是标识要返回或者操作的字段，可以用于查询和写入操作。

用于查询

指定字段

在查询操作中field方法是使用最频繁的。

```
$Model->field('id,title,content')->select();
```

这里使用field方法指定了查询的结果集中包含id,title,content三个字段的值。执行的SQL相当于：

```
SELECT id,title,content FROM table
```

可以给某个字段设置别名，例如：

```
$Model->field('id,nickname as name')->select();
```

执行的SQL语句相当于：

```
SELECT id,nickname as name FROM table
```

使用SQL函数

可以在field方法中直接使用函数，例如：

```
$Model->field('id,SUM(score)')->select();
```

执行的SQL相当于：

```
SELECT id,SUM(score) FROM table
```

> 除了select方法之外，所有的查询方法，包括find等都可以使用field方法。

使用数组参数

field方法的参数可以支持数组，例如：

```
$Model->field(array('id','title','content'))->select();
```

最终执行的SQL和前面用字符串方式是等效的。

数组方式的定义可以为某些字段定义别名，例如：

```
$Model->field(array('id','nickname'=>'name'))->select();
```

执行的SQL相当于：

```
SELECT id,nickname as name FROM table
```

对于一些更复杂的字段要求，数组的优势则更加明显，例如：

```
$Model->field(array('id','concat(name,'-',id)'=>'truename','LEFT(title,7)'=>'sub_title'))->select();
```

执行的SQL相当于：

```
SELECT id,concat(name,'-',id) as truename,LEFT(title,7) as sub_title FROM table
```

获取所有字段

如果有一个表有非常多的字段，需要获取所有的字段（这个也许很简单，因为不调用field方法或者直接使用空的field方法都能做到）：

```
$Model->select();
$Model->field()->select();
$Model->field('*')->select();
```

上面三个用法是等效的，都相当于执行SQL：

```
SELECT * FROM table
```

但是这并不是我说的获取所有字段，我希望显式的调用所有字段（对于对性能要求比较高的系统，这个要求并不过分，起码是一个比较好的习惯），那么OK，仍然很简单，下面的用法可以完成预期的作用：

```
$Model->field(true)->select();
```

`field(true)` 的用法会显式的获取数据表的所有字段列表，哪怕你的数据表有100个字段。

字段排除

如果我希望获取排除数据表中的 `content` 字段（文本字段的值非常耗内存）之外的所有字段值，我们就可以使用field方法的排除功能，例如下面的方式就可以实现所说的功能：

```
$Model->field('content',true)->select();
```

则表示获取除了content之外的所有字段，要排除更多的字段也可以：

```
$Model->field('user_id,content',true)->select();
//或者用
$Model->field(array('user_id','content'),true)->select();
```

用于写入

除了查询操作之外，field方法还有一个非常重要的安全功能--字段合法性检测。field方法结合create方法使用就可以完成表单提交的字段合法性检测，如果我们在表单提交的处理方法中使用了：

```
$Model->field('title,email,content')->create();
```

即表示表单中的合法字段只有 title，email 和 content 字段，无论用户通过什么手段更改或者添加了浏览器的提交字段，都会直接屏蔽。因为，其他是所有字段我们都不希望由用户提交来决定，你可以通过自动完成功能定义额外的字段写入。

同样的，field也可以结合add和save方法，进行字段过滤，例如：

```
$Model->field('title,email,content')->save($data);
```

如果data数据中包含有title,email,content之外的字段数据的话，也会过滤掉。

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。

用法如下：

```
$Model->where('status=1')->order('id desc')->limit(5)->select();
```

注意：连贯操作方法没有顺序，可以在select方法调用之前随便改变调用顺序。

支持对多个字段的排序，例如：

```
$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。

如果你的字段和mysql关键字有冲突，那么建议采用数组方式调用，例如：

```
$Model->where('status=1')->order(array('order','id'=>'desc'))->limit(5)->select();
```

LIMIT

limit方法也是模型类的连贯操作方法之一，主要用于指定查询和操作的数量，特别在分页查询的时候使用较多。ThinkPHP的limit方法可以兼容所有的数据库驱动类的。

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('status=1')->field('id,name')->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 = M('User');
$User->where('score=100')->limit(3)->save(array('level'=>'A'));
```

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（可能还取决于where条件和order排序的影响 这个暂且不提）。

你也可以这样使用，作用是一样的：

```
$Article = M('Article');
$Article->limit(10,25)->select();
```

对于大数据表，尽量使用limit限制查询结果，否则会导致很大的内存开销和性能问题。

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。

和limit方法一样，page方法也支持2个参数的写法，例如：

```
$Article->page(1,10)->select();
// 和下面的用法等效
$Article->page('1,10')->select();
```

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();
```

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('user_id,username,max(score)')->group('user_id')->select();
```

生成的SQL语句是：

```
SELECT user_id,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('user_id,test_time,username,max(score)')->group('user_id,test_time')->select();
```

生成的SQL语句是：

```
SELECT user_id,test_time,username,max(score) FROM think_score GROUP BY user_id,test_time
```

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
```

JOIN

JOIN方法也是连贯操作方法之一，用于根据两个或多个表中的列之间的关系，从这些表中查询数据。

join通常有下面几种类型，不同类型的join操作会影响返回的数据结果。

- INNER JOIN: 等同于 JOIN（默认的JOIN类型）,如果表中有至少一个匹配，则返回行

- 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

join方法可以支持以上四种类型，例如：

```
$Model = M('Artist');
$Model
->join('think_work ON think_artist.id = think_work.artist_id')
->join('think_card ON think_artist.card_id = think_card.id')
->select();
```

join方法支持多次调用，但指定的数据表必须是全称，但我们可以这样来定义：

```
$Model
->join('__WORK__ ON __ARTIST__.id = __WORK__.artist_id')
->join('__CARD__ ON __ARTIST__.card_id = __CARD__.id')
->select();
```

`__WORK__`和 `__CARD__`在最终解析的时候会转换为 `think_work`和 `think_card`。

默认采用INNER JOIN 方式，如果需要用其他的JOIN方式，可以改成

```
$Model->join('RIGHT JOIN __WORK__ ON __ARTIST__.id = __WORK__.artist_id')->select();
```

或者使用：

```
$Model->join('__WORK__ ON __ARTIST__.id = __WORK__.artist_id','RIGHT')->select();
```

> join方法的第二个参数支持的类型包括：INNER LEFT RIGHT FULL。

如果join方法的参数用数组的话，只能使用一次join方法，并且不能和字符串方式混合使用。 例如：

```
join(array(' __WORK__ ON __ARTIST__.id = __WORK__.artist_id','__CARD__ ON __ARTIST__.card_id = __C
ARD__.id'))
```

使用数组方式的情况下，第二个参数无效。因此必须在字符串中显式定义join类型，例如：

```
join(array(' LEFT JOIN __WORK__ ON __ARTIST__.id = __WORK__.artist_id','RIGHT JOIN __CARD__ ON __
ARTIST__.card_id = __CARD__.id'))
```

UNION

UNION操作用于合并两个或多个 SELECT 语句的结果集。

使用示例：

```
$Model->field('name')
    ->table('think_user_0')
    ->union('SELECT name FROM think_user_1')
    ->union('SELECT name FROM think_user_2')
    ->select();
```

数组用法：

```
$Model->field('name')
    ->table('think_user_0')
    ->union(array('field'=>'name','table'=>'think_user_1'))
    ->union(array('field'=>'name','table'=>'think_user_2'))
    ->select();
```

或者

```
$Model->field('name')
    ->table('think_user_0')
    ->union(array('SELECT name FROM think_user_1','SELECT name FROM think_user_2'))
    ->select();
```

支持UNION ALL 操作，例如：

```
$Model->field('name')
    ->table('think_user_0')
    ->union('SELECT name FROM think_user_1',true)
    ->union('SELECT name FROM think_user_2',true)
    ->select();
```

或者

```
$Model->field('name')
    ->table('think_user_0')
    ->union(array('SELECT name FROM think_user_1','SELECT name FROM think_user_2'),true)
    ->select();
```

每个union方法相当于一个独立的SELECT语句。

> 注意：UNION 内部的 SELECT 语句必须拥有相同数量的列。列也必须拥有相似的数据类型。同时，每条 SELECT 语句中的列的顺序必须相同。

DISTINCT

DISTINCT 方法用于返回唯一不同的值 。

例如：

```
$Model->distinct(true)->field('name')->select();
```

生成的SQL语句是： `SELECT DISTINCT name FROM think_user`

distinct方法的参数是一个布尔值。

LOCK

Lock方法是用于数据库的锁机制，如果在查询或者执行操作的时候使用：

```
lock(true);
```

就会自动在生成的SQL语句最后加上 `FOR UPDATE` 或者 `FOR UPDATE NOWAIT`（Oracle数据库）。

CACHE

cache方法用于查询缓存操作，也是连贯操作方法之一。

cache可以用于 `select` 、 `find` 和 `getField` 方法，以及其衍生方法，使用cache方法后，在缓存有效期之内不会再次进行数据库查询操作，而是直接获取缓存中的数据，关于数据缓存的类型和设置可以参考缓存部分。

下面举例说明，例如，我们对find方法使用cache方法如下：

```
$Model = M('User');
$Model->where('id=5')->cache(true)->find();
```

第一次查询结果会被缓存，第二次查询相同的数据的时候就会直接返回缓存中的内容，而不需要再次进行数据库查询操作。

默认情况下， 缓存有效期和缓存类型是由DATA_CACHE_TIME和DATA_CACHE_TYPE配置参数决定的，但cache方法可以单独指定，例如：

```
$Model = M('User');
$Model->cache(true,60,'xcache')->find();
```

表示对查询结果使用xcache缓存，缓存有效期60秒。

cache方法可以指定缓存标识：

```
$Model = M('User');
$Model->cache('key',60)->find();
```

> 指定查询缓存的标识可以使得查询缓存更有效率。

这样，在外部就可以通过S方法直接获取查询缓存的数据，例如：

```
$Model = M('User');
$result = $Model->cache('key',60)->find();
$data = S('key');
```

COMMENT

COMMENT方法 用于在生成的SQL语句中添加注释内容，例如：

```
$this->comment('查询考试前十名分数')
->field('username,score')
->limit(10)
->order('score desc')
->select();
```

最终生成的SQL语句是：

```
SELECT username,score FROM think_score ORDER BY score desc LIMIT 10 /* 查询考试前十名分数 */
```

RELATION

USING

fetchSql

fetchSql用于直接返回SQL而不是执行查询，适用于任何的CURD操作方法。 例如：

```
$result = M('User')->fetchSql(true)->find(1);
```

输出result结果为： `SELECT * FROM think_user where id = 1`

TOKEN

token方法可用于临时关闭令牌验证，例如：

```
$model->token(false)->create();
```

即可在提交表单的时候临时关闭令牌验证（即使开启了TOKEN_ON参数）。

STRICT

strict为3.2.3新增连贯操作，用于设置数据写入和查询是否严格检查是否存在字段。默认情况下不合法数据字段自动删除，如果设置了严格检查则会抛出异常。 例如:

```
$model->strict(true)->add($data);
```

INDEX

index方法用于数据集的强制索引操作，例如：

```
$Model->index('user')->select();
```

对查询强制使用user索引，user必须是数据表实际创建的索引名称。

命名范围

在应用开发过程中，使用最多的操作还是数据查询操作，凭借ThinkPHP的连贯操作的特性，可以使得查询操作变得更优雅和清晰，命名范围功能则是给模型操作定义了一系列的封装，让你更方便的操作数据。

命名范围功能的优势在于可以一次定义多次调用，并且在项目中也能起到分工配合的规范，避免开发人员在写CURD操作的时候出现问题，项目经理只需要合理的规划命名范围即可。

定义属性

要使用命名范围功能，主要涉及到模型类的`_scope`属性定义和`scope`连贯操作方法的使用。

我们首先定义_scope属性：

```
namespace Home\Model;
use Think\Model;
class NewsModel extends Model {
    protected $_scope = array(
        // 命名范围normal
        'normal'=>array(
            'where'=>array('status'=>1),
        ),
        // 命名范围latest
        'latest'=>array(
            'order'=>'create_time DESC',
            'limit'=>10,
        ),
    );
}
```

`_scope`属性是一个数组，每个数组项表示定义一个命名范围，命名范围的定义格式为：

```
'命名范围标识'=>array(
    '属性1'=>'值1',
    '属性2'=>'值2',
    ...
)
```

命名范围标识：可以是任意的字符串，用于标识当前定义的命名范围名称。

命名范围支持的属性包括：

属性	描述
where	查询条件
field	查询字段
order	结果排序
table	查询表名

属性	描述
limit	结果限制
page	结果分页
having	having查询
group	group查询
lock	查询锁定
distinct	唯一查询
cache	查询缓存

每个命名范围的定义可以包括这些属性中一个或者多个。

方法调用

属性定义完成后，接下来就是使用`scope`方法进行命名范围的调用了，每调用一个命名范围，就相当于执行了命名范围中定义的相关操作选项对应的连贯操作方法。

调用某个命名范围

最简单的调用方式就直接调用某个命名范围，例如：

```
$Model = D('News'); // 这里必须使用D方法 因为命名范围在模型里面定义
$Model->scope('normal')->select();
$Model->scope('latest')->select();
```

生成的SQL语句分别是：

```
SELECT * FROM think_news WHERE status=1
SELECT * FROM think_news ORDER BY create_time DESC LIMIT 10
```

调用多个命名范围

也可以支持同时调用多个命名范围定义，例如：

```
$Model->scope('normal')->scope('latest')->select();
```

或者简化为：

```
$Model->scope('normal,latest')->select();
```

生成的SQL都是：

```
SELECT * FROM think_news WHERE status=1 ORDER BY create_time DESC LIMIT 10
```

如果两个命名范围的定义存在冲突，则后面调用的命名范围定义会覆盖前面的相同属性的定义。

如果调用的命名范围标识不存在，则会忽略该命名范围，例如：

```
$Model->scope('normal,new')->select();
```

上面的命名范围中new是不存在的，因此只有normal命名范围生效，生成的SQL语句是：

```
SELECT * FROM think_news WHERE status=1
```

默认命名范围

系统支持默认命名范围功能，如果你定义了一个default命名范围，例如：

```
protected $_scope = array(
    // 默认的命名范围
    'default'=>array(
        'where'=>array('status'=>1),
        'limit'=>10,
    ),
);
```

那么调用default命名范围可以直接使用：

```
$Model->scope()->select();
```

而无需再传入命名范围标识名

```
$Model->scope('default')->select();
```

虽然这两种方式是等效的。

命名范围调整

如果你需要在normal命名范围的基础上增加额外的调整，可以使用：

```
$Model->scope('normal',array('limit'=>5))->select();
```

生成的SQL语句是：

```
SELECT * FROM think_news WHERE status=1 LIMIT 5
```

当然，也可以在两个命名范围的基础上进行调整，例如：

```
$Model->scope('normal,latest',array('limit'=>5))->select();
```

生成的SQL是：

```
SELECT * FROM think_news WHERE status=1 ORDER BY create_time DESC LIMIT 5
```

自定义命名范围

又或者，干脆不用任何现有的命名范围，我直接传入一个命名范围：

```
$Model->scope(array('field'=>'id,title','limit'=>5,'where'=>'status=1','order'=>'create_time DESC'))->
select();
```

这样，生成的SQL变成：

```
SELECT id,title FROM think_news WHERE status=1 ORDER BY create_time DESC LIMIT 5
```

与连贯操作混合使用

命名范围一样可以和之前的连贯操作混合使用，例如定义了命名范围_scope属性：

```
protected $_scope = array(
    'normal'=>array(
        'where'=>array('status'=>1),
        'field'=>'id,title',
        'limit'=>10,
    ),
);
```

然后在使用的时候，可以这样调用：

```
$Model->scope('normal')->limit(8)->order('id desc')->select();
```

这样，生成的SQL变成：

```
SELECT id,title FROM think_news WHERE status=1 ORDER BY id desc LIMIT 8
```

如果定义的命名范围和连贯操作的属性有冲突，则后面调用的会覆盖前面的。

如果是这样调用：

```
$Model->limit(8)->scope('normal')->order('id desc')->select();
```

生成的SQL则是：

```
SELECT id,title FROM think_news WHERE status=1 ORDER BY id desc LIMIT 10
```

动态调用

除了采用scope方法调用命名范围外，我们还支持直接调用命名范围名称的方式来动态调用，例如：

```
$Model->scope('normal',array('limit'=>5))->select();
```

查询操作也可以采用：

```
$Model->normal(array('limit'=>5))->select();
```

的方式调用。 `normal(array('limit'=>5))` 表示调用normal命名范围，并且传入额外的 `array('limit'=>5)` 参数。

由于采用的是 `__call` 魔术方法机制，因此这样调用的前提是你定义的命名范围名称没有和现有操作方法冲突。

CURD操作

ThinkPHP提供了灵活和方便的数据操作方法，对数据库操作的四个基本操作（CURD）：创建、更新、读取和删除的实现是最基本的，也是必须掌握的，在这基础之上才能熟悉更多实用的数据操作方法。

CURD操作通常是可以和连贯操作配合完成的。

数据创建

在进行数据操作之前，我们往往需要手动创建需要的数据，例如对于提交的表单数据：

```
// 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
//……
```

创建数据对象

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。

很简单的例子：

```
// 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();
```

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);
```

甚至还可以支持从对象创建新的数据对象

```
// 从User数据对象创建新的Member数据对象
$User = stdClass();
$User->name = 'ThinkPHP';
$User->email = 'ThinkPHP@gmail.com';
$Member = M("Member");
$Member->create($User);
```

创建完成的数据可以直接读取和修改，例如：

```
$data['name'] = 'ThinkPHP';
$data['email'] = 'ThinkPHP@gmail.com';
$User->create($data);
// 创建完成数据对象后可以直接读取数据
echo $User->name;
echo $User->email;
// 也可以直接修改创建完成的数据
$User->name = 'onethink'; // 修改name字段数据
$User->status = 1; // 增加新的字段数据
```

数据操作状态

create方法的第二个参数可以指定创建数据的操作状态，默认情况下是自动判断是写入还是更新操作。

也可以显式指定操作状态，例如：

```
$Member = M("User");
// 指定更新数据操作状态
$Member->create($_POST,Model::MODEL_UPDATE);
```

系统内置的数据操作包括 Model::MODEL_INSERT （或者1）和 Model::MODEL_UPDATE （或者2），当没有指定的时候，系统根据数据源是否包含主键数据来自动判断，如果存在主键数据，就当成 Model::MODEL_UPDATE 操作。

不同的数据操作状态可以定义不同的数据验证和自动完成机制，所以，你可以自定义自己需要的数据操作状态，例如，可以设置登录操作的数据状态（假设为3）：

```
$Member = M("User");
// 指定更新数据操作状态
$Member->create($_POST,3);
```

事实上，create方法所做的工作远非这么简单，在创建数据对象的同时，完成了一系列的工作，我们来看下create方法的工作流程就能明白：

步骤	说明	返回
1	获取数据源（默认是POST数组）	
2	验证数据源合法性（非数组或者对象会过滤）	失败则返回false
3	检查字段映射	
4	判断数据状态（新增或者编辑，指定或者自动判断）	
5	数据自动验证	失败则返回false
6	表单令牌验证	失败则返回false
7	表单数据赋值（过滤非法字段和字符串处理）	
8	数据自动完成	
9	生成数据对象（保存在内存）	

因此，我们熟悉的令牌验证、自动验证和自动完成功能，其实都必须通过create方法才能生效。

如果没有定义自动验证的话，create方法的返回值是创建完成的数据对象数组，例如：

```
$data['name'] = 'thinkphp';
$data['email'] = 'thinkphp@gmail.com';
$data['status'] = 1;
$User = M('User');
$data = $User->create($data);
dump($data);
```

输出结果为：

```
array (size=3)
  'name' => string 'thinkphp' (length=8)
  'email' => string 'thinkphp@gmail.com' (length=18)
  'status'=> int 1
```

Create方法创建的数据对象是保存在内存中，并没有实际写入到数据库中，直到使用 add 或者 save 方法才会真正写入数据库。

因此在没有调用add或者save方法之前，我们都可以改变create方法创建的数据对象，例如：

```
$User = M('User');
$User->create(); //创建User数据对象
$User->status = 1; // 设置默认的用户状态
$User->create_time = time(); // 设置用户的创建时间
$User->add(); // 把用户对象写入数据库
```

如果只是想简单创建一个数据对象，并不需要完成一些额外的功能的话，可以使用data方法简单的创建数据对象。 使用如下：

```
// 实例化User模型
$User = M('User');
// 创建数据后写入到数据库
$data['name'] = 'ThinkPHP';
$data['email'] = 'ThinkPHP@gmail.com';
$User->data($data)->add();
```

Data方法也支持传入数组和对象，使用data方法创建的数据对象不会进行自动验证和过滤操作，请自行处理。但在进行add或者save操作的时候，数据表中不存在的字段以及非法的数据类型（例如对象、数组等非标量数据）是会自动过滤的，不用担心非数据表字段的写入导致SQL错误的问题。

支持的连贯操作

在执行create方法之前，我们可以调用相关的连贯操作方法，配合完成数据创建操作。

create方法支持的连贯操作方法包括：

连贯操作	作用	支持的参数类型
field	用于定义合法的字段	字符串和数组
validate	用于数据自动验证	数组
auto	用于数据自动完成	数组
token	用于令牌验证	布尔值

更多的用法参考后续的内容。

字段合法性过滤

如果在create方法之前调用field方法，则表示只允许创建指定的字段数据，其他非法字段将会被过滤，例如：

```
$data['name'] = 'thinkphp';
$data['email'] = 'thinkphp@gmail.com';
$data['status'] = 1;
$data['test'] = 'test';
$User = M('User');
$data = $User->field('name,email')->create($data);
dump($data);
```

输出结果为：

```
array (size=2)
  'name' => string 'thinkphp' (length=8)
  'email' => string 'thinkphp@gmail.com' (length=18)
```

最终只有 `name` 和 `email` 字段的数据被允许写入， `status` 和 `test` 字段直接被过滤了，哪怕status也是数据表中的合法字段。

如果我们有自定义模型类，对于数据新增和编辑操作的话，我们还可以直接在模型类里面通过设置 `insertFields` 和 `updateFields` 属性来定义允许的字段，例如：

```
namespace Home\Model;
use Think\Model;
class UserModel extends Model{
    protected $insertFields = 'name,email'; // 新增数据的时候允许写入name和email字段
    protected $updateFields = 'email'; // 编辑数据的时候只允许写入email字段
}
```

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"); // 实例化User对象
$data['name'] = 'ThinkPHP';
$data['email'] = 'ThinkPHP@gmail.com';
$User->add($data);
```

如果是Mysql数据库的话，还可以支持在数据插入时允许更新操作:

```
add($data='',$options=array(),$replace=false)
```

其中add方法增加$replace参数(是否添加数据时允许覆盖)，true表示覆盖，默认为false

或者使用data方法连贯操作

```
$User = M("User"); // 实例化User对象
$User->data($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：

```
$User = M("User"); // 实例化User对象
// 根据表单提交的POST数据创建数据对象
if($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
        $insertId = $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。

支持的连贯操作

在执行add方法之前，我们可以调用相关的连贯操作方法，配合完成数据写入操作。

写入操作支持的连贯操作方法包括：

连贯操作	作用	支持的参数类型
table	用于定义要操作的数据表名称	字符串和数组
data	用于指定要写入的数据对象	数组和对象
field	用于定义要写入的字段	字符串和数组

连贯操作	作用	支持的参数类型
relation	用于关联查询（需要关联模型支持）	字符串
validate	用于数据自动验证	数组
auto	用于数据自动完成	数组
filter	用于数据过滤	字符串
scope	用于命名范围	字符串、数组
bind	用于数据绑定操作	数组
token	用于令牌验证	布尔值
comment	用于SQL注释	字符串
fetchSql	不执行SQL而只是返回SQL	布尔值

可以支持不执行SQL而只是返回SQL语句，例如：

```
$User = M("User"); // 实例化User对象
$data['name'] = 'ThinkPHP';
$data['email'] = 'ThinkPHP@gmail.com';
$sql = $User->fetchSql(true)->add($data);
echo $sql;
// 输出结果类似于
// INSERT INTO think_user (name,email) VALUES ('ThinkPHP','ThinkPHP@gmail.com')
```

字段过滤

如果写入了数据表中不存在的字段数据，则会被直接过滤，例如：

```
$data['name'] = 'thinkphp';
$data['email'] = 'thinkphp@gmail.com';
$data['test'] = 'test';
$User = M('User');
$User->data($data)->add();
```

其中test字段是不存在的，所以写入数据的时候会自动过滤掉。

> 在3.2.2版本以上，如果开启调试模式的话，则会抛出异常，提示：`非法数据对象：[test=>test]`

如果在add方法之前调用field方法，则表示只允许写入指定的字段数据，其他非法字段将会被过滤，例如：

```
$data['name'] = 'thinkphp';
$data['email'] = 'thinkphp@gmail.com';
$data['test'] = 'test';
$User = M('User');
$User->field('name')->data($data)->add();
```

最终只有name字段的数据被允许写入，email和test字段直接被过滤了，哪怕email也是数据表中的合法字段。

字段内容过滤

通过filter方法可以对数据的值进行过滤处理，例如：

```
$data['name'] = '<b>thinkphp</b>';
$data['email'] = 'thinkphp@gmail.com';
$User = M('User');
$User->data($data)->filter('strip_tags')->add();
```

写入数据库的时候会把name字段的值转化为 `thinkphp` 。

> filter方法的参数是一个回调类型，支持函数或者闭包定义。

批量写入

在某些情况下可以支持数据的批量写入，例如：

```
// 批量添加数据
$dataList[] = array('name'=>'thinkphp','email'=>'thinkphp@gamil.com');
$dataList[] = array('name'=>'onethink','email'=>'onethink@gamil.com');
$User->addAll($dataList);
```

> 该功能需要3.2.3以上版本，3.2.3以下版本仅对mysql数据库支持

数据读取

在ThinkPHP中读取数据的方式很多，通常分为读取数据、读取数据集和读取字段值。

数据查询方法支持的连贯操作方法有：

连贯操作	作用	支持的参数类型
where	用于查询或者更新条件的定义	字符串、数组和对象
table	用于定义要操作的数据表名称	字符串和数组

连贯操作	作用	支持的参数类型
alias	用于给当前数据表定义别名	字符串
field	用于定义要查询的字段（支持字段排除）	字符串和数组
order	用于对结果排序	字符串和数组
group	用于对查询的group支持	字符串
having	用于对查询的having支持	字符串
join	用于对查询的join支持	字符串和数组
union	用于对查询的union支持	字符串、数组和对象
distinct	用于查询的distinct支持	布尔值
lock	用于数据库的锁机制	布尔值
cache	用于查询缓存	支持多个参数
relation	用于关联查询（需要关联模型支持）	字符串
result	用于返回数据转换	字符串
scope	用于命名范围	字符串、数组
bind	用于数据绑定操作	数组
comment	用于SQL注释	字符串
fetchSql	不执行SQL而只是返回SQL	布尔值

> 注意：某些情况下有些连贯操作是无效的，例如limit方法对find方法是无效的。

读取数据

读取数据是指读取数据表中的一行数据（或者关联数据），主要通过 find 方法完成，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 = $User->where('status=1 AND name="thinkphp"')->find();
dump($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关联数组（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());
```

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。

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。

示例如下：

```
$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：

```
$User->getField('id',true); // 获取id数组
//返回数据格式如array(1,2,3,4,5)一维数组，其中value就是id列的每行的值
```

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');
//两个字段的情况下返回的是array(`id`=>`nickname`)的关联数组，以id的值为key，nickname字段值为value
```

这样返回的list是一个数组，键名是用户的id字段的值，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');
//返回的数组格式是array(`id`=>array(`id`=>value,`nickname`=>value,`email`=>value))是一个二维数组，key还是id字段的值，但value是整行的array数组，类似于select()方法的结果遍历将id的值设为数组key
```

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
```

可以配合使用order方法使用。更多的查询方法可以参考查询语言章节。

数据更新

ThinkPHP的数据更新操作包括更新数据和更新字段方法。

更新数据

更新数据使用 `save` 方法，例如：

```
$User = M("User"); // 实例化User对象
// 要修改的数据对象属性赋值
$data['name'] = 'ThinkPHP';
$data['email'] = 'ThinkPHP@gmail.com';
$User->where('id=5')->save($data); // 根据条件更新记录
```

也可以改成对象方式来操作：

```
$User = M("User"); // 实例化User对象
// 要修改的数据对象属性赋值
$User->name = 'ThinkPHP';
$User->email = 'ThinkPHP@gmail.com';
$User->where('id=5')->save(); // 根据条件更新记录
```

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。

因此下面的代码不会更改数据库的任何记录

```
$User->save($data);
```

除非使用下面的方式：

```
$User = M("User"); // 实例化User对象
// 要修改的数据对象属性赋值
$data['id'] = 5;
$data['name'] = 'ThinkPHP';
$data['email'] = 'ThinkPHP@gmail.com';
$User->save($data); // 根据条件保存修改的数据
```

如果id是数据表的主键的话，系统自动会把主键的值作为更新条件来更新其他字段的值。

数据更新方法支持的连贯操作方法有：

连贯操作	作用	支持的参数类型
where	用于查询或者更新条件的定义	字符串、数组和对象
table	用于定义要操作的数据表名称	字符串和数组
alias	用于给当前数据表定义别名	字符串
field	用于定义允许更新的字段	字符串和数组
order	用于对数据排序	字符串和数组
lock	用于数据库的锁机制	布尔值
relation	用于关联更新（需要关联模型支持）	字符串
scope	用于命名范围	字符串、数组

连贯操作	作用	支持的参数类型
bind	用于数据绑定操作	数组
comment	用于SQL注释	字符串
fetchSql	不执行SQL而只是返回SQL	布尔值

和add方法一样，save方法支持使用 field 方法过滤字段和 filter 方法过滤数据，例如：

```
$User = M("User"); // 实例化User对象
// 要修改的数据对象属性赋值
$data['name'] = 'test';
$data['email'] = '<b>test@gmail.com</b>';
$User->where('id=5')->field('email')->filter('strip_tags')->save($data); // 根据条件保存修改的数据
```

当使用field('email')的时候，只允许更新email字段的值（采用strip_tags方法过滤），name字段的值将不会被修改。

还有一种方法是通过create或者data方法创建要更新的数据对象，然后进行保存操作，这样save方法的参数可以不需要传入。

```
$User = M("User"); // 实例化User对象
// 要修改的数据对象属性赋值
$data['name'] = 'ThinkPHP';
$data['email'] = 'ThinkPHP@gmail.com';
$User->where('id=5')->data($data)->save(); // 根据条件保存修改的数据
```

使用create方法的例子：

```
$User = M("User"); // 实例化User对象
// 根据表单提交的POST数据创建数据对象
$User->create();
$User->save(); // 根据条件保存修改的数据
```

更新字段

如果只是更新个别字段的值，可以使用 setField 方法。

使用示例：

```
$User = M("User"); // 实例化User对象
// 更改用户的name值
$User-> where('id=5')->setField('name','ThinkPHP');
```

setField方法支持同时更新多个字段，只需要传入数组即可，例如：

```
$User = M("User"); // 实例化User对象
// 更改用户的name和email的值
$data = array('name'=>'ThinkPHP','email'=>'ThinkPHP@gmail.com');
$User-> where('id=5')->setField($data);
```

而对于统计字段（通常指的是数字类型）的更新，系统还提供了 setInc 和 setDec 方法。

```
$User = M("User"); // 实例化User对象
$User->where('id=5')->setInc('score',3); // 用户的积分加3
$User->where('id=5')->setInc('score'); // 用户的积分加1
$User->where('id=5')->setDec('score',5); // 用户的积分减5
$User->where('id=5')->setDec('score'); // 用户的积分减1
```

3.2.3版本开始，setInc和setDec方法支持延迟更新，用法如下：

```
$Article = M("Article"); // 实例化Article对象
$Article->where('id=5')->setInc('view',1); // 文章阅读数加1
$Article->where('id=5')->setInc('view',1,60); // 文章阅读数加1，并且延迟60秒更新（写入）
```

数据删除

ThinkPHP删除数据使用delete方法，例如：

```
$Form = M('Form');
$Form->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"); // 实例化User对象
$User->where('id=5')->delete(); // 删除id为5的用户数据
$User->delete('1,2,5'); // 删除主键为1,2和5的用户数据
$User->where('status=0')->delete(); // 删除所有状态为0的用户数据
```

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')->order('create_time')->limit('5')->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();
```

数据删除方法支持的连贯操作方法有：

连贯操作	作用	支持的参数类型
where	用于查询或者更新条件的定义	字符串、数组和对象
table	用于定义要操作的数据表名称	字符串和数组
alias	用于给当前数据表定义别名	字符串
order	用于对数据排序	字符串和数组
lock	用于数据库的锁机制	布尔值
relation	用于关联删除（需要关联模型支持）	字符串
scope	用于命名范围	字符串、数组
bind	用于数据绑定操作	数组
comment	用于SQL注释	字符串
fetchSql	不执行SQL而只是返回SQL	布尔值

ActiveRecord

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，记录映射到对象。最大的特点就是使用方便和便于理解（因为采用了对象化），提供了开发的最佳体验，从而达到敏捷开发的目的。

下面我们用AR模式来换一种方式重新完成CURD操作。

创建数据

```
$User = M("User"); // 实例化User对象
// 然后直接给数据对象赋值
$User->name = 'ThinkPHP';
$User->email = 'ThinkPHP@gmail.com';
// 把数据对象添加到数据库
$User->add();
```

如果使用了create方法创建数据对象的话，仍然可以在创建完成后进行赋值

```
$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
// 增加或者更改其中的属性
$User->status = 1;
$User->create_time = time();
// 把数据对象添加到数据库
$User->add();
```

查询记录

AR模式的数据查询比较简单，因为更多情况下面查询条件都是以主键或者某个关键的字段。这种类型的查询，ThinkPHP有着很好的支持。 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，如果按照之前的方式，我们可能会使用下面的方法：

```
$User = M("User"); // 实例化User对象
// 查找id为8的用户数据
$User->where('id=8')->find();
```

用AR模式的话可以直接写成：

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

如果要根据某个字段查询，例如查询姓名为ThinkPHP的可以用：

```
$User = M("User"); // 实例化User对象
$User->getByName("ThinkPHP");
```

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：

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

如果要查询数据集，可以直接使用：

```
// 查找主键为1、3、8的多个数据
$userList = $User->select('1,3,8');
```

更新记录

在完成查询后，可以直接修改数据对象然后保存到数据库。

```
$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
```

上面这种方式仅仅是示例，不代表保存操作之前一定要先查询。因为下面的方式其实是等效的：

```
$User->id = 1;
$User->name = 'TOPThink'; // 修改数据对象
$User->save(); // 保存当前数据对象
```

删除记录

可以删除当前查询的数据对象

```
$User->find(2);
$User->delete(); // 删除当前的数据对象
```

或者直接根据主键进行删除

```
$User->delete(8); // 删除主键为8的数据
$User->delete('5,6'); // 删除主键为5、6的多个数据
```

字段映射

ThinkPHP的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，假设我们的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映射到数据表的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会自动处理，

```
// 实例化User模型
$User = D('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。

查询语言

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数组或者对象来作为查询条件，因为会更加安全。

使用字符串作为查询条件

这是最传统的方式，但是安全性不高，例如：

```
$User = M("User"); // 实例化User对象
$User->where('type=1 AND status=1')->select();
```

最后生成的SQL语句是

```
SELECT * FROM think_user WHERE type=1 AND status=1
```

> 采用字符串查询的时候，我们可以配合使用字符串条件的安全预处理机制。

使用数组作为查询条件

这种方式是最常用的查询方式，例如：

```
$User = M("User"); // 实例化User对象
$condition['name'] = 'thinkphp';
$condition['status'] = 1;
// 把查询条件传入查询方法
$User->where($condition)->select();
```

最后生成的SQL语句是

```
SELECT * FROM think_user WHERE `name`='thinkphp' AND status=1
```

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"); // 实例化User对象
$condition['name'] = 'thinkphp';
$condition['account'] = 'thinkphp';
$condition['_logic'] = 'OR';
// 把查询条件传入查询方法
$User->where($condition)->select();
```

最后生成的SQL语句是

```
SELECT * FROM think_user WHERE `name`='thinkphp' OR `account`='thinkphp'
```

使用对象方式来查询

这里以stdClass内置对象为例：

```
$User = M("User"); // 实例化User对象
// 定义查询条件
$condition = new stdClass();
$condition->name = 'thinkphp';
$condition->status= 1;
$User->where($condition)->select();
```

最后生成的SQL语句和上面一样

```
SELECT * FROM think_user WHERE `name`='thinkphp' AND status=1
```

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。

在使用数组和对象方式查询的时候，如果传入了不存在的查询字段是会被自动过滤的，例如：

```
$User = M("User"); // 实例化User对象
$condition['name'] = 'thinkphp';
$condition['status'] = 1;
$condition['test'] = 'test';
// 把查询条件传入查询方法
$User->where($condition)->select();
```

因为数据库的test字段是不存在的，所以系统会自动检测并过滤掉 $condition['test'] = 'test' 这一查询条件。

> 如果是3.2.2版本以上，当开启调试模式的话，则会抛出异常，显示：错误的查询条件 。

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');
```

表达式不分大小写，支持的查询表达式有下面几种，分别表示的含义是：

表达式	含义	协助记忆
EQ	等于（=）	equal
NEQ	不等于（<>）	not equal
GT	大于（>）	greater
EGT	大于等于（>=）	equal or greater
LT	小于（<）	less than
ELT	小于等于（<=）	equal or less than
LIKE	模糊查询	
[NOT] BETWEEN	（不在）区间查询	
[NOT] IN	（不在）IN 查询	
EXP	表达式查询，支持SQL语法	expression

表达式查询的用法示例如下：

EQ ：等于（=）

例如：

```
$map['id']  = array('eq',100);
```

和下面的查询等效

```
$map['id']  = 100;
```

表示的查询条件就是 `id = 100`

NEQ：不等于（<>）

例如：

```
$map['id']  = array('neq',100);
```

表示的查询条件就是 `id <> 100`

GT：大于（>）

例如：

```
$map['id']  = array('gt',100);
```

表示的查询条件就是 `id > 100`

EGT：大于等于（>=）

例如：

```
$map['id']  = array('egt',100);
```

表示的查询条件就是 `id >= 100`

LT：小于（<）

例如：

```
$map['id']  = array('lt',100);
```

表示的查询条件就是 `id < 100`

ELT：小于等于（<=）

例如：

```
$map['id']  = array('elt',100);
```

表示的查询条件就是 id <= 100

[NOT] LIKE： 同sql的LIKE

例如：

```
$map['name'] = array('like','thinkphp%');
```

查询条件就变成 name like 'thinkphp%' 如果配置了DB_LIKE_FIELDS参数的话，某些字段也会自动进行模糊查询。例如设置了：

```
'DB_LIKE_FIELDS'=>'title|content'
```

的话，使用

```
$map['title'] = 'thinkphp';
```

查询条件就会变成 title like '%thinkphp%' 支持数组方式，例如

```
$map['a'] =array('like',array('%thinkphp%','%tp'),'OR');
$map['b'] =array('notlike',array('%thinkphp%','%tp'),'AND');
```

生成的查询条件就是：

```
(a like '%thinkphp%' OR a like '%tp') AND (b not like '%thinkphp%' AND b not like '%tp')
```

[NOT] BETWEEN ：同sql的[not] between

查询条件支持字符串或者数组，例如：

```
$map['id']  = array('between','1,8');
```

和下面的等效：

```
$map['id']  = array('between',array('1','8'));
```

查询条件就变成 id BETWEEN 1 AND 8

[NOT] IN： 同sql的[not] in

查询条件支持字符串或者数组，例如：

```
$map['id']  = array('not in','1,5,8');
```

和下面的等效：

```
$map['id']  = array('not in',array('1','5','8'));
```

查询条件就变成 `id NOT IN (1,5, 8)`

EXP：表达式

支持更复杂的查询情况 例如：

```
$map['id']  = array('in','1,3,8');
```

可以改成：

```
$map['id']  = array('exp',' IN (1,3,8) ');
```

exp查询的条件不会被当成字符串，所以后面的查询条件可以使用任何SQL支持的语法，包括使用函数和字段名称。查询表达式不仅可用于查询条件，也可以用于数据更新，例如：

```
$User = M("User"); // 实例化User对象
// 要修改的数据对象属性赋值
$data['name'] = 'ThinkPHP';
$data['score'] = array('exp','score+1');// 用户的积分加1
$User->where('id=5')->save($data); // 根据条件保存修改的数据
```

快捷查询

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

不同字段相同的查询条件

```
$User = M("User"); // 实例化User对象
$map['name|title'] = 'thinkphp';
// 把查询条件传入查询方法
$User->where($map)->select();
```

上面的查询其实可以等效于

```
$User = M("User"); // 实例化User对象
$map['name'] = 'thinkphp';
$map['title'] = 'thinkphp';
$map['_logic'] = 'OR';
// 把查询条件传入查询方法
$User->where($map)->select();
```

查询条件就变成 `name= 'thinkphp' OR title = 'thinkphp'`

不同字段不同的查询条件

```
$User = M("User"); // 实例化User对象
$map['status&title'] =array('1','thinkphp','_multi'=>true);
// 把查询条件传入查询方法
$User->where($map)->select();
```

上面的查询等效于：

```
$User = M("User"); // 实例化User对象
$map['status'] = 1;
$map['title']  = 'thinkphp';
// 把查询条件传入查询方法
$User->where($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'`

，查询字段支持更多的，例如：

```
$map['status&score&title'] =array('1',array('gt','0'),'thinkphp','_multi'=>true);
```

等效于：

```
$map['status'] = 1;
$map['score'] = array('gt',0);
$map['title'] = 'thinkphp';
```

查询条件就变成 `status= 1 AND score >0 AND  title = 'thinkphp'`

> 注意：快捷查询方式中“|”和“&”不能同时使用。

区间查询

ThinkPHP支持对某个字段的区间查询，例如：

```
$map['id'] = array(array('gt',1),array('lt',10)) ;
```

得到的查询条件是：`( id > 1) AND ( id < 10)`

```
$map['id'] = array(array('gt',3),array('lt',10), 'or') ;
```

得到的查询条件是：`( id > 3) OR ( id < 10)`

```
$map['id']  = array(array('neq',6),array('gt',3),'and');
```

得到的查询条件是：`( id != 6) AND ( id > 3)`

> 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：

```
$map['name']  = array(array('like','%a%'), array('like','%b%'), array('like','%c%'), 'ThinkPHP','or');
```

最后的查询条件是：
`( name LIKE '%a%') OR ( name LIKE '%b%') OR ( name LIKE '%c%') OR ( name = 'ThinkPHP')`

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。

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"); // 实例化User对象
$map['id'] = array('neq',1);
$map['name'] = 'ok';
$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 )
```

请求字符串查询方式

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')
```

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：

```
$where['name']  = array('like', '%thinkphp%');
$where['title']  = array('like','%thinkphp%');
$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。

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）

用法示例：

```
$User = M("User"); // 实例化User对象
```

获取用户数：

```
$userCount = $User->count();
```

或者根据字段统计：

```
$userCount = $User->count("id");
```

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->where('score>0')->min('score');
```

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');
```

并且所有的统计查询均支持连贯操作的使用。

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：

QUERY方法

query方法用于执行SQL查询操作，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 \Think\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。

可以在query方法中使用表名的简化写法，便于动态更改表前缀，例如：

```
$Model = new \Think\Model() // 实例化一个model对象 没有对应任何数据表
$Model->query("select * from __PREFIX__user where status=1");
// 3.2.2版本以上还可以直接使用
$Model->query("select * from __USER__ where status=1");
```

和上面的写法等效，会自动读取当前设置的表前缀。

EXECUTE方法

execute用于更新和写入数据的sql操作，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 \Think\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。

也可以在execute方法中使用表名的简化写法，便于动态更改表前缀，例如：

```
$Model = new \Think\Model() // 实例化一个model对象 没有对应任何数据表
$Model->execute("update __PREFIX__user set name='thinkPHP' where status=1");
// 3.2.2版本以上还可以直接使用
$Model->execute("update __USER__ set name='thinkPHP' where status=1");
```

和上面的写法等效，会自动读取当前设置的表前缀。

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

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');
$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。

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。

子查询

从3.0版本开始新增了子查询支持，有两种使用方式：

1、使用select方法 当select方法的参数为false的时候，表示不进行查询只是返回构建SQL，例如：

```
// 首先构造子查询SQL
$subQuery = $model->field('id,name')->table('tablename')->group('field')->where($where)->order('status')->select(false);
```

当select方法传入false参数的时候，表示不执行当前查询，而只是生成查询SQL。

2、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。

自动验证

自动验证是ThinkPHP模型层提供的一种数据验证方法，可以在使用create创建数据对象的时候自动进行数据验证。

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：

1. 静态方式：在模型类里面通过$_validate属性定义验证规则。
2. 动态方式：使用模型类的validate方法动态创建自动验证规则。

无论是什么方式，验证规则的定义是统一的规则，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]),
    ......
);
```

说明

验证字段 （必须）

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，验证字段是可以随意设置的，例如expire有效期规则是和表单字段无关的。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。

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。

提示信息 （必须）

用于验证失败后的提示信息定义

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

附加规则 （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）

规则	说明
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

- 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。

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！'), //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
   );
}
```

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：

```
$User = D("User"); // 实例化User对象
if (!$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"); // 实例化User对象
$data = getData(); // 通过getData方法获取数据源的（数组）数据
if (!$User->create($data)){
    // 对data数据进行验证
    exit($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 extends Model{
  protected $_validate = array(
    array('verify','require','验证码必须！'),  // 都有时间都验证
    array('name','checkName','帐号错误！',1,'function',4),  // 只在登录时候验证
    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

```
$User = D("User"); // 实例化User对象
if (!$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
}
```

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

错误信息多语言支持

如果你希望支持多语言的错误信息提示，那么可以在验证规则里面如下定义：

```
protected $_validate = array(
    array('verify','require','{%VERIFY_CODE_MUST}'),
    array('name','','{%ACCOUNT_EXISTS}',0,'unique',1),
);
```

其中 VERIFY_CODE_MUST 和 ACCOUNT_EXISTS 是我们在语言包里面定义的多语言变量。

如果是采用动态验证方式，则比较简单，直接在定义验证规则的时候使用L方法即可，例如：

```
$rules = array(
    array('verify','require',L('VERIFY_CODE_MUST')),
    array('name','',L('ACCOUNT_EXISTS'),0,'unique',1),
);
```

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：

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

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，在ThinkPHP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或者save方法实现数据写入。

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。

1. 静态方式：在模型类里面通过$_auto属性定义处理规则。
2. 动态方式：使用模型类的auto方法动态创建自动处理规则。

两种方式的定义规则都采用：

```
array(
    array(完成字段1,完成规则,[完成条件,附加规则]),
    array(完成字段2,完成规则,[完成条件,附加规则]),
    ......
);
```

说明

完成字段（必须）

需要进行处理的数据表实际字段名称。

完成规则（必须）

需要处理的规则，配合附加规则完成。

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

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）

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，我们在模型类定义 _auto 属性：

```
namespace Home\Model;
use Think\Model;
class UserModel extends Model{
    protected $_auto = 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('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：

```
$User = D("User"); // 实例化User对象
if (!$User->create()){ // 创建数据对象
    // 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(); // 生成数据对象
$User->add(); // 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。

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();
$User->add();
```

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。

参数绑定

参数绑定是指绑定一个参数到预处理的SQL语句中的对应命名占位符或问号占位符指定的变量，并且可以提高SQL处理的效率。

手动绑定

参数手动绑定需要调用连贯操作的bind方法，例如：

```
$Model = M('User');
$where['name'] = ':name';
$list = $Model->where($where)->bind(':name',I('name'))->select();
```

目前不支持 ？方式 进行占位符，统一使用 :var 方式进行占位符，驱动内部会自动进行处理。

> 参数绑定的参数可以是条件或者要data数据中的参数，CURD操作均可以支持参数绑定bind方法。

可以支持指定绑定变量的类型参数，例如：

```
$Model = M('User');
$where['id'] = ':id';
$list = $Model->where($where)->bind(':id',I('id'),\PDO::PARAM_INT)->select();
```

也可以批量绑定，例如：

```
$Model = M('User');
$where['id']    =    ':id';
$where['name']  =    ':name';
$bind[':id']    =    array(I('id'),\PDO::PARAM_INT);
$bind[':name']  =    array(I('name'),\PDO::PARAM_STR);
$list = $Model->where($where)->bind($bind)->select();
```

自动绑定

对于某些操作的情况（例如模型的写入和更新方法），采用了参数的自动绑定，例如我们在使用

```
$Model = M('User');
$Model->name = 'thinkphp';
$Model->email = 'thinkphp@qq.com';
$Model->add();
```

会自动对写入的数据进行参数绑定操作。其操作等效于：

```
$Model = M('User');
$Model->name = ':name';
$Model->email = ':email';
$bind[':name']    = 'thinkphp';
$bind[':email']   = 'thinkphp@qq.com';
$Model->bind($bind)->add();
```

> 自动绑定不支持参数类型等额外设置，如果有必要请使用上面的手动绑定方式。

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。有些时候，我们建立模型类但又不需要

进行数据库操作，仅仅是借助模型类来封装一些业务逻辑，那么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：

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。

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。

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、服务层等不同的模型层，我们可以在模块目录下面创建 Model 、 Logic 和 Service 目录，把对用户表的所有模型操作分成三层：

- 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 用于定义用户相关的服务接口等

三个模型层的定义如下：

Model类：Home\Model\UserModel.class.php

```
namespace Home\Model;
class UserModel extends \Think\Model{

}
```

实例化方法： `D('User');`

Logic类：Home\Logic\UserLogic.class.php

```
namespace Home\Logic;
class UserLogic extends \Think\Model{

}
```

实例化方法： `D('User','Logic');`

Api类：Home\Api\UserApi.class.php

```
namespace Home\Api;
class UserApi extends \Think\Model{

}
```

实例化方法： `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。

视图模型

视图定义

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'),
   );
 }
```

我们来解释一下定义的格式代表了什么。

`$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。

例如：

```
'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_user"
// 或者使用简化定义（自动获取表前缀）
// '_table'=>"__USER__"
```

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 JOI
N 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.cate
gory_id'),
```

视图查询

接下来，我们就可以和使用普通模型一样对视图模型进行操作了 。

```
$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。

```
$Model = D("Blog");
$Model->table('think_blog Blog,think_category Category,think_user User')
 ->field('Blog.id,Blog.name,Blog.title,Category.title as category_name,User.name as username')
 ->order('Blog.id desc')
 ->where('Blog.category_id=Category.id AND Blog.user_id=User.id')
 ->select();
```

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。

关联模型

关联关系

通常我们所说的关联关系包括下面三种：

```
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_MANY 和 BELONGS_TO
多对多关联 ：MANY_TO_MANY
```

关联关系必然有一个参照表，例如：

- 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。

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，所有的关联定义都统一在模型类的$_link 成员变量里面定义，并且可以支持动态定义。要支持关联操作，模型类必须继承`Think\Model\RelationModel`类，关联定义的格式是：

```
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
        ...
    );
}
```

下面我们首先来分析下各个关联方式的定义：

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',
            // 定义更多的关联属性
            ......
            ),
        );
}
```

关联HAS_ONE支持的关联属性有：

mapping_type :关联类型

这个在HAS_ONE 关联里面必须使用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_key ： 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。

BELONGS_TO

Belongs_to 关联表示当前模型从属于另外一个父对象，例如每个用户都属于一个部门。我们可以做如下关联定义。

```
'Dept' => self::BELONGS_TO
```

完整方式定义为：

```
'Dept' => array(
    'mapping_type'  => self::BELONGS_TO,
    'class_name'    => 'Dept',
    'foreign_key'   => 'userId',
    'mapping_name'  => 'dept',
    // 定义更多的关联属性
    ......
),
```

关联BELONGS_TO定义支持的关联属性有：

属性	描述
class_name	要关联的模型类名
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。
foreign_key	关联的外键名称
mapping_fields	关联要查询的字段
condition	关联条件
parent_key	自引用关联的关联字段 默认为parent_id 自引用关联是一种比较特殊的关联，也就是关联表就是当前表。
as_fields	直接把关联的字段值映射成数据对象中的某个字段

HAS_MANY

HAS_MANY 关联表示当前模型拥有多个子对象，例如每个用户有多篇文章，我们可以这样来定义：

```
'Article' => self::HAS_MANY
```

完整定义方式为：

```
'Article' => array(
    'mapping_type'  => self::HAS_MANY,
    'class_name'    => 'Article',
    'foreign_key'   => 'userId',
    'mapping_name'  => 'articles',
    'mapping_order' => 'create_time desc',
    // 定义更多的关联属性
    ......
),
```

关联HAS_MANY定义支持的关联属性有：

属性	描述
class_name	要关联的模型类名
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。
foreign_key	关联的外键名称
parent_key	自引用关联的关联字段 默认为parent_id
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。
mapping_limit	关联要返回的记录数目
mapping_order	关联查询的排序

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定义 foreign_key 。

MANY_TO_MANY

MANY_TO_MANY 关联表示当前模型可以属于多个对象，而父对象则可能包含有多个子对象，通常两者之间需要一个中间表类约束和关联。例如每个用户可以属于多个组，每个组可以有多个用户：

```
'Group' => self::MANY_TO_MANY
```

完整定义方式为：

```
'Group' => array(
    'mapping_type'      =>  self::MANY_TO_MANY,
    'class_name'        =>  'Group',
    'mapping_name'      =>  'groups',
    'foreign_key'       =>  'userId',
    'relation_foreign_key'  =>  'groupId',
    'relation_table'    =>  'think_group_user' //此处应显式定义中间表名称，且不能使用C函数读取表前缀
    )
```

MANY_TO_MANY支持的关联属性定义有：

属性	描述
class_name	要关联的模型类名

属性	描述
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。
foreign_key	关联的外键名称 外键的默认规则是当前数据对象名称_id
relation_foreign_key	关联表的外键名称 默认的关联表的外键名称是表名_id
mapping_limit	关联要返回的记录数目
mapping_order	关联查询的排序
relation_table	多对多的中间关联表名称

多对多的中间表默认表规则是：`数据表前缀_关联操作的主表名_关联表名`

如果think_user 和 think_group 存在一个对应的中间表，默认的表名应该是 如果是由group来操作关联表，中间表应该是 think_group_user，如果是从user表来操作，那么应该是think_user_group，也就是说，多对多关联的设置，必须有一个Model类里面需要显式定义中间表，否则双向操作会出错。 中间表无需另外的id主键（但是这并不影响中间表的操作），通常只是由 user_id 和 group_id 构成。 默认会通过当前模型的getRelationTableName方法来自动获取，如果当前模型是User，关联模型是Group，那么关联表的名称也就是使用 user_group这样的格式，如果不是默认规则，需要指定relation_table属性。

3.2.2版本开始，relation_table定义支持简化写法，例如：

```
'relation_table'=>'__USER_GROUP__'
```

关联查询

由于性能问题，新版取消了自动关联查询机制，而统一使用relation方法进行关联操作，relation方法不但可以启用关联还可以控制局部关联操作，实现了关联操作一切尽在掌握之中。

```
$User = D("User");
$user = $User->relation(true)->find(1);
```

输出$user结果可能是类似于下面的数据：

```
array(
   'id'       => 1,
   'account'   => 'ThinkPHP',
   'password'  => '123456',
   'Profile'   => array(
      'email'     => 'liu21st@gmail.com',
      'nickname'  => '流年',
      ),
   )
```

我们可以看到，用户的关联数据已经被映射到数据对象的属性里面了。其中Profile就是关联定义的mapping_name属性。

如果我们按照下面的方式定义了as_fields属性的话，

```
protected $_link = array(
    'Profile'=>array(
        'mapping_type'  => self::HAS_ONE,
        'class_name'    => 'Profile',
        'foreign_key'   => 'userId',
        'as_fields' => 'email,nickname',
        ),
    );
```

查询的结果就变成了下面的结果

```
array(
    'id'        => 1,
    'account'   => 'ThinkPHP',
    'password'  => 'name',
    'email'     => 'liu21st@gmail.com',
    'nickname'  => '流年',
    )
```

email和nickname两个字段已经作为user数据对象的字段来显示了。

如果关联数据的字段名和当前数据对象的字段有冲突的话，怎么解决呢？

我们可以用下面的方式来变化下定义：

```
'as_fields' => 'email,nickname:username',
```

表示关联表的nickname字段映射成当前数据对象的username字段。

默认会把所有定义的关联数据都查询出来，有时候我们并不希望这样，就可以给relation方法传入参数来控制要关联查询的。

```
$User = D("User");
$user = $User->relation('Profile')->find(1);
```

关联查询一样可以支持select方法，如果要查询多个数据，并同时获取相应的关联数据，可以改成：

```
$User = D("User");
$list = $User->relation(true)->Select();
```

如果希望在完成的查询基础之上 再进行关联数据的查询，可以使用

```
$User = D("User");
$user = $User->find(1);
// 表示对当前查询的数据对象进行关联数据获取
$profile = $User->relationGet("Profile");
```

事实上，除了当前的参考模型User外，其他的关联模型是不需要创建的。

关联操作

除了关联查询外，系统也支持关联数据的自动写入、更新和删除

关联写入

```
$User = D("User");
$data = array();
$data["account"]    = "ThinkPHP";
$data["password"]   = "123456";
$data["Profile"]    = array(
  'email'    =>'liu21st@gmail.com',
  'nickname'    =>'流年',
);
$result = $User->relation(true)->add($data);
```

这样就会自动写入关联的Profile数据。

同样，可以使用参数来控制要关联写入的数据：

```
$result = $User->relation("Profile")->add($data);
```

> 当MANY_TO_MANY时，不建议使用关联插入。

关联更新

数据的关联更新和关联写入类似

```
$User = D("User");
$data["account"]    = "ThinkPHP";
$data["password"]   = "123456";
$data["Profile"]    = array(
    'email'    =>'liu21st@gmail.com',
    'nickname'    =>'流年',
);
$result = $User-> relation(true)->where(array('id'=>3))->save($data);
```

Relation(true)会关联保存User模型定义的所有关联数据，如果只需要关联保存部分数据，可以使用：

```
$result = $User->relation("Profile")->save($data);
```

这样就只会同时更新关联的Profile数据。

关联保存的规则：

HAS_ONE： 关联数据的更新直接赋值

HAS_MANY： 的关联数据如果传入主键的值 则表示更新 否则就表示新增

MANY_TO_MANY： 的数据更新是删除之前的数据后重新写入

关联删除

```
//删除用户ID为3的记录的同时删除关联数据
$result = $User->relation(true)->delete("3");
// 如果只需要关联删除部分数据，可以使用
$result = $User->relation("Profile")->delete("3");
```

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。如果需要使用高级模型的下面这些功能，记得需要继承Think\Model\AdvModel类或者采用动态模型。

```
namespace Home\Model;
use Think\Model\AdvModel;
class UserModel extends AdvModel{
 }
```

我们下面的示例都假设UserModel类继承自Think\Model\AdvModel类。

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Create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，包括写入过滤和读取过滤。

字段过滤的设置方式只需要在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。

序列化字段

序列化字段是新版推出的新功能，可以用简单的数据表字段完成复杂的表单数据存储，尤其是动态的表单数据字段。 要使用序列化字段的功能，只需要在模型中定义serializeField属性，定义格式如下：

```
protected $serializeField = 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"); // 实例化User对象
 // 然后直接给数据对象赋值
$User->name = 'ThinkPHP';
$User->email = 'ThinkPHP@gmail.com';
$User->address = '上海徐汇区';
 // 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;
```

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。

只读字段

只读字段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，我们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。

下面举个例子说明下：

```
$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';
$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。

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 ）” 。

悲观锁（ Pessimistic Locking ）

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
```

乐观锁（ Optimistic Locking ）

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。

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。

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
    }
 }
```

Mongo模型

Mongo模型是专门为Mongo数据库驱动而支持的Model扩展，如果需要操作Mongo数据库的话，自定义的模型类必须继承Think\Model\MongoModel。

Mongo模型为操作Mongo数据库提供了更方便的实用功能和查询用法，包括：

1. 对MongoId对象和非对象主键的全面支持；
2. 保持了动态追加字段的特性；

3. 数字自增字段的支持；
4. 执行SQL日志的支持；
5. 字段自动检测的支持；
6. 查询语言的支持；
7. MongoCode执行的支持；

主键

系统很好的支持Mongo的主键类型，Mongo默认的主键名是 _id，也可以通过设置pk属性改变主键名称（也许你需要用其他字段作为数据表的主键），例如：

```
namespace Home\Model;
use Think\Model\MongoModel;
Class UserModel extends MongoModel {
    Protected $pk = 'id';
}
```

主键支持三种类型（通过_idType属性设置），分别是：

类型	描述
self::TYPE_OBJECT或者1	（默认类型）采用MongoId对象，写入或者查询的时候传入数字或者字符会自动转换，获取的时候会自动转换成字符串。
self::TYPE_INT或者2	整形，支持自动增长，通过设置_autoInc 属性
self::TYPE_STRING或者3	字符串hash

设置主键类型示例：

```
namespace Home\Model;
use Think\Model\MongoModel;
Class UserModel extends MongoModel {
    Protected $_idType = self::TYPE_INT;
    protected $_autoinc =  true;
}
```

字段检测

MongoModel默认关闭字段检测，是为了保持Mongo的动态追加字段的特性，如果你的应用不需要使用Mongo动态追加字段的特性，可以设置 autoCheckFields 为true即可开启字段检测功能，提高安全性。一旦开启字段检测功能后，系统会自动查找当前数据表的第一条记录来获取字段列表。

如果你关闭字段检测功能的话，将不能使用查询的字段排除功能。

连贯操作

MongoModel中有部分连贯操作暂时不支持，包括：group、union、join、having、lock和distinct操作。其他连贯操作都可以很好的支持，例如：

```
$Model = new Think\Model\MongoModel("User");
$Model->field("name,email,age")->order("status desc")->limit("10,8")->select();
```

查询支持

Mongo数据库的查询条件和其他数据库有所区别。

1. 首先，支持所有的普通查询和快捷查询；
2. 表达式查询增加了一些针对MongoDb的查询用法；
3. 统计查询目前只能支持count操作，其他的可能要自己通过MongoCode来实现了；

MongoModel的组合查询支持

```
_string 采用MongoCode查询
_query 和其他数据库的请求字符串查询相同
_complex MongoDb暂不支持
```

MongoModel提供了MongoCode方法，可以支持MongoCode方式的查询或者操作。

表达式查询

表达式查询采用下面的方式：

```
$map['字段名'] = array('表达式','查询条件');
```

因为MongoDb的特性，MongoModel的表达式查询和其他的数据库有所区别，增加了一些新的用法。

表达式不分大小写，支持的查询表达式和Mongo原生的查询语法对照如下：

查询表达式	含义	Mongo原生查询条件
neq 或者ne	不等于	$ne
lt	小于	$lt
lte 或者elt	小于等于	$lte
gt	大于	$gt
gte 或者egt	大于等于	$gte

查询表达式	含义	Mongo原生查询条件
like	模糊查询 用MongoRegex正则模拟	无
mod	取模运算	$mod
in	in查询	$in
nin或者not in not	in查询	$nin
all	满足所有条件	$all
between	在某个的区间	无
not between	不在某个区间	无
exists	字段是否存在	$exists
size	限制属性大小	$size
type	限制字段类型	$type
regex	MongoRegex正则查询	MongoRegex实现
exp	使用MongoCode查询	无

注意，在使用like查询表达式的时候，和mysql的方式略有区别，对应关系如下：

Mysql模糊查询	Mongo模糊查询
array('like','%thinkphp%');	array('like','thinkphp');
array('like','thinkphp%');	array('like','^thinkphp');
array('like','%thinkphp');	array('like','thinkphp$');

LIKE： 同sql的LIKE 例如：

```
$map['name'] = array('like','^thinkphp');
```

查询条件就变成 `name like 'thinkphp%'`

设置支持 Mongo的数据更新设置用于数据保存和写入操作，可以支持：

表达式	含义	Mongo原生用法
inc	数字字段增长或减少	$inc
set	字段赋值	$set
unset	删除字段值	$unset
push	追加一个值到字段（必须是数组类型）里面去	$push
pushall	追加多个值到字段（必须是数组类型）里面去	$pushall

表达式	含义	Mongo原生用法
addtoset	增加一个值到字段（必须是数组类型）内，而且只有当这个值不在数组内才增加	$addtoset
pop	根据索引删除字段（必须是数组字段）中的一个值	$pop
pull	根据值删除字段（必须是数组字段）中的一个值	$pull
pullall	一次删除字段（必须是数组字段）中的多个值	$pullall

例如，

```
$data['id'] = 5;
$data['score'] = array('inc',2);
$Model->save($data);
```

其他

MongoModel增加了几个方法

mongoCode 执行MongoCode

getMongoNextId ([字段名]) 获取自增字段的下一个ID，可用于数字主键或者其他需要自增的字段，参数为空的时候表示或者主键的。

Clear 清空当前数据表方法

视图

模板定义

每个模块的模板文件是独立的，为了对模板文件更加有效的管理，ThinkPHP对模板文件进行目录划分，默认的模板文件定义规则是：

```
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：`./Application/Home/View/User/add.html`

如果你的默认视图层不是View，例如：

```
'DEFAULT_V_LAYER'      =>  'Template', // 设置默认的视图层名称
```

那么，对应的模板文件就变成了：`./Application/Home/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'
```

定义后，User控制器的add操作 对应的模板文件就变成是：`./Application/Home/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：`./Application/Home/View/User_add.html`

支持把模板目录设置到模块目录之外，有两种方式：

一、改变所有模块的模板文件目录

可以通过设置TMPL_PATH常量来改变所有模块的模板目录所在，例如：

```
define('TMPL_PATH','./Template/');
```

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

二、改变某个模块的模板文件目录 我们可以在模块配置文件中设置VIEW_PATH参数单独定义某个模块的视图目录，例如：

```
'VIEW_PATH'=>'./Theme/'
```

把当前模块的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/View/User/add.html 变成了 ./Theme/User/add.html 。

> 如果同时定义了TMPL_PATH常量和VIEW_PATH设置参数，那么以当前模块的VIEW_PATH参数设置优先。

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 // 没有启用模板主题之前
View/default/User/add.html // 启用模板主题之后
```

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。

```
// 在控制器中动态改变模板主题
$this->theme('blue')->display('add');
```

模板赋值

如果要在模板中输出变量，必须在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统一使用assign赋值。

```
$this->assign('name',$value);
// 下面的写法是等效的
$this->name = $value;
```

assign方法必须在display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，就可以在模板文件中输出变量了，如果使用的是内置模板的话，就可以这样输出：{$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']    =    'thinkphp';
$array['email']   =    'liu21st@gmail.com';
$array['phone']   =    '12335678';
$this->assign($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。

> 关于更多的模板标签使用，我们会在后面模板标签中详细讲解。

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

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'])模板文件的写法支持下面几种：

用法	描述
不带任何参数	自动定位当前操作的模板文件

用法	描述
[模块@][控制器:][操作]	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）

下面是一个最典型的用法，不带任何参数：

```
// 不带任何参数 自动定位当前操作的模板文件
$this->display();
```

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`

如果有更改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。

> 通常默认的视图目录是View

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');
```

表示调用当前控制器下面的edit模板

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

表示调用Member控制器下面的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。

> 要注意模板文件位置是相对于项目的入口文件，而不是模板目录。

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。

用法：

```
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。

例如：

```
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。

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。

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。

模板

本章的内容主要讲述了如何使用内置的模板引擎来定义模板文件，以及使用加载文件、模板布局和模板继承等高级功能。

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服务的内置模板引擎。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：

- 支持XML标签库和普通标签的混合定义；
- 支持直接使用PHP代码书写；
- 支持文件包含；
- 支持多级标签嵌套；
- 支持布局模板功能；
- 一次编译多次运行，编译和运行效率非常高；
- 模板文件和布局模板更新，自动更新模板缓存；
- 系统变量无需赋值直接输出；
- 支持多维数组的快速输出；
- 支持模板变量的默认值；
- 支持页面代码去除Html空白；
- 支持变量组合调节器和格式化功能；
- 允许定义模板禁用函数和禁用PHP语法；
- 通过标签库方式扩展。

每个模板文件在执行过程中都会生成一个编译后的缓存文件，其实就是一个可以运行的PHP文件。模板缓存默认位于项目的Runtime/模块/Cache目录下面，以模板文件的md5编码作为缓存文件名保存的。如果在模板标签的使用过程中发现问题，可以尝试通过查看模板缓存文件找到问题所在。

内置的模板引擎支持普通标签和XML标签方式两种标签定义，分别用于不同的目的：

标签类型	描述
普通标签	主要用于输出变量和做一些基本的操作
XML标签	主要完成一些逻辑判断、控制和循环输出，并且可扩展

这种方式的结合保证了模板引擎的简洁和强大的有效融合。

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}>！
```

后面的内容我们都以默认的标签定义来说明。

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';
$this->assign('data',$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}
```

系统变量

系统变量输出

普通的模板变量需要首先赋值后才能在模板中输出，但是系统变量则不需要，可以直接在模板中输出，系统变量的输出通常以{$Think 打头，例如：

```
{$Think.server.script_name} // 输出$_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 `$_SERVER`、`$_ENV`、`$_POST`、`$_GET`、`$_REQUEST`、`$_SESSION`和`$_COOKIE`变量。

常量输出

还可以输出常量

```
{$Think.const.MODULE_NAME}
```

或者直接使用

```
{$Think.MODULE_NAME}
```

配置输出

输出配置参数使用：

```
{$Think.config.db_charset}
{$Think.config.url_model}
```

语言变量

输出语言变量可以使用：

```
{$Think.lang.page_error}
{$Think.lang.var_error}
```

使用函数

我们往往需要对模板输出变量使用函数，可以使用：

```
{$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}
```

但完全没用这个必要。

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。

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"}
```

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	使用示例
+	{$a+$b}
-	{$a-$b}
*	{$a*$b}
/	{$a/$b}
%	{$a%$b}
++	{$a++} 或 {++$a}
--	{$a--} 或 {--$a}
综合运算	{$a+$b*10+$c}

在使用运算符的时候，不再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
```

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。

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。

内置标签

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>
```

更多的Cx标签库中的标签用法，参考内置标签。

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>
```

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>
```

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。

修改定界符

模板文件可以包含普通模板标签和XML模板标签，标签的定界符都可以重新配置。

普通标签

内置模板引擎的普通模板标签默认以{ 和 } 作为开始和结束标识，并且在开始标记紧跟标签的定义，如果之间有空格或者换行则被视为非模板标签直接输出。 例如：`{$name}` 、`{$vo.name}` 、`{$vo['name']|strtoupper}` 都属于普通模板标签。

要更改普遍模板的起始标签和结束标签，请使用下面的配置参数：

```
TMPL_L_DELIM  //模板引擎普通标签开始标记
TMPL_R_DELIM  //模板引擎普通标签结束标记
```

例如在项目配置文件中增加下面的配置：

```
'TMPL_L_DELIM'    =>    '<{',
'TMPL_R_DELIM'    =>    '}>'
```

普通标签的定界符就被修改了，原来的 `{$name}` 和 `{$vo.name}` 必须使用 `<{$name}>` 和 `<{$vo.name}>` 才能生效了。

> 如果你定制了普通标签的定界符，记得修改下默认的系统模板。

XML标签

普通模板标签主要用于模板变量输出和模板注释。如果要使用其它功能，请使用XML模板标签。XML模板标签可以用于模板变量输出、文件包含、条件控制、循环输出等功能，而且完全可以自己扩展功能。如果你觉得XML标签无法在正在使用的编辑器里面无法编辑，还可以更改XML标签库的起始和结束标签，请修改下面的配置参数：

```
TAGLIB_BEGIN    //标签库标签开始标签
TAGLIB_END    //标签库标签结束标记
```

例如在项目配置文件中增加下面的配置：

```
'TAGLIB_BEGIN'=>'[',
'TAGLIB_END'=>']',
```

原来的

```
<eq name="name" value="value">
相等
<else/>
不相等
</eq>
```

就必须改成

```
[eq name="name" value="value"]
相等
[else/]
不相等
[/eq]
```

> 注意：XML标签和普通标签的定界符不能冲突，否则会导致解析错误。

三元运算

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。

包含文件

在当前模版文件中包含其他的模版文件使用include标签，标签用法：

```
<include file='模版表达式或者模版文件1,模版表达式或者模版文件2,...' />
```

使用模版表达式

模版表达式的定义规则为：模块@主题/控制器/操作

例如：

```
<include file="Public/header" /> // 包含头部模版header
<include file="Public/menu" /> // 包含菜单模版menu
<include file="Blue/Public/menu" /> // 包含blue主题下面的menu模版
```

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。

使用模版文件

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" />
```

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。

内置标签

变量输出使用普通标签就足够了，但是要完成其他的控制、循环和判断功能，就需要借助模板引擎的标签库功能了，系统内置标签库的所有标签无需引入标签库即可直接使用。

内置标签包括：

标签名	作用	包含属性
include	包含外部模板文件（闭合）	file
import	导入资源文件（闭合 包括js css load别名）	file,href,type,value,basepath
volist	循环数组数据输出	name,id,offset,length,key,mod
foreach	数组或对象遍历输出	name,item,key
for	For循环数据输出	name,from,to,before,step
switch	分支判断输出	name
case	分支判断输出（必须和switch配套使用）	value,break
default	默认情况输出（闭合 必须和switch配套使用）	无
compare	比较输出（包括eq neq lt gt egt elt heq nheq等别名）	name,value,type
range	范围判断输出（包括in notin between notbetween别名）	name,value,type
present	判断是否赋值	name
notpresent	判断是否尚未赋值	name
empty	判断数据是否为空	name
notempty	判断数据是否不为空	name
defined	判断常量是否定义	name
notdefined	判断常量是否未定义	name
define	常量定义（闭合）	name,value
assign	变量赋值（闭合）	name,value
if	条件判断输出	condition
elseif	条件判断输出（闭合 必须和if标签配套使用）	condition
else	条件不成立输出（闭合 可用于其他标签）	无
php	使用php代码	无

Volist标签

volist标签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。 在控制器中首先对模版赋值：

```
$User = M('User');
$list = $User->limit(10)->select();
$this->assign('list',$list);
```

在模版定义如下，循环输出用户的编号和姓名：

```
<volist name="list" id="vo">
{$vo.id}:{$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" >
<eq name="mod" value="1">{$vo.name}</eq>
</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" >
{$vo.name}
<eq name="mod" value="4"><br/></eq>
</volist>
```

为空的时候输出提示：

```
<volist name="list" id="vo" empty="暂时没有数据" >
{$vo.id}|{$vo.name}
</volist>
```

empty属性不支持直接传入html语法，但可以支持变量输出，例如：

```
$this->assign('empty','<span class="empty">没有数据</span>');
$this->assign('list',$list);
```

然后在模板中使用：

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

输出循环变量

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

如果没有指定key属性的话，默认使用循环变量i，例如：

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

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>
```

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：

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

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}
</foreach>
```

也可以定义索引的变量名

```
<foreach name="list" item="vo" key="k" >
  {$k}|{$vo}
</foreach>
```

For标签

用法：

```
<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;
}
```

Switch标签

用法：

```
<switch name="变量" >
<case value="值1" break="0或1">输出内容1</case>
<case value="值2">输出内容2</case>
<default />默认情况
</switch>
```

使用方法如下：

```
<switch name="User.level">
   <case value="1">value1</case>
   <case value="2">value2</case>
   <default />default
</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>
   <default />default
</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。

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" value="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或者 equal	等于
neq 或者notequal	不等于
gt	大于
egt	大于等于
lt	小于
elt	小于等于
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。

例如，要求name变量的值等于value就输出，可以使用：

```
<eq name="name" value="value">value</eq>
```

或者

```
<equal name="name" value="value">value</equal>
```

也可以支持和else标签混合使用：

```
<eq name="name" value="value">
相等
<else/>
不相等
</eq>
```

当 name变量的值大于5就输出

```
<gt name="name" value="5">value</gt>
```

当name变量的值不小于5就输出

```
<egt name="name" value="5">value</egt>
```

比较标签中的变量可以支持对象的属性或者数组，甚至可以是系统变量，例如： 当vo对象的属性（或者数组，或者自动判断）等于5就输出

```
<eq name="vo.name" value="5">
{$vo.name}
</eq>
```

当vo对象的属性等于5就输出

```
<eq name="vo:name" value="5">
{$vo.name}
</eq>
```

当$vo['name']等于5就输出

```
<eq name="vo['name']" value="5">
{$vo.name}
</eq>
```

而且还可以支持对变量使用函数 当vo对象的属性值的字符串长度等于5就输出

```
<eq name="vo:name|strlen" value="5">{$vo.name}</eq>
```

变量名可以支持系统变量的方式，例如：

```
<eq name="Think.get.name" value="value">相等<else/>不相等</eq>
```

通常比较标签的值是一个字符串或者数字，如果需要使用变量，只需要在前面添加“$”标志： 当vo对象的属性等于$a就输出

```
<eq name="vo:name" value="$a">{$vo.name}</eq>
```

所有的比较标签可以统一使用compare标签（其实所有的比较标签都是compare标签的别名），例如：当name变量的值等于5就输出

```
<compare name="name" value="5" type="eq">value</compare>
```

等效于

```
<eq name="name" value="5" >value</eq>
```

其中type属性的值就是上面列出的比较标签名称

范围判断标签

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。

IN和NOTIN

用法： 假设我们中控制器中给id赋值为1：

```
$id =    1;
$this->assign('id',$id);
```

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="1,2,3">
id在范围内
</in>
```

最后会输出：`id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>
```

> 更多的系统变量用法可以参考系统变量部分。

value属性也可以使用变量，例如：

```
<in name="id" value="$range">
id在范围内
</in>
```

$range变量可以是数组，也可以是以逗号分隔的字符串。

value属性还可以使用系统变量，例如：

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

BETWEEN 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2
</notbetween>
```

也可以使用else标签把两个用法合并，例如：

```
<between name="id" value="1,10">
输出内容1
<else/>
输出内容2
</between>
```

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>
```

RANGE

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。

IF标签

用法示例：

```
<if condition="($name eq 1) OR ($name gt 100) "> value1
<elseif condition="$name eq 2"/>value2
<else /> value3
</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。

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>
```

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>
```

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>
```

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'];

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="$name" />
```

或者

```
<define name="MY_DEFINE_NAME" value="$Think.get.name" />
```

标签嵌套

模板引擎支持标签的多层嵌套功能，可以对标签库的标签指定可以嵌套。

系统内置的标签中，volist、switch、if、elseif、else、foreach、compare（包括所有的比较标签）、（not）present、（not）empty、（not）defined等标签都可以嵌套使用。例如：

```
<volist name="list" id="vo">
   <volist name="vo['sub']" id="sub">
      {$sub.name}
   </volist>
</volist>
```

上面的标签可以用于输出双重循环。

> 嵌套层次是由标签库中的标签定义的时候的level属性决定的。

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" />
```

使用PHP代码

Php代码可以和标签在模板文件中混合使用，可以在模板文件里面书写任意的PHP语句代码 ，包括下面两种方式：

使用php标签

例如：

```
<php>echo 'Hello,world!';</php>
```

我们建议需要使用PHP代码的时候尽量采用php标签，因为原生的PHP语法可能会被配置禁用而导致解析错误。

使用原生php代码

```
<?php echo 'Hello,world!'; ?>
```

注意：php标签或者php代码里面就不能再使用标签（包括普通标签和XML标签）了，因此下面的几种方式都是无效的：

```
<php><eq name='name'value='value'>value</eq></php>
```

Php标签里面使用了 `eq` 标签，因此无效

```
<php>if( {$user} != 'ThinkPHP' ) echo  'ThinkPHP' ;</php>
```

Php标签里面使用了 `{$user}` 普通标签输出变量 ，因此无效。

```
<php>if( $user.name != 'ThinkPHP' ) echo  'ThinkPHP' ;</php>
```

Php标签里面使用了 `$user.name` 点语法变量输出 ，因此无效。

> 简而言之，在PHP标签里面不能再使用PHP本身不支持的代码。

如果设置了 `TMPL_DENY_PHP` 参数为true，就不能在模板中使用原生的PHP代码，但是仍然支持PHP标签输出。

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。

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。

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。

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。

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。

有三种布局模板的支持方式：

第一种方式：全局配置方式

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__}'
```

> 一个布局模板同时只能有一个特定替换字符串。

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。

第二种方式：模板标签方式

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。

第三种方式：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。

模板替换

在进行模板渲染之前，系统还会对读取的模板内容进行一些特殊字符串替换操作，也就是实现了模板输出的替换和过滤。该替换操作仅针对内置的模版引擎。

这个机制可以使得模板文件的定义更加方便，默认的替换规则有：

```
__ROOT__： 会替换成当前网站的地址（不含域名）
__APP__： 会替换成当前应用的URL地址 （不含域名）
__MODULE__：会替换成当前模块的URL地址 （不含域名）
__CONTROLLER__（__或者__URL__ 兼容考虑）： 会替换成当前控制器的URL地址（不含域名）
__ACTION__：会替换成当前操作的URL地址 （不含域名）
__SELF__： 会替换成当前的页面URL
__PUBLIC__：会被替换成当前网站的公共目录 通常是 /Public/
```

> 默认情况下，模板替换只会替换模板文件的特殊字符串，不会替换动态数据中的输出的内容。

注意这些特殊的字符串是严格区别大小写的，并且这些特殊字符串的替换规则是可以更改或者增加的，我们只需要在应用或者模块的配置文件中配置TMPL_PARSE_STRING就可以完成。如果有相同的数组索引，就会更改系统的默认规则。例如：

```
'TMPL_PARSE_STRING'  =>array(
    '__PUBLIC__' => '/Common', // 更改默认的/Public 替换规则
    '__JS__'     => '/Public/JS/', // 增加新的JS类库路径替换规则
    '__UPLOAD__' => '/Uploads', // 增加新的上传路径替换规则
)
```

有了模板替换规则后，模板中的所有 __PUBLIC__ 字符串都会被替换，那如果确实需要输出 __PUBLIC__ 字符串到模板呢，我们可以通过增加替换规则的方式，例如：

```
'TMPL_PARSE_STRING'  =>array(
    '--PUBLIC--' => '__PUBLIC__', // 采用新规则输出`__PUBLIC__`字符串
)
```

这样增加替换规则后，如果我们要在模板中使用 __PUBLIC__ 字符串，只需要在模板中添加--PUBLIC--，其他替换字符串的输出方式类似。

调试

调试模式

ThinkPHP有专门为开发过程而设置的调试模式，开启调试模式后，会牺牲一定的执行效率，但带来的方便和除错功能非常值得。

> 我们强烈建议ThinkPHP开发人员在开发阶段始终开启调试模式（直到正式部署后关闭调试模式），方便及时发现隐患问题和分析、解决问题。

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';
```

调试模式的优势在于：

- 开启日志记录，任何错误信息和调试信息都会详细记录，便于调试；
- 关闭模板缓存，模板修改可以即时生效；
- 记录SQL日志，方便分析SQL；
- 关闭字段缓存，数据表字段修改不受缓存影响；
- 严格检查文件大小写（即使是Windows平台），帮助你提前发现Linux部署可能导致的隐患问题；
- 通过页面Trace功能更好的调试和发现错误；

在开启调试模式的状态下，系统会首先导入框架默认的调试模式配置文件，该文件位于系统目录的

`Conf\debug.php`。

通常情况下，调试配置文件里面可以进行一些开发模式所需要的配置。例如，配置额外的数据库连接用于调试，开启日志写入便于查找错误信息、开启页面Trace输出更多的调试信息等等。

如果检测到应用的配置目录中有存在debug.php文件，则会自动加载该配置文件，并且和系统项目配置文件以及系统调试配置文件合并，也就是说，debug.php配置文件只需要配置和项目配置文件以及系统调试配置文件不同的参数或者新增的参数。

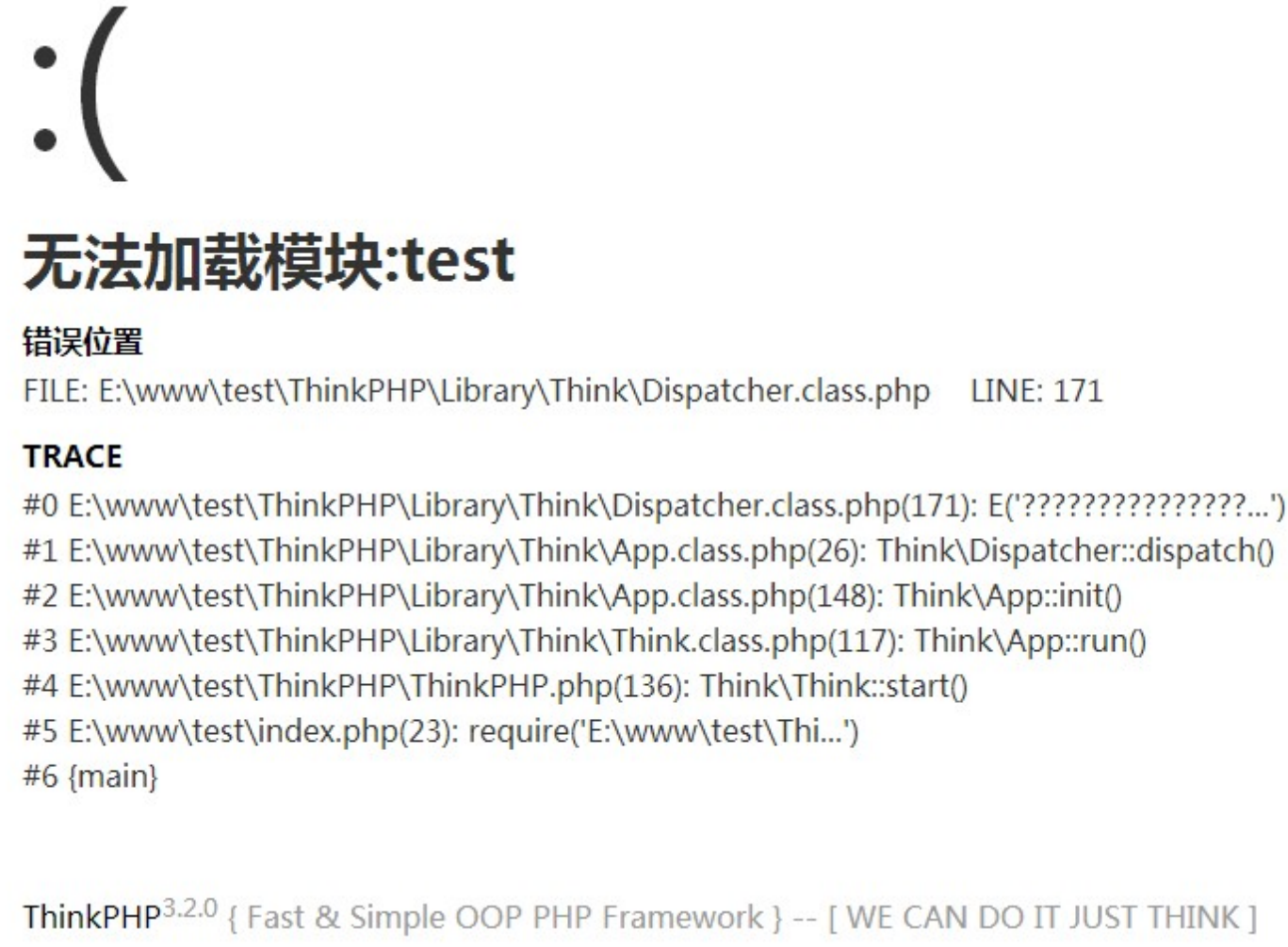
> 由于调试模式没有任何缓存，因此涉及到较多的文件IO操作和模板实时编译，所以在开启调试模式的情况下，性能会有一定的下降，但不会影响部署模式的性能。另外需要注意的是，一旦关闭调试模式，项目的调试配置文件即刻失效。

一旦关闭调试模式，发生错误后不会提示具体的错误信息，如果你仍然希望看到具体的错误信息，那么可以如下设置：

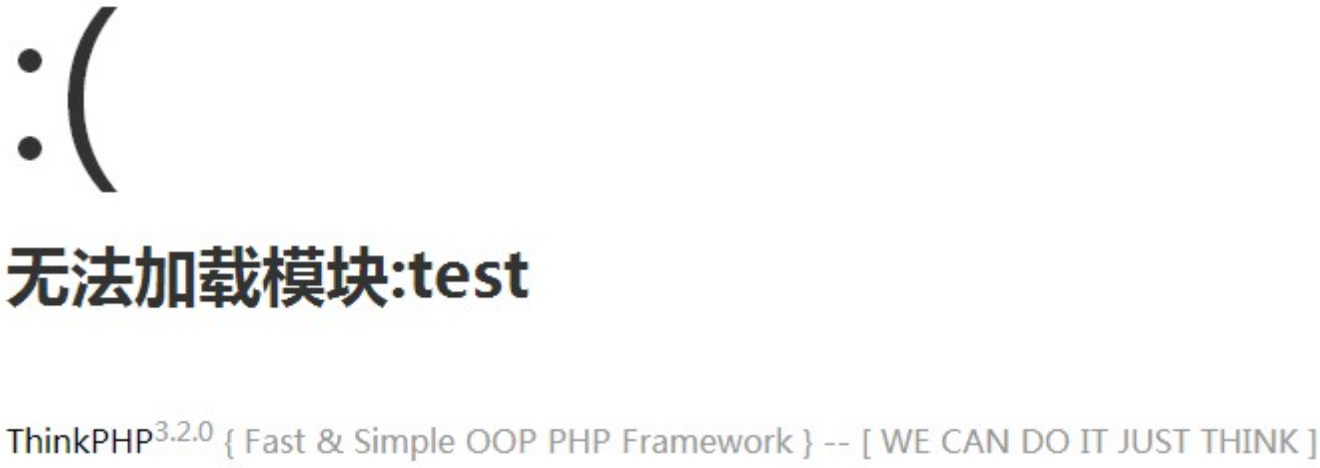
```
'SHOW_ERROR_MSG'        =>  true,    // 显示错误信息
```

异常处理

和PHP默认的异常处理不同，ThinkPHP抛出的不是单纯的错误信息，而是一个人性化的错误页面，如下图所示：



只有在调试模式下面才能显示具体的错误信息，如果在部署模式下面，你可能看到的是一个简单的提示文字，例如：



一旦关闭调试模式，发生错误后不会提示具体的错误信息，如果你仍然希望看到具体的错误信息，那么可以如下设置：

```
'SHOW_ERROR_MSG'        =>  true,    // 显示错误信息
```

如果你试图在部署模式下访问一个不存在的模块或者操作，会发送404错误。

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\Exception('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。

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

记录方式

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 。

手动记录

一般情况下，系统的日志记录是自动的，无需手动记录，但是某些时候也需要手动记录日志信息，Log类提供了3个方法用于记录日志。

方法	描述
Log::record()	记录日志信息到内存
Log::save()	把保存在内存中的日志信息（用指定的记录方式）写入
Log::write()	实时写入一条日志信息

由于系统在请求结束后会自动调用Log::save方法，所以通常，你只需要调用Log::record记录日志信息即可。

record方法用法如下：

```
Think\Log::record('测试日志信息');
```

默认的话记录的日志级别是ERR，也可以指定日志级别：

```
Think\Log::record('测试日志信息，这是警告级别','WARN');
```

record方法只会记录当前配置允许记录的日志级别的信息，如果应用配置为：

```
'LOG_LEVEL'  =>'EMERG,ALERT,CRIT,ERR', // 只记录EMERG ALERT CRIT ERR 错误
```

那么上面的record方法记录的日志信息会被直接过滤，或者你可以强制记录：

```
Think\Log::record('测试日志信息，这是警告级别','WARN',true);
```

采用record方法记录的日志信息不是实时保存的，如果需要实时记录的话，可以采用write方法，例如：

```
Think\Log::write('测试日志信息，这是警告级别，并且实时写入','WARN');
```

write方法写入日志的时候 不受配置的允许日志级别影响，可以实时写入任意级别的日志信息。

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）。

> 在部署模式下面，显示的调试信息没有调试模式完整，通常我们建议页面Trace配合调试模式一起使用。

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,
```

该参数默认为关闭，开启后并且你的页面有模板输出的话，页面右下角会显示ThinkPHP的LOGO：

我们看到的LOGO后面的数字就是当前页面的执行时间（单位是秒） 点击该图标后，会展开详细的页面Trace信息，如图：

页面Trace框架有6个选项卡，分别是基本、文件、流程、错误、SQL和调试，点击不同的选项卡会切换到不同的Trace信息窗口。

选项卡	描述
基本	当前页面的基本摘要信息，例如执行时间、内存开销、文件加载数、查询次数等等。
文件	详细列出当前页面执行过程中加载的文件及其大小。
流程	会列出当前页面执行到的行为和相关流程（待完善）。
错误	当前页面执行过程中的一些错误信息，包括警告错误。
SQL	当前页面执行到的SQL语句信息。
调试	开发人员在程序中进行的调试输出。

页面Trace的选项卡是可以定制和扩展的，默认的配置为：

```
'TRACE_PAGE_TABS'=>array(
   'base'=>'基本',
    'file'=>'文件',
    'think'=>'流程',
    'error'=>'错误',
    'sql'=>'SQL',
    'debug'=>'调试'
)
```

也就是我们看到的默认情况下显示的选项卡，如果你希望增加新的选项卡：用户，则可以修改配置如下：

```
'TRACE_PAGE_TABS'=>array(
    'base'=>'基本',
    'file'=>'文件',
    'think'=>'流程',
    'error'=>'错误',
    'sql'=>'SQL',
    'debug'=>'调试',
    'user'=>'用户'
)
```

也可以把某几个选项卡合并，例如：

```
'TRACE_PAGE_TABS'=>array(
    'base'=>'基本',
    'file'=>'文件',
    'think'=>'流程',
    'error|debug|sql'=>'调试',
    'user'=>'用户'
)
```

我们把刚才的用户信息调试输出到用户选项卡，trace方法的用法如下：

```
trace($user,'用户信息','user');
```

第三个参数表示选项卡的标识，和我们在 TRACE_PAGE_TABS 中配置的对应。 默认情况下，页面Trace窗口显示的信息是不会保存的，如果希望保存这些trace信息，我们可以配置 PAGE_TRACE_SAVE 参数

```
'PAGE_TRACE_SAVE'=>true
```

开启页面trace信息保存后，每次的页面Trace信息会以日志形式保存到项目的日志目录中，命名格式是：当前日期_trace.log，例如： 12-06-21_trace.log

如果不希望保存所有的选项卡的信息，可以设置需要保存的选项卡，例如：

```
'PAGE_TRACE_SAVE' => array('base','file','sql');
```

设置后只会保存base、file和sql三个选项卡的信息。

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
```

断点调试

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');
```

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)
```

相关参数的使用如下：

参数	描述
var（必须）	要输出的变量，支持所有变量类型
echo（可选）	是否直接输出，默认为true，如果为false则返回但不输出
label（可选）	变量输出的label标识，默认为空
strict（可选）	输出变量类型，默认为true，如果为false则采用print_r输出

如果echo参数为false 则返回要输出的字符串

使用示例：

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

在浏览器输出的结果是：

```
array(12) {
  ["id"]          => string(1) "3"
  ["name"]        => string(0) ""
  ["user_id"]     => string(1) "0"
  ["cate_id"]     => string(1) "0"
  ["title"]       => string(4) "test"
  ["content"]     => string(4) "test"
  ["create_time"] => string(1) "0"
  ["update_time"] => string(1) "0"
  ["status"]      => string(1) "0"
  ["read_count"]  => string(1) "0"
  ["comment_count"] => string(1) "0"
  ["tags"]        => string(0) ""
}
```

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。

错误调试

如果需要我们可以使用E方法输出错误信息并中断执行，例如：

```
//输出错误信息，并中止执行
E($msg);
```

模型调试

调试执行的SQL语句

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();
// 3.2版本中可以使用简化的方法
echo $User->_sql();
```

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。

> 注意：Mongo数据库驱动由于接口的特殊性，不存在执行SQL的概念，因此SQL日志记录功能是额外封装实现的，所以出于性能考虑，只有在开启调试模式的时候才支持使用getLastSql方法获取最后执行的SQL记录。

调试数据库错误信息

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。

缓存

在项目中，合理的使用缓存对性能有较大的帮助。ThinkPHP提供了方便的缓存方式，包括数据缓存、静态缓存和查询缓存等，支持包括文件方式、APC、Db、Memcache、Shmop、Sqlite、Redis、Eaccelerator和Xcache在内的动态数据缓存类型，以及可定制的静态缓存规则，并提供了快捷方法进行存取操作。

数据缓存

在ThinkPHP中进行缓存操作，一般情况下并不需要直接操作缓存类，因为系统内置对缓存操作进行了封装，直接采用S方法即可，例如：

缓存初始化

```
// 缓存初始化
S(array('type'=>'xcache','expire'=>60));
```

缓存初始化可以支持的参数根据不同的缓存方式有所区别，常用的参数是：

参数	描述
expire	缓存有效期（时间为秒）
prefix	缓存标识前缀
type	缓存类型

系统目前已经支持的缓存类型包括：
Apachenote、Apc、Db、Eaccelerator、File、Memcache、Redis、Shmop、Sqlite、Wincache和Xca
。

如果S方法不传入type参数初始化的话，则读取配置文件中设置的 DATA_CACHE_TYPE 参数值作为默认类型。同样的道理，prefix参数如果没有传入会读取配置文件的 DATA_CACHE_PREFIX 参数值，expire参数没有传入则读取 DATA_CACHE_TIME 配置值作为默认。

有些缓存方式会有一些自身特殊的参数，例如Memcache缓存，还需要配置其他的参数：

```
S(array(
    'type'=>'memcache',
    'host'=>'192.168.1.10',
    'port'=>'11211',
    'prefix'=>'think',
    'expire'=>60)
);
```

对于全局的缓存方式，一般我们建议添加prefix（缓存前缀）参数用以区分不同的应用，以免混淆。

缓存设置

```
// 设置缓存
S('name',$value);
```

会按照缓存初始化时候的参数进行缓存数据，也可以在缓存设置的时候改变参数，例如：

```
// 缓存数据300秒
S('name',$value,300);
```

甚至改变之前的缓存方式或者更多的参数：

```
// 采用文件方式缓存数据300秒
S('name',$value,array('type'=>'file','expire'=>300));
```

如果你在缓存设置的时候采用上面的数组方式传入参数的话，会影响到后面的缓存存取。

缓存读取

```
// 读取缓存
$value = S('name');
```

缓存读取的是前面缓存设置的值，这个值会受缓存初始化或者缓存设置的时候传入的参数影响。 如果缓存标识不存在或者已经过期，则返回false，否则返回缓存值。

缓存删除

```
// 删除缓存
S('name',null);
```

删除缓存标识为name的缓存数据。

对象方式操作缓存

我们可以采用对象方式操作缓存，例如：

```
// 初始化缓存
$cache = S(array('type'=>'xcache','prefix'=>'think','expire'=>600));
$cache->name = 'value'; // 设置缓存
$value = $cache->name; // 获取缓存
unset($cache->name); // 删除缓存
```

> 如果你设置了缓存前缀的话，对应的缓存操作只是对应该缓存前缀标识的，不会影响其他的缓存。

关于文件缓存方式的安全机制

如果你使用的是文件方式的缓存机制，那么可以设置DATA_CACHE_KEY参数，避免缓存文件名被猜测到，例如：

```
'DATA_CACHE_KEY'=>'think'
```

缓存队列

数据缓存可以支持缓存队列，简单的说就是可以限制缓存的数量，只需要在初始化的时候指定 length 参数：

```
S(array('type'=>'xcache','length'=>100,'expire'=>60));
```

设置了 length 参数后，系统只会缓存最近的100条缓存数据。

快速缓存

如果你的存储数据没有有效期的需求，那么系统还提供了一个快速缓存方法F可以用来更快的操作。

> F方法可以支持不同的存储类型，如果是文件类型的话，默认保存在DATA_PATH目录下面。

快速缓存Data数据

```
F('data',$Data);
```

快速缓存Data数据，保存到指定的目录

```
F('data',$Data,TEMP_PATH);
```

获取缓存数据

```
$Data = F('data');
```

删除缓存数据

```
F('data',NULL);
```

F方法支持自动创建缓存子目录，在 DATA_PATH 目录下面缓存data数据，如果User子目录不存在，则自动创建：

```
F('User/data',$Data);
```

系统内置的数据字段信息缓存就是用了快速缓存机制。

查询缓存

对于及时性要求不高的数据查询，我们可以使用查询缓存功能来提高性能，而且无需自己使用缓存方法进行缓存和获取。

查询缓存功能支持所有的数据库，并且支持所有的缓存方式和有效期。

在使用查询缓存的时候，只需要调用Model类的cache方法，例如：

```
$Model->cache(true)->where('status=1')->select();
```

如果使用了 cache(true) ，则在查询的同时会根据当前的查询条件等信息生成一个带有唯一标识的查询缓存，如果指定了key的话，则直接生成名称为key的查询缓存 ，例如：

```
$Model->cache('cache_name')->select();
```

> 指定key的方式会让查询缓存更加高效。

默认情况下缓存方式采用DATA_CACHE_TYPE参数设置的缓存方式（系统默认值为File表示采用文件方式缓存），缓存有效期是DATA_CACHE_TIME参数设置的时间，也可以单独制定查询缓存的缓存方式和有效期：

```
$Model->cache(true,60,'xcache')->select();
```

表示当前查询缓存的缓存方式为xcache，并且缓存有效期为60秒。

同样的查询，如果没有使用cache方法，则不会获取或者生成任何缓存，即便是之前调用过Cache方法。

如果指定了查询缓存的key的话，则可以在外部通过S方法直接获取查询缓存的内容，例如：

```
$value = S('cache_name');
```

除了select方法之外，查询缓存还支持find和getField方法，以及他们的衍生方法（包括统计查询和动态查询方法）。

```
// 对查询数据缓存60秒
$Model->where($map)->cache('key',60)->find();
```

具体应用的时候可以根据需要选择缓存方式和缓存有效期。

静态缓存

要使用静态缓存功能，需要开启 HTML_CACHE_ON 参数，并且使用 HTML_CACHE_RULES 配置参数设置静态缓存规则文件 。

> 虽然也可以在应用配置文件中定义静态缓存规则，但是建议是在模块配置文件中为不同的模块定义静态缓存规则。

静态规则定义

静态规则的定义方式如下：

```
'HTML_CACHE_ON'     =>    true, // 开启静态缓存
'HTML_CACHE_TIME'   =>    60,   // 全局静态缓存有效期（秒）
'HTML_FILE_SUFFIX'  =>    '.shtml', // 设置静态缓存文件后缀
'HTML_CACHE_RULES'  =>    array(  // 定义静态缓存规则
    // 定义格式1 数组方式
    '静态地址'    =>    array('静态规则', '有效期', '附加规则'),
    // 定义格式2 字符串方式
    '静态地址'    =>    '静态规则',
)
```

定义格式1采用数组方式 便于单独为某个静态规则设置不同的有效期，定义格式2采用字符串方式订阅静态规则，同时采用 HTML_CACHE_TIME 设置的全局静态缓存有效期。

> 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。 并且静态缓存支持不同的存储类型。 静态缓存仅在GET请求下面有效。

静态地址

静态地址包括下面几种定义格式：

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。

第二种是定义全局的控制器静态规则，例如定义所有的User控制器的静态规则为：

```
'user:'=>array('User/{:action}_{id}','600')
```

其中，{:action} 表示当前的操作名称

第三种是定义某个控制器的操作的静态规则，例如，我们需要定义Blog控制器的read操作进行静态缓存

```
'blog:read'=>array('{id}',0)
```

第四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}'),
```

表示根据当前的URL进行缓存。

静态规则

静态规则是用于定义要生成的静态文件的名称，静态规则的定义要确保不会冲突，写法可以包括以下情况：

1、使用系统变量

包括 _GET 、 _REQUEST 、 _SERVER 、 _SESSION 、 _COOKIE 格式：

```
{$_×××|function}
```

例如：

```
{$_GET.name}
{$_SERVER.REQUEST_URI|md5}
```

2、使用框架特定的变量

{:module} 、 {:controller} 和 {:action} 分别表示当前模块名、控制器名和操作名。

例如：

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

3、使用_GET变量

{var|function} 也就是说 {id} 其实等效于 {$_GET.id}

4、直接使用函数

{|function} 例如：

```
{|time}
```

5、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。

静态缓存有效期

单位为秒。如果不定义，则会获取配置参数 HTML_CACHE_TIME 的设置值，如果定义为0则表示永久缓存。

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']);`

安全

在项目开发完成准备部署之前，应该检查下是否存在安全隐患，这一部分内容帮助你一起来加强项目的安全问题，指导你如何使用表单令牌、字段类型验证、输入过滤、上传安全、防止XSS攻击和目录安全保护等功能。

输入过滤

永远不要相信客户端提交的数据，所以对于输入数据的过滤势在必行，我们建议：

- 开启令牌验证避免数据的重复提交；
- 使用自动验证和自动完成机制进行初步过滤；
- 使用系统提供的I函数获取用户输入数据；
- 对不同的应用需求设置不同的安全过滤函数，常见的安全过滤函数包括stripslashes、htmlentities、htmlspecialchars和strip_tags等；

使用I函数过滤

使用系统内置的 I函数 是避免输入数据出现安全隐患的重要手段，I函数默认的过滤方法是 `htmlspecialchars` ，如果我们需要采用其他的方法进行安全过滤，有两种方式：

如果是全局的过滤方法，那么可以设置DEFAULT_FILTER，例如：

```
'DEFAULT_FILTER'     =>  'strip_tags',
```

设置了DEFAULT_FILTER后，所有的I函数调用默认都会使用 strip_tags 进行过滤。

当然，我们也可以设置多个过滤方法，例如：

```
'DEFAULT_FILTER'     =>  'strip_tags,stripslashes',
```

如果是仅需要对个别数据采用特殊的过滤方法，可以在调用I函数的时候传入过滤方法，例如：

```
I('post.id',0,'intval'); // 用intval过滤$_POST['id']
I('get.title','','strip_tags'); // 用strip_tags过滤$_GET['title']
```

> 要尽量避免直接使用$_GET $_POST $_REQUEST 等数据，这些可能会导致安全的隐患。 就算你要获取整个$_GET数据，我们也建议你使用 `I('get.')` 的方式

写入数据过滤

如果你没有使用I函数进行数据过滤的话，还可以在模型的写入操作之前调用filter方法对数据进行安全过滤，例如：

```
$this->data($data)->filter('strip_tags')->add();
```

表单合法性检测

在处理表单提交的数据的时候，建议尽量采用Think\Model类提供的create方法首先进行数据创建，然后再写入数据库。

> create方法在创建数据的同时，可以进行更为安全的处理操作，而且这一切让你的表单处理变得更简单。

使用create方法创建数据对象的时候，可以使用数据的合法性检测，支持两种方式：

配置insertFields 和 updateFields属性

可以分别为新增和编辑表单设置 insertFields 和 updateFields 属性，使用create方法创建数据对象的时候，不在定义范围内的属性将直接丢弃，避免表单提交非法数据。

insertFields 和 updateFields属性的设置采用字符串（逗号分割多个字段）或者数组的方式。

设置的字段应该是实际的数据表字段，而不受字段映射的影响。例如：

```
namespace Home\Model;
class UserModel extends \Think\Model{
    protected $insertFields = array('account','password','nickname','email');
    protected $updateFields = array('nickname','email');
}
```

定义后，调用add方法写入用户数据的时候，只能写入 'account','password','nickname','email' 这几个字段，编辑的时候只能更新 'nickname','email' 两个字段。

在使用的时候，我们调用create方法的时候，会根据提交类型自动识别insertFields和updateFields属性：

```
D('User')->create();
```

直接调用field方法

如果不想定义insertFields和updateFields属性，可以在调用create方法之前直接调用field方法，例如，实现和上面的例子同样的作用： 在新增用户数据的时候，使用：

```
M('User')->field('account,password,nickname,email')->create();
```

而在更新用户数据的时候，使用：

```
M('User')->field('nickname,email')->create();
```

这里的字段也是实际的数据表字段。

> field方法也可以使用数组方式。

使用字段合法性检测后，你不再需要担心用户在提交表单的时候注入非法字段数据了。

表单令牌

ThinkPHP支持表单令牌验证功能，可以有效防止表单的重复提交等安全防护。

要启用表单令牌功能，需要配置行为绑定，在应用或者模块的配置目录下面的行为定义文件tags.php中，添加：

```
return array(
    // 添加下面一行定义即可
    'view_filter' => array('Behavior\TokenBuild'),
   // 如果是3.2.1以上版本 需要改成
   // 'view_filter' => array('Behavior\TokenBuildBehavior'),
);
```

表示在`view_filter`标签位置执行表单令牌检测行为。

表单令牌验证相关的配置参数有：

```
'TOKEN_ON'      =>    true,  // 是否开启令牌验证 默认关闭
'TOKEN_NAME'    =>    '__hash__',    // 令牌验证的表单隐藏字段名称，默认为__hash__
'TOKEN_TYPE'    =>    'md5',  //令牌哈希验证规则 默认为MD5
'TOKEN_RESET'   =>    true,  //令牌验证出错后是否重置令牌 默认为true
```

如果开启表单令牌验证功能，系统会自动在带有表单的模板文件里面自动生成以TOKEN_NAME为名称的隐藏域，其值则是TOKEN_TYPE方式生成的哈希字符串，用于实现表单的自动令牌验证。

自动生成的隐藏域位于表单Form结束标志之前，如果希望自己控制隐藏域的位置，可以手动在表单页面添

加{__TOKEN__}标识，系统会在输出模板的时候自动替换。

> 如果页面中存在多个表单，建议添加标识，并确保只有一个表单需要令牌验证。

如果个别页面输出不希望进行表单令牌验证，可以在控制器中的输出方法之前动态关闭表单令牌验证，例如：

```
C('TOKEN_ON',false);
$this->display();
```

模型类在创建数据对象的同时会自动进行表单令牌验证操作，如果你没有使用create方法创建数据对象的话，则需要手动调用模型的 autoCheckToken 方法进行表单令牌验证。如果返回false，则表示表单令牌验证错误。例如：

```
$User = M("User"); // 实例化User对象
 // 手动进行令牌验证
 if (!$User->autoCheckToken($_POST)){
 // 令牌验证错误
 }
```

防止SQL注入

对于WEB应用来说，SQL注入攻击无疑是首要防范的安全问题，系统底层对于数据安全方面本身进行了很多的处理和相应的防范机制，例如：

```
$User = M("User"); // 实例化User对象
$User->find($_GET["id"]);
```

即便用户输入了一些恶意的id参数，系统也会强制转换成整型，避免恶意注入。这是因为，系统会对数据进行强制的数据类型检测，并且对数据来源进行数据格式转换。而且，对于字符串类型的数据，ThinkPHP都会进行escape_string处理(real_escape_string,mysql_escape_string)，还支持参数绑定。

通常的安全隐患在于你的查询条件使用了字符串参数，然后其中一些变量又依赖由客户端的用户输入。

要有效的防止SQL注入问题，我们建议：

> - 查询条件尽量使用数组方式，这是更为安全的方式；
> - 如果不得已必须使用字符串查询条件，使用预处理机制；
> - 使用自动验证和自动完成机制进行针对应用的自定义过滤；
> - 如果环境允许，尽量使用PDO方式，并使用参数绑定。

查询条件预处理

where方法使用字符串条件的时候，支持预处理（安全过滤），并支持两种方式传入预处理参数，例如：

```
$Model->where("id=%d and username='%s' and xx='%f'",array($id,$username,$xx))->select();
// 或者
$Model->where("id=%d and username='%s' and xx='%f'",$id,$username,$xx)->select();
```

模型的query和execute方法 同样支持预处理机制，例如：

```
$model->query('select * from user where id=%d and status=%d',$id,$status);
//或者
$model->query('select * from user where id=%d and status=%d',array($id,$status));
```

execute方法用法同query方法。

目录安全文件

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：

```
define('DIR_SECURE_FILENAME', 'default.html');
define('APP_PATH','./Application/');
require './ThinkPHP/ThinkPHP.php';
```

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', 'deny Access!');
```

> 注意：目录安全文件仅在第一次生成模块目录的时候生成。如果是3.2.1版本以上，则可以调用代码生成，例如：

```
// dirs变量是要生成安全文件的目录数组
\Think\Build::buildDirSecure($dirs);
```

保护模板文件

因为模板文件中可能会泄露数据表的字段信息，有两种方法可以保护你的模板文件不被访问到：

第一种方式是配置.htaccess文件，针对Apache服务器而言。

把以下代码保存在模块的模板目录（默认是View）下保存存为.htaccess。

```
<Files *.html>
Order Allow,Deny
Deny from all
 </Files>
```

如果你的模板文件后缀不是html可以将*.html改成你的模板文件的后缀。

第二种方式是针对独立的服务器，不适合虚拟主机用户。 把项目目录部署到网站WEB目录之外，这样，整个项目目录都不能直接访问，当然模板文件也保护起来了。

上传安全

网站的上传功能也是一个非常容易被攻击的入口，所以对上传功能的安全检查是尤其必要的。

系统提供的上传类 Think\Upload 提供了安全方面的支持，包括对文件后缀、文件类型、文件大小以及上传图片文件的合法性检查，确保你已经在上传操作中启用了这些合法性检查。

更多的关于 Think\Upload 类的用法参考这里。

防止XSS攻击

XSS（跨站脚本攻击）可以用于窃取其他用户的Cookie信息，要避免此类问题，可以采用如下解决方案：

- 直接过滤所有的JavaScript脚本；
- 转义Html元字符，使用htmlentities、htmlspecialchars等函数；
- 系统的扩展函数库提供了XSS安全过滤的remove_xss方法；
- 新版对URL访问的一些系统变量已经做了XSS处理。

其他安全建议

下面的一些安全建议也是非常重要的：

> - 对所有公共的操作方法做必要的安全检查，防止用户通过URL直接调用；
> - 不要缓存需要用户认证的页面；
> - 对用户的上传文件，做必要的安全检查，例如上传路径和非法格式；
> - 如非必要，不要开启服务器的目录浏览权限；
> - 对于项目进行充分的测试，不要生成业务逻辑的安全隐患（这可能是最大的安全问题）；
> - 最后一点，做好服务器的安全防护；

扩展

类库扩展

ThinkPHP的类库主要包括公共类库和应用类库，都是基于命名空间进行定义和扩展的。只要按照规范定义，都可以实现自动加载。

公共类库

公共类库通常是指 ThinkPHP/Library 目录下面的类库，例如：

```
Think目录：系统核心类库
Org目录：第三方公共类库
```

这些目录下面的类库都可以自动加载，你只要把相应的类库放入目录中，然后添加或者修改命名空间定义。 你可以在Org/Util/目录下面添加一个Image.class.php 文件，然后添加命名空间如下：

```
namespace Org\Util;
class Image {
}
```

这样，就可以用下面的方式直接实例化Image类了：

```
$image = new \Org\Util\Image;
```

除了这些目录之外，你完全可以在 ThinkPHP/Library 目录下面添加自己的类库目录，例如，我们添加一个Com目录用于企业类库扩展：

Com\Sina\App类（位于Com/Sina/App.class.php ）

```
namespace Com\Sina;
class App {
}
```

Com\Sina\Rank类（位于Com/Sina/Rank.class.php）

```
namespace Com\Sina;
class Rank {
}
```

公共类库除了在系统的Library目录之外，还可以自定义其他的命名空间，我们只需要注册一个新的命名空间，在应用或者模块配置文件中添加下面的设置参数：

```
'AUTOLOAD_NAMESPACE' => array(
    'Lib'     => APP_PATH.'Lib',
)
```

我们在应用目录下面创建了一个Lib目录用于放置公共的Lib扩展，如果我们要把上面两个类库放到Lib\Sina目录下面，只需要调整为：

Lib\Sina\App类（位于Lib/Sina/App.class.php ）

```
namespace Lib\Sina;
class App {
}
```

Lib\Sina\Rank类（位于Lib/Sina/Rank.class.php）

```
namespace Lib\Sina;
class Rank {
}
```

如果你的类库没有采用命名空间的话，需要使用import方法先加载类库文件，然后再进行实例化，例如：我们定义了一个Counter类（位于Com/Sina/Util/Counter.class.php）：

```
class Counter {
}
```

在使用的时候，需要按下面方式调用：

```
import('Com.Sina.Util.Couter');
$object = new \Counter();
```

应用类库

应用类库通常是在应用或者模块目录下面的类库，应用类库的命名空间一般就是模块的名称为根命名空间，例如： Home\Model\UserModel类（位于Application\Home\Model）

```
namespace Home\Model;
use Think\Model;
class UserModel extends Model{
}
```

Common\Util\Pay类（位于Application\Common\Util）

```
namespace Common\Util;
class Pay {
}
```

Admin\Api\UserApi类（位于Application\Admin\Api）

```
namespace Admin\Api;
use Think\Model;
class UserApi extends Model{
}
```

> 记住一个原则，命名空间的路径和实际的文件路径对应的话 就可以实现直接实例化的时候自动加载。

驱动扩展

这里说的驱动扩展是一种泛指，ThinkPHP采用驱动式设计，很多功能的扩展都是基于驱动的思想，包括数据库驱动、缓存驱动、标签库驱动和模板引擎驱动等。

事实上，每个类库都可以设计自己的驱动，因此3.2版本的驱动目录没有独立出来，而是放到各个类库的命名空间下面，例如：`Think\Log` 类的驱动放到 `Think\Log\Driver` 命名空间下面，`Think\Db` 类的驱动放到了 `Think\Db\Driver` 命名空间下面。

> 当然，这只是建议的位置，你完全可以根据项目的需要，把自己的驱动独立存放，例如：`Home\Driver\Cache\Sae.class.php` 则是一种把Cache驱动独立存放的方式（内置的核心类库都支持给驱动指定单独的命名空间）。

缓存驱动

缓存驱动默认位于`Think\Cache\Driver`命名空间下面,目前已经提供了包括APC、Db、Memcache、Shmop、Sqlite、Redis、Eaccelerator和Xcache缓存方式的驱动扩展，缓存驱动必须继承Think\Cache类，并实现下面的驱动接口：

方法说明	接口方法
架构方法	__construct($options='')
读取缓存	get($name)

方法说明	接口方法
写入缓存	set($name,$value,$expire=null)
删除缓存	rm($name)
清空缓存	clear()

下面是一个典型的缓存驱动类定义：

```
namespace Think\Cache\Driver;
use Think\Cache;
/**
 * Test缓存驱动
 */
class Test extends Cache {
    /**
     * 读取缓存
     * @access public
     * @param string $name 缓存变量名
     * @return mixed
     */
     public function get($name) {
        // 获取名称为name的缓存
     }
    /**
     * 写入缓存
     * @access public
     * @param string $name 缓存变量名
     * @param mixed $value  存储数据
     * @param integer $expire  有效时间（秒）
     * @return boolean
     */
     public function set($name, $value, $expire = null) {
        // 设置缓存
     }
    /**
     * 删除缓存
     * @access public
     * @param string $name 缓存变量名
     * @return boolean
     */
     public function rm($name) {
        // 删除名称为name的缓存
     }

    /**
     * 清除缓存
     * @access public
     * @return boolean
     */
    public function clear() {
        // 清空缓存
    }
}
```

> 注意：缓存驱动的有效期参数约定，如果设置为0 则表示永久缓存。

如果要让缓存驱动支持缓存队列功能，需要在缓存接口的set操作方法设置成功后添加如下代码：

```
if($this->options['length']>0) {
        // 记录缓存队列
        $this->queue($name);
 }
```

要配置当前默认的缓存驱动类型可以使用CACHE_TYPE参数，例如：

```
'CACHE_TYPE'=>'test'
```

数据库驱动

默认的数据库驱动位于 Think\Db\Driver 命名空间下面，驱动类必须继承 Think\Db 类，每个数据库驱动必须要实现的接口方法包括（具体参数可以参考现有的数据库驱动类库）：

驱动方法	方法说明
架构方法	__construct($config='')
数据库连接方法	connect($config='',$linkNum=0,$force=false)
释放查询方法	free()
查询操作方法	query($str)
执行操作方法	execute($str)
开启事务方法	startTrans()
事务提交方法	commit()
事务回滚方法	rollback()
获取查询数据方法	getAll()
获取字段信息方法	getFields($tableName)
获取数据库的表	getTables($dbName='')
关闭数据库方法	close()
获取错误信息方法	error()
SQL安全过滤方法	escapeString($str)

数据库的CURD接口方法（通常这些方法无需重新定义）

方法	说明
写入	insert($data,$options=array(),$replace=false)
更新	update($data,$options)

方法	说明
删除	delete($options=array())
查询	select($options=array())

介于不同数据库的查询方法存在区别，所以经常需要对查询的语句进行重新定义，这就需要修改针对查询的 selectSql 属性。该属性定义了当前数据库驱动的查询表达式，默认的定义是：

```
'SELECT%DISTINCT% %FIELD% FROM %TABLE%%JOIN%%WHERE%%GROUP%%HAVING%%ORDER%%LIMIT% %UNION%'
```

驱动可以更改或者删除个别查询定义，或者更改某个替换字符串的解析方法，这些方法包括：

方法名	说明	对应
parseTable	数据库表名解析	%TABLE%
parseWhere	数据库查询条件解析	%WHERE%
parseLimit	数据库查询Limit解析	%LIMIT%
parseJoin	数据库JOIN查询解析	%JOIN%
parseOrder	数据库查询排序解析	%ORDER%
parseGroup	数据库group查询解析	%GROUP%
parseHaving	数据库having解析	%HAVING%
parseDistinct	数据库distinct解析	%DISTINCT%
parseUnion	数据库union解析	%UNION%
parseField	数据库字段解析	%FIELD%

驱动的其他方法根据自身驱动需要和特性进行添加，例如，有些数据库的特殊性，需要覆盖父类Db类中的解析和过滤方法，包括：

方法名	说明
parseKey	数据库字段名解析
parseValue	数据库字段值解析
parseSet	数据库set分析
parseLock	数据库锁机制

定义了驱动扩展后，需要使用的时候，设置相应的数据库类型即可：

```
'DB_TYPE'=>'odbc', // 数据库类型配置不区分大小写
```

日志驱动

日志驱动默认的命名空间位于`Think\Log\Driver`,驱动类需要实现的接口方法包括：

方法	说明
架构方法	__construct($config=array())
写入方法	write($log,$destination='')

> 日志驱动只需要实现写入方法即可，log参数是日志信息字符串。

Session驱动

默认的session驱动的命名空间是`Think\Session\Driver`,并实现下面的驱动接口：

方法说明	接口方法
打开Session	open($savePath, $sessionName)
关闭Session	close()
读取Session	read($id)
写入Session	write($id, $data)
删除Session	destory($id)
Session 过期回收	gc($maxlifetime)

假设我们实现了一个Db类型的session驱动，那么只需要在配置文件中使用：

```
'SESSION_TYPE'=>'Db'
// 或者
'SESSION_OPTIONS'=>array(
    'type'=>'Db',
)
```

系统在初始化Session的时候会自动处理，采用Db机制来处理session。

存储驱动

存储驱动完成了不同环境下面的文件存取操作，也是ThinkPHP支持分布式和云平台的基础。

默认的存储驱命名空间位于 Think\Storage\Driver ,每个存储驱动必须继承Think\Storage，并且实现下列接口方法（具体参数可以参考现有的存储驱动类库）：

驱动方法	方法说明
架构方法	__construct($config='')
读取文件内容	read($filename,$type='')
写文件	put($filename,$content,$type='')
文件追加	append($filename,$content,$type='')
加载文件	load($filename,$vars=null,$type='')
判断文件是否存在	has($filename,$type='')
删除文件	unlink($filename,$type='')
读取文件信息	get($filename,$name,$type='')

其中type参数是为了区分不同的读写场景而设置的。

目前使用到的场景包括 runtime（用于编译缓存的文件操作）、html（用于静态缓存的文件操作）和F（用于F函数的文件操作）。

要使用自己定义的存储驱动的话，需要在应用入口文件定义：

```
define('STORAGE_TYPE', 'MyStorage');
```

> 存储类型的特殊性决定了我们只能在入口文件中改变存储类型

模板引擎驱动

模板引擎驱动完成了第三方模板引擎的支持，通过定义模板引擎驱动，我们可以支持Smarty、TemplateLite、SmartTemplate和EaseTemplate等第三方模板引擎。

默认的模板引擎驱动的命名空间位于 Think\Template\Driver ，需要实现的接口方法只有一个 fetch($templateFile,$var) 用于渲染模板文件并输出。

下面是一个Smarty模板引擎扩展的示例：

```
namespace Think\Template\Driver;
class Smarty {

    /**
     * 渲染模板输出
     * @access public
     * @param string $templateFile 模板文件名
     * @param array $var 模板变量
     * @return void
     */
    public function fetch($templateFile,$var) {
        $templateFile   =   substr($templateFile,strlen(THEME_PATH));
        vendor('Smarty.Smarty#class');
        $tpl            =   new \Smarty();
        $tpl->caching       = C('TMPL_CACHE_ON');
        $tpl->template_dir  = THEME_PATH;
        $tpl->compile_dir   = CACHE_PATH ;
        $tpl->cache_dir     = TEMP_PATH ;
        if(C('TMPL_ENGINE_CONFIG')) {
            $config  =  C('TMPL_ENGINE_CONFIG');
            foreach ($config as $key=>$val){
                $tpl->{$key}   =  $val;
            }
        }
        $tpl->assign($var);
        $tpl->display($templateFile);
    }
}
```

如果要使用Smarty模板引擎的话，只需要配置

```
'TMPL_ENGINE_TYPE'=>'Smarty',
'TMPL_ENGINE_CONFIG'=>array(
    'plugins_dir'=>'./Application/Smarty/Plugins/',
),
```

标签库驱动

任何一个模板引擎的功能都不可能是为你量身定制的，具有一个良好的可扩展机制也是模板引擎的另外一个考量，Smarty采用的是插件方法来实现扩展，Think\Template由于采用了标签库技术，比Smarty提供了更为强大的定制功能，和Java的TagLibs一样可以支持自定义标签库和标签，每个标签都有独立的解析方法，所以可以根据标签库的定义规则来增加和修改标签解析规则。

在Think\Template中标签库的体现是采用XML命名空间的方式。每个标签库对应一个标签库驱动类，每个驱动类负责对标签库中的所有标签的解析。

标签库驱动类的作用其实就是把某个标签定义解析成为有效的模版文件（可以包括PHP语句或者HTML标签），标签库驱动的命名空间位于 Think\Template\TagLib ，标签库驱动必须继承 Think\Template\TagLib 类，例如：

```
namespace Think\Template\Taglib;
use Think\Template\TagLib;
Class Test extends TagLib{
}
```

首先需要定义标签库的标签定义，标签定义包含了所有标签库中支持的所有标签，定义方式如下：

```
protected $tags   =  array(
  // 定义标签
 'input'    =>    array('attr'=>'type,name,id,value','close'=>0), // input标签
 'textarea' =>    array('attr'=>'name,id')，
 );
```

标签库的所有支持标签都在tags属性中进行定义，tags属性是一个二维数组，每个元素就是一个标签定义，索引名就是标签名，采用小写定义，调用的时候不区分大小写。

每个标签定义支持的属性包括：

属性名	说明
attr	标签支持的属性列表，用逗号分隔
close	标签是否为闭合方式（0闭合 1不闭合），默认为不闭合
level	标签的嵌套层次（只有不闭合的标签才有嵌套层次）
alias	标签别名

定义了标签属性后，就需要定义每个标签的解析方法了，每个标签的解析方法在定义的时候需要添加“_”前缀，传入两个参数，对应属性数组和内容字符串（针对非闭合标签）。必须返回标签的字符串解析输出，在标签解析类中可以调用模板类的实例。下面是一个input解析方法的定义：

```
// input标签解析
public function _input($tag,$content)  {
    $name  =  $tag['name'];
    $id  =  $tag['id'];
    $type  =  $tag['type'];
    $value  =  $this->autoBuildVar($tag['value']);
    $str = "<input type='".$type."' id='".$id."' name='".$name."' value='".$value."' />";
    return $str;
}
// textarea标签解析
public function _textarea($tag,$content)  {
    $name  =  $tag['name'];
    $id  =  $tag['id'];
    $str  =  '<textarea id="'.$id.'" name="'.$name.'">'.$content.'</textarea>';
    return $str;
}
```

定义好标签库扩展之后，我们就可以在模板中使用了，首先我们必须要告诉模板申明Test标签库，用taglib标签，例如：

```
<taglib name='Test' />
```

name属性支持申明多个标签库，用逗号分隔即可。申明Test标签库之后，就可以使用Test标签库中的所有标签库了，调用方式如下：

```
<test:input type='radio' id='test' name='mail' value='value' />
<test:textarea id="content" name="content">$value</test:textarea>
```

> 注意：调用扩展标签库的标签的时候，必须加上标签库的XML命名空间前缀。

Input标签定义value属性可以支持变量传入，所以value被认为是一个变量名，如果在控制器中已经给value模板变量赋值，例如：

```
$this->assign('value','my test value');
```

最后标签被模板引擎编译后，就会输出：

```
<input type='radio' id='test' name='mail' value='my test value' />
<textarea id="content" name="content">my test vale</textarea>
```

行为扩展

行为（Behavior）是一个比较抽象的概念，你可以想象成在应用执行过程中的一个动作或者处理，在框架的执行流程中，各个位置都可以有行为产生，例如路由检测是一个行为，静态缓存是一个行为，用户权限检测也是行为，大到业务逻辑，小到浏览器检测、多语言检测等等都可以当做是一个行为，甚至说你希望给你的网站用户的第一次访问弹出Hello，world！这些都可以看成是一种行为，行为的存在让你无需改动框架和应用，而在外围通过扩展或者配置来改变或者增加一些功能。

而不同的行为之间也具有位置共同性，比如，有些行为的作用位置都是在应用执行前，有些行为都是在模板输出之后，我们把这些行为发生作用的位置称之为标签（位），当应用程序运行到这个标签的时候，就会被拦截下来，统一执行相关的行为，类似于AOP编程中的“切面”的概念，给某一个切面绑定相关行为就成了一种类AOP编程的思想。

系统核心提供的标签位置包括下面几个（按照执行顺序排列）：

```
app_init    应用初始化标签位
path_info   PATH_INFO检测标签位
app_begin   应用开始标签位
action_name 操作方法名标签位
action_begin    控制器开始标签位
view_begin  视图输出开始标签位
view_parse  视图解析标签位
template_filter 模板内容解析标签位
view_filter 视图输出过滤标签位
view_end    视图输出结束标签位
action_end  控制器结束标签位
app_end         应用结束标签位
```

在每个标签位置，可以配置多个行为定义，行为的执行顺序按照定义的顺序依次执行。除非前面的行为里面中断执行了（某些行为可能需要中断执行，例如检测机器人或者非法执行行为），否则会继续下一个行为的执行。 行为定义： 通过Common\Conf\tags.php配置文件定义，格式如下：

```
<?php
return array(
    'action_begin'=>array('Home\\Behaviors\\TestBehavior','Home\\Behaviors\\Test1Behavior'),
);
```

上面注册了两个行为，分别是Home模块下的test和test1行为，类文件位于Home模块目录下的Behaviors目录，可以自定义目录。

行为必须是一个包含命名空间路径的类，如上的 Home\Behaviors\TestBehavior 对应的类是 Home/Behaviors/TestBehavior.class.php。

除了这些系统内置标签之外，开发人员还可以在应用中添加自己的应用标签。 比如在控制器的_initialize方法中：

```
\Think\Hook::add('action_begin','Home\\Behaviors\\TestBehavior');//同时添加多个行为，只要将第二个参数换成数组即可。
```

行为类的定义,以上面的test行为为例：

```
<?php
namespace Home\Behaviors;
class TestBehavior extends \Think\Behavior{
    //行为执行入口
    public function run(&$param){

    }
}
```

行为类建议继承\Think\Behavior，必须实现run(&$param)方法，行为是通过这个方法执行的。

行为的触发： 只要在合适的地方通过以下代码

```
\Think\Hook::listen('标签名'[,参数]);
// 或者
// tag('标签名'[,参数]);
```

当应用执行到这个地方的时候将自动触发指定标签名下的所有行为类。

> 注意：动态注册的行为必须在Hook::listen之前，即：先注册行为，才能触发行为。
>
> listen方法可以传入并且只接受一个参数，如果需要传入多个参数，请使用数组，该参数为引用传值，所以只能传入变量。 参数可以被run(&$param)中的$param接收。

标签扩展

标签库加载

模板中加载标签库，预加载自定义标签库，扩展内置标签库的加载 请参考：http://document.thinkphp.cn/manual_3_2.html#taglib

自定义标签库开发注意事项

- 标签库请放置 ThinkPHP\Library\Think\Template\TagLib 目录下，若需要存放在指定位置，请在加载标签库配置时使用命名空间，如

```
<taglib name="Home\\TagLib\\MyTag"/>
```

- 标签库类请使用命名空间，否则无法加载类，主要是在类开头包含如下代码：

```
<?php
    namespace Think\Template\TagLib;
    use Think\Template\TagLib;
    defined('THINK_PATH') or exit();
```

自定义标签库位置的话，请自行修改第一行 namespace 的定义。

关于标签库开发

暂时可以参考3.0的官方教程中关于标签库扩展的部分，建议自己分析内置标签库Cx。

Widget扩展

Widget扩展一般用于页面组件的扩展。

举个例子，我们在页面中实现一个分类显示的Widget，首先我们要定义一个Widget控制器层CateWidget，如下：

```
namespace Home\Widget;
use Think\Controller;
class CateWidget extends Controller {
    public function menu(){
        echo 'menuWidget';
    }
}
```

然后，我们在模版中通过W方法调用这个Widget。

```
{:W('Cate/Menu')}
```

执行后的输出结果是： menuWidget

传入参数

如果需要在调用Widget的时候 使用参数，可以这样定义：

```
namespace Home\Widget;
use Think\Controller;
class CateWidget extends Controller {
    public function menu($id,$name){
        echo $id.':'.$name;
    }
}
```

模版中的参数调用，使用：

```
{:W('Cate/Menu',array(5,'thinkphp'))}
```

> 传入的参数是一个数组，顺序对应了menu方法定义的参数顺序。

则会输出

```
5:thinkphp
```

模板支持

Widget可以支持使用独立的模板，例如：

```
namespace Home\Widget;
use Think\Controller;
class CateWidget extends Controller {
    public function menu(){
        $menu = M('Cate')->getField('id,title');
        $this->assign('menu',$menu);
        $this->display('Cate:menu');
    }
}
```

CateWiget类渲染了一个模版文件 `View/Cate/menu.html` 。 在menu.html模版文件中的用法：

```
<foreach name="menu" item="title">
{$key}:{$title}
</foreach>
```

应用模式

应用模式提供了对核心框架进行改造的机会，可以让你的应用适应更多的环境和不同的要求。

每个应用模式有自己的模式定义文件，用于配置当前模式需要加载的核心文件和配置文件，以及别名定义、行为扩展定义等等。根据模式定义文件的定义位置和入口是否需要定义模式，可以分为显式应用模式和隐含应用模式。

显式应用模式

显式应用模式的模式定义文件位于 ThinkPHP\Mode 目录，如果我们要增加一个应用模式，只需要在该目录下面定义一个模式定义文件即可，下面是一个典型的模式定义文件（lite.php）：

```
return array(
    // 配置文件
    'config'    =>  array(
        THINK_PATH.'Conf/convention.php',   // 系统惯例配置
        CONF_PATH.'config.php',      // 应用公共配置
    ),

    // 别名定义
    'alias'     =>  array(
        'Think\Exception'         => CORE_PATH . 'Exception'.EXT,
        'Think\Model'             => CORE_PATH . 'Model'.EXT,
        'Think\Db'                => CORE_PATH . 'Db'.EXT,
        'Think\Cache'             => CORE_PATH . 'Cache'.EXT,
        'Think\Cache\Driver\File' => CORE_PATH . 'Cache/Driver/File'.EXT,
        'Think\Storage'           => CORE_PATH . 'Storage'.EXT,
    ),

    // 函数和类文件
    'core'      =>  array(
        MODE_PATH.'Lite/functions.php',
        COMMON_PATH.'Common/function.php',
        MODE_PATH . 'Lite/App'.EXT,
        MODE_PATH . 'Lite/Dispatcher'.EXT,
        MODE_PATH . 'Lite/Controller'.EXT,
        MODE_PATH . 'Lite/View'.EXT,
        CORE_PATH . 'Behavior'.EXT,
    ),
    // 行为扩展定义
    'tags'  =>  array(
        'view_parse'    =>  array(
            'Behavior\ParseTemplate', // 模板解析 支持PHP、内置模板引擎和第三方模板引擎
        ),
        'template_filter'=> array(
            'Behavior\ContentReplace', // 模板输出替换
        ),
    ),
);
```

我们在ThinkPHP/Mode/Lite目录下面创建 functions.php 函数库文件，以及 App.class.php 、 Dispatcher.class.php 、 Controller.class.php 和 View.class.php ，这些类都是针对我们新的应用模式定制的核心类，但是和标准模式的命名空间是一致的，也就是说都在Think命名空间下面。

> ThinkPHP/Mode/Lite目录用于存放该应用模式下面的所有自定义文件。

应用模式定义文件定义好后，我们就可以在应用中使用该模式了，例如：

```
define('MODE_NAME','lite');
define('APP_PATH','./Application/');
require './ThinkPHP/ThinkPHP.php';
```

隐含应用模式

隐含应用模式的模式定义文件位于应用的配置目录下面 `Application/Common/Conf/core.php` ，模式定义文件和显式应用模式的定义文件一样。 使用隐含应用模式的时候，不需要在入口文件中定义MODE_NAME，或者说存在隐含应用模式定义文件的时候，MODE_NAME定义无效。

> 注意：如果应用中定义的应用模式需要使用其他的存储类型，需要在入口文件中定义。

```
define('STORAGE_TYPE','Bae');
```

部署

PATH_INFO支持

> 如果发生在本地测试正常，但是一旦部署到服务器环境后会发生只能访问首页的情况，很有可能是你的服务器或者空间不支持PATH_INFO所致。

系统内置提供了对PATH_INFO的兼容判断处理，但是不能确保在所有的环境下面都可以支持。如果你确认你的空间不支持PATH_INFO的URL方式的话，有下面几种方式可以处理：

修改URL_PATHINFO_FETCH配置参数新版内置了通过对 ORIG_PATH_INFO，REDIRECT_PATH_INFO，REDIRECT_URL 三个系统$_SERVER变量的判断处理来兼容读取$_SERVER['PATH_INFO']，如果你的主机环境有更特殊的设置，可以修改URL_PATHINFO_FETCH参数，改成你的环境配置对应的PATH_INFO的系统变量兼容获取名称，例如：

```
'URL_PATHINFO_FETCH' => 'ORIG_PATH_INFO,REDIRECT_URL,其他参数...'
```

如果你的环境没有任何对应的系统变量，那么可以封装一个获取方法，例如：

```
function get_path_info(){
    // 根据你的环境兼容获取PATH_INFO 具体代码略
    return $path; // 直接返回获取到的PATH_INFO信息
}
```

然后我们修改下URL_PATHINFO_FETCH参数的配置值，改为：

```
'URL_PATHINFO_FETCH'    =>   ':get_path_info'
```

配置后，系统会自动读取get_path_info方法来获取$_SERVER['PATH_INFO']的值。

配置你的WEB服务器重写规则模拟PATH_INFO实现如果你有服务器或者空间的配置权限，可以考虑通过配置URL重写规则来模拟实现。 具体可以参考后面URL重写中的内容。

采用兼容URL模式运行这是最坏的方法，配置你的URL模式为3（表示兼容URL模式）然后在需要生成URL的地方采用U方法动态生成即可。

URL重写

可以通过URL重写隐藏应用的入口文件index.php,下面是相关服务器的配置参考：

[Apache]

1. httpd.conf配置文件中加载了mod_rewrite.so模块
2. AllowOverride None 将None改为 All
3. 把下面的内容保存为.htaccess文件放到应用入口文件的同级目录下

```
<IfModule mod_rewrite.c>
 RewriteEngine on
 RewriteCond %{REQUEST_FILENAME} !-d
 RewriteCond %{REQUEST_FILENAME} !-f
 RewriteRule ^(.*)$ index.php/$1 [QSA,PT,L]
</IfModule>
```

[IIS]

如果你的服务器环境支持ISAPI_Rewrite的话，可以配置httpd.ini文件，添加下面的内容：

```
RewriteRule (.*)$ /index\.php\?s=$1 [I]
```

在IIS的高版本下面可以配置web.Config，在中间添加rewrite节点：

```
<rewrite>
 <rules>
 <rule name="OrgPage" stopProcessing="true">
 <match url="^(.*)$" />
 <conditions logicalGrouping="MatchAll">
 <add input="{HTTP_HOST}" pattern="^(.*)$" />
 <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" />
 <add input="{REQUEST_FILENAME}" matchType="IsDirectory" negate="true" />
 </conditions>
 <action type="Rewrite" url="index.php/{R:1}" />
 </rule>
 </rules>
 </rewrite>
```

[Nginx]

在Nginx低版本中，是不支持PATHINFO的，但是可以通过在Nginx.conf中配置转发规则实现：

```
 location / { // .....省略部分代码
  if (!-e $request_filename) {
  rewrite  ^(.*)$  /index.php?s=$1  last;
  break;
   }
}
```

> 其实内部是转发到了ThinkPHP提供的兼容模式的URL，利用这种方式，可以解决其他不支持PATHINFO的WEB服务器环境。

如果你的ThinkPHP安装在二级目录，Nginx的伪静态方法设置如下，其中youdomain是所在的目录名称。

```
location /youdomain/ {
    if (!-e $request_filename){
        rewrite  ^/youdomain/(.*)$  /youdomain/index.php?s=$1  last;
    }
}
```

原来的访问URL：

`http://serverName/index.php/模块/控制器/操作/[参数名/参数值...]`

设置后，我们可以采用下面的方式访问：

`http://serverName/模块/控制器/操作/[参数名/参数值...]`

> 默认情况下，URL地址中的模块不能省略，如果你需要简化某个模块的URL访问地址，可以通过设置模块列表和默认模块或者采用子域名部署到模块的方式解决，请参考后面的模块和域名部署部分。

模块部署

3.2对模块的访问是自动判断的，所以通常情况下无需配置模块列表即可访问，在部署模块的时候，默认情况下都是基于类似于子目录的URL方式来访问模块的，例如：

```
http://serverName/Home/New/index //访问Home模块
http://serverName/Admin/Config/index //访问Admin模块
http://serverName/User/Member/index //访问User模块
```

允许模块列表

如果直接访问：`http://serverName/New/index` 会报错，不过通过下面的设置可以把Home模块的访问

URL地址简化：

```
// 允许访问的模块列表
'MODULE_ALLOW_LIST'    =>    array('Home','Admin','User');
'DEFAULT_MODULE'       =>    'Home',  // 默认模块
```

这个时候再次访问 http://serverName/New/index 就不会报错了，并且实际访问的就是Home模块。默认情况下， MODULE_ALLOW_LIST 为空，表示允许任何模块的访问，不过最终是否允许访问还受 MODULE_DENY_LIST 参数的影响。

> 域名绑定的模块不受 MODULE_ALLOW_LIST 的影响

禁止模块访问

如果你的应用有很多的模块，你只是想禁止访问个别模块的话，可以配置禁止访问的模块列表（用于被其他模块调用或者不开放访问），默认配置中是禁止访问Common模块和Runtime模块（Runtime目录是默认的运行时目录），我们可以增加其他的禁止访问模块列表：

```
// 设置禁止访问的模块列表
 'MODULE_DENY_LIST'      =>  array('Common','Runtime','User'),
```

这个时候，我们再访问 http://serverName/User/Member/index 的话，就会报错。

> 域名绑定的模块同样不受 MODULE_DENY_LIST 影响

模块映射

如果不希望用户直接访问某个模块，可以设置模块映射（对后台的保护会比较实用）。

```
'URL_MODULE_MAP'    =>    array('test'=>'admin'),
```

> 注意：设置了模块映射后，原来的Admin模块将不能访问，只能访问test模块。

我们访问 http://serverName/Admin 将会报模块不存在的错误，而 http://serverName/test 则可以正常访问Admin模块。

如果你同时还设置了 MODULE_ALLOW_LIST 参数的话，必须将允许模块列表中的原来的模块改成映射后的模块名，例如：

```
'MODULE_ALLOW_LIST'    =>    array('Home','Test','User'),
'DEFAULT_MODULE'       =>    'Home',
'URL_MODULE_MAP'       =>    array('test'=>'admin'),
```

> 模块映射的模块必须使用小写定义

域名部署

ThinkPHP支持模块（甚至可以包含控制器）的完整域名、子域名和IP部署功能，让你的模块变得更加灵活，模块绑定到域名或者IP后，URL地址中的模块名称就可以省略了，所以还可以起到简化URL的作用。

开启域名部署

无论是子域名还是IP部署，首先要在应用配置文件中开启`APP_SUB_DOMAIN_DEPLOY`，这是前提，然后配置域名部署规则`APP_SUB_DOMAIN_RULES`。

```
'APP_SUB_DOMAIN_DEPLOY'   =>    1, // 开启子域名或者IP配置
'APP_SUB_DOMAIN_RULES'    =>    array(
    /* 域名部署配置
    *格式1: '子域名或泛域名或IP'=> '模块名[/控制器名]';
    *格式2: '子域名或泛域名或IP'=> array('模块名[/控制器名]','var1=a&var2=b&var3=*');
    */
)
```

域名部署的定义格式2和1的区别在于格式2可以隐式传入额外的参数。

域名和IP的解析涉及到DNS解析以及Apache等服务器的配置，这块不再详细描述，请参考相关百度资料。

> 域名或者IP部署到模块并不需要设置模块访问列表。

完整域名部署

可以在域名规则中直接定义完整的域名，例如：

```
'APP_SUB_DOMAIN_DEPLOY'   =>    1, // 开启子域名配置
'APP_SUB_DOMAIN_RULES'    =>    array(
    'admin.domain1.com'  => 'Admin',  // admin.domain1.com域名指向Admin模块
    'test.domain2.com'   => 'Test',  // test.domain2.com域名指向Test模块
),
```

在域名部署之前的访问地址：`http://www.domain.com/Admin/Index/index` 和 `http://www.domain.com/Test/Index/index`

域名部署后的访问地址变成：`http://admin.domain1.com/Index/index` 和 `http://test.domain2.com/Index/index`

子域名部署

子域名部署包括任意级子域名的支持，在你的应用配置文件中增加如下配置参数：

```
'APP_SUB_DOMAIN_DEPLOY'   =>    1, // 开启子域名配置
'APP_SUB_DOMAIN_RULES'    =>    array(
    'admin'       => 'Admin',  // admin子域名指向Admin模块
    'test'        => 'Test',  // test子域名指向Test模块
),
```

部署之前的访问地址：`http://www.domain.com/Admin/Index/index`

部署后的访问地址变成： `http://admin.domain.com/Index/index`

如果你的部署域名后缀是二级后缀，例如 `com.cn` 、 `net.cn` 或者 `org.cn` 之类的话，为了让系统更好的识别你的子域名，需要配置 APP_DOMAIN_SUFFIX 如下：

```
'APP_DOMAIN_SUFFIX'=>'com.cn'
```

> APP_DOMAIN_SUFFIX参数不支持设置多个后缀，如果你是一级域名后缀的话则该参数可以无需任何设置。

传入参数

子域名部署的时候，可以传入隐式的参数，例如：

```
'APP_SUB_DOMAIN_DEPLOY'   =>    1, // 开启子域名配置
'APP_SUB_DOMAIN_RULES'    =>    array(
    'admin'       => array('Admin','var1=1&var2=2'),  // admin子域名指向Admin模块
),
```

访问 `http://admin.domain.com/Index/index` 的同时会传入 `$_GET['var1'] = 1` 和 `$_GET['var2'] = 2` 两个参数。

控制器绑定

子域名部署还可以支持绑定某个控制器，例如：

```
'APP_SUB_DOMAIN_DEPLOY'   =>    1, // 开启子域名配置
'APP_SUB_DOMAIN_RULES'    =>    array(
    'test.admin'   => 'Admin/Test',  // test.admin子域名指向Admin模块的Test控制器
),
```

部署之前的访问地址： `http://www.domain.com/Admin/Test/index`

部署后的访问地址： http://test.admin.domain.com/index

泛域名部署

如果要部署某个模块到泛域名支持，可以使用：

```
'APP_SUB_DOMAIN_DEPLOY'   =>    1, // 开启子域名配置
'APP_SUB_DOMAIN_RULES'    =>    array(
    'admin'    =>    'Admin',  // admin域名指向Admin模块
    '*'        =>    array('Test','var1=1&var2=*'), // 二级泛域名指向Test模块
    '*.user'   =>    array('User','status=1&name=*'),  // 三级泛域名指向User模块
),
```

配置后，我们可以访问：

```
http://hello.domain.com/Index/index
```

访问Test模块 并隐式传入 $_GET['var1'] = 1 和 $_GET['var2'] = 'hello' 两个参数。

访问如下地址：

```
http://think.user.domain.com/Index/index
```

访问User模块，并隐式传入 $_GET['status'] = 1 和 $_GET['name'] = 'think' 两个参数。

在配置传入参数的时候，如果需要使用当前的泛域名作为参数，可以直接设置为 “*” 即可。

> 目前只支持二级域名和三级域名的泛域名部署。

IP访问部署

可以为某些模块配置IP访问规则，例如：

```
'APP_SUB_DOMAIN_DEPLOY'   =>    1, // 开启子域名配置
'APP_SUB_DOMAIN_RULES'    =>    array(
    '22.56.78.9'  => 'Admin',  // 22.56.78.9指向Admin模块
),
```

入口绑定

入口绑定是指在应用的入口文件中绑定某个模块，甚至还可以绑定某个控制器和操作，用来简化URL地址的访问。

绑定模块

例如，我们定义了一个入口文件admin.php，希望可以直接访问Admin模块，那么我们就可以在admin.php中进行模块绑定，定义如下：

```
// 绑定访问Admin模块
define('BIND_MODULE','Admin');
// 定义应用目录
define('APP_PATH','./Application/');
require './ThinkPHP/ThinkPHP.php';
```

> 在3.2.0版本中常量定义需要改成：

```
$_GET['m'] = 'Admin';
```

在入口文件中绑定模块后，访问的URL地址中就不需要传入模块名称了。
假设我们要访问Admin模块的Index控制器的test操作方法，访问地址如下：
`http://serverName/admin.php/Index/test/var/name` 如果访问
`http://serverName/admin.php/Admin/Index/test/var/name` 就会出现下面的错误提示：

绑定控制器

和绑定模块一样，我们还可以绑定控制器（一般是和模块绑定结合使用）。 例如：

```
// 绑定访问Admin模块
define('BIND_MODULE','Admin');
// 绑定访问Index控制器
define('BIND_CONTROLLER','Index');
// 定义应用目录
define('APP_PATH','./Application/');
require './ThinkPHP/ThinkPHP.php';
```

我们前面的URL访问就可以换成： `http://serverName/admin.php/test/var/name`

> 在3.2.0版本中常量定义需要改成：

```
$_GET['m'] = 'Admin';
$_GET['c'] = 'Index';
```

绑定操作

原则上，我们还可以在入口文件中绑定操作（虽然这种情况实际使用中不多见）。

```
// 绑定访问Admin模块
define('BIND_MODULE','Admin');
// 绑定访问Index控制器
define('BIND_CONTROLLER','Index');
// 绑定访问test操作
define('BIND_ACTION','test');
// 定义应用目录
define('APP_PATH','./Application/');
require './ThinkPHP/ThinkPHP.php';
```

我们前面的URL访问就可以换成： `http://serverName/admin.php/var/name`

> 3.2.0版本不支持操作绑定。

替换入口

3.2版本支持根据当前的运行环境生成Lite文件，可以替换框架的入口文件或者应用入口文件，提高运行效率。

> 我们的建议是在生产环境中关闭调试模式后生成Lite文件。
> 注意，目前SAE平台不支持直接生成Lite文件。

生成Lite文件

要生成Lite文件，需要在入口文件中增加常量定义：

```
define('BUILD_LITE_FILE',true);
```

默认情况下，再次运行后会在Runtime目录下面生成一个 `lite.php` 文件。

如果你需要修改Lite文件的位置或者名称，可以在应用配置文件中增加配置如下：

```
'RUNTIME_LITE_FILE'=> APP_PATH.'lite.php'
```

配置后，生成的Lite文件的位置为 `APP_PATH.'lite.php'` 。

Lite文件的编译文件内容是系统默认的，如果希望改变或者增加其他的编译文件的话，可以在外部定义编译列表文件，例如： 我们在应用配置目录下面增加lite.php定义如下：

```
return array(
    THINK_PATH.'Common/functions.php',
    COMMON_PATH.'Common/function.php',
    CORE_PATH . 'Think'.EXT,
    CORE_PATH . 'Hook'.EXT,
    CORE_PATH . 'App'.EXT,
    CORE_PATH . 'Dispatcher'.EXT,
    CORE_PATH . 'Model'.EXT,
    CORE_PATH . 'Log'.EXT,
    CORE_PATH . 'Log/Driver/File'.EXT,
    CORE_PATH . 'Route'.EXT,
    CORE_PATH . 'Controller'.EXT,
    CORE_PATH . 'View'.EXT,
    CORE_PATH . 'Storage'.EXT,
    CORE_PATH . 'Storage/Driver/File'.EXT,
    CORE_PATH . 'Exception'.EXT,
    BEHAVIOR_PATH . 'ParseTemplateBehavior'.EXT,
    BEHAVIOR_PATH . 'ContentReplaceBehavior'.EXT,
);
```

所有在lite.php文件中定义的文件都会纳入Lite文件的编译缓存中。你还可以对生成的lite文件进行修改。

> 如果你修改了框架文件和应用函数和配置文件的话，需要删除Lite文件重新生成。

由于SAE等云平台不支持文件写入，因此不支持直接生成Lite文件。

替换入口

Lite文件可以用于替换框架入口文件或者应用入口文件。

替换框架入口文件

Lite文件生成后，就可以把原来的应用入口文件中的框架入口文件修改如下：

```
require './ThinkPHP/ThinkPHP.php';
// 改成
require './Runtime/lite.php';
```

> 替换Lite文件后，应用编译缓存不再需要。

替换应用入口文件

如果你的入口文件没有其他代码和逻辑的话，还可以直接把lite.php文件作为应用的入口文件访问。 把lite.php 文件复制到应用入口文件的相同目录，并直接改名为index.php即可和原来一样正常访问（原来的应用入口文件可以备份以备用于重新生成Lite文件的时候使用）。

> 注意：如果你的环境或者目录位置发生变化，以及更改了核心框架和应用函数、配置等文件后，则需要重新生成Lite文件。

专题

SESSION支持

系统提供了Session管理和操作的完善支持，全部操作可以通过一个内置的session函数完成，该函数可以完成Session的设置、获取、删除和管理操作。

session初始化设置

如果session方法的第一个参数传入数组则表示进行session初始化设置，例如：

```
session(array('name'=>'session_id','expire'=>3600));
```

支持传入的session参数包括：

参数名	说明
id	session_id值
name	session_name 值
path	session_save_path 值
prefix	session 本地化空间前缀
expire	session.gc_maxlifetime 设置值
domain	session.cookie_domain 设置值
use_cookies	session.use_cookies 设置值
use_trans_sid	session.use_trans_sid 设置值
type	session处理类型，支持驱动扩展

Session初始化设置方法无需手动调用，在Think\App类的初始化工作结束后会自动调用，通常项目只需要配置 SESSION_OPTIONS 参数即可， SESSION_OPTIONS 参数的设置是一个数组，支持的索引名和前面的session初始化参数相同。

默认情况下，初始化之后系统会自动启动session，如果不希望系统自动启动session的话，可以设置 SESSION_AUTO_START 为false，例如：

```
'SESSION_AUTO_START' =>false
```

关闭自动启动后可以项目的公共文件或者在控制器中通过手动调用 session_start 或者 session('[start]')

启动session。

session赋值

Session赋值比较简单，直接使用：

```
session('name','value');  //设置session
```

3.2.3版本开始，session赋值操作支持二维，例如：

```
session('user.user_id',10);  //设置session
```

session取值

Session取值使用：

```
$value = session('name');
// 获取所有的session 3.2.2版本新增
$value = session();
```

3.2.3版本开始支持二维数组取值，例如：

```
$value = session('user.user_id');
```

session删除

删除某个session的值使用：

```
session('name',null); // 删除name
```

3.2.3版本开始支持删除二维数组，例如：

```
session('user.user_id',null); // 删除session
```

要删除所有的session，可以使用：

```
session(null); // 清空当前的session
```

session判断

要判断一个session值是否已经设置，可以使用

```
// 判断名称为name的session值是否已经设置
session('?name');
```

3.2.3版本开始，支持判断二维数组，例如：

```
session('?user.user_id');
```

session管理

session方法支持一些简单的session管理操作，用法如下：

```
session('[操作名]');
```

支持的操作名包括：

操作名	含义
start	启动session
pause	暂停session写入
destroy	销毁session
regenerate	重新生成session id

使用示例如下：

```
session('[pause]'); // 暂停session写入
session('[start]'); // 启动session
session('[destroy]'); // 销毁session
session('[regenerate]'); // 重新生成session id
```

本地化支持

如果在初始化session设置的时候传入 prefix 参数或者单独设置了 SESSION_PREFIX 参数的话，就可以启用本地化session管理支持。启动本地化session后，所有的赋值、取值、删除以及判断操作都会自动支持本地化session。

本地化session支持开启后，生成的session数据格式由原来的 $_SESSION['name'] 变成 $_SESSION['前缀']['name'] 。

session handler支持

初始化session设置的时候如果传入了 type 参数或者设置了 SESSION_TYPE 参数的话，则会自动引入对应的Session处理驱动，驱动目录位于Library/Think/Session/Driver目录下面（详见扩展部分）。

Cookie支持

系统内置了一个cookie函数用于支持和简化Cookie的相关操作，该函数可以完成Cookie的设置、获取、删除操作。

Cookie设置

```
cookie('name','value');  //设置cookie
cookie('name','value',3600); // 指定cookie保存时间
```

还可以支持参数传入的方式完成复杂的cookie赋值，下面是对cookie的值设置3600秒有效期，并且加上cookie前缀think_

```
cookie('name','value',array('expire'=>3600,'prefix'=>'think_'))
```

数组参数可以采用query形式参数

```
cookie('name','value','expire=3600&prefix=think_')
```

和上面的用法等效。

> 后面的参数支持 prefix,expire,path,domain和httponly（**3.2.2版本新增**）五个索引参数，如果没有传入或者传入空值的话，会默认取 COOKIE_PREFIX 、 COOKIE_EXPIRE 、 COOKIE_PATH 、 COOKIE_DOMAIN 和 COOKIE_HTTPONLY 五个配置参数。如果只传入个别参数，那么也会和默认的配置参数合并。

支持给cookie设置数组值（采用JSON编码格式保存），例如：

```
cookie('name',array('value1','value2'));
```

Cookie获取

获取cookie很简单，无论是怎么设置的cookie，只需要使用：

```
$value = cookie('name');
```

如果没有设置cookie前缀的话 相当于

```
$value = $_COOKIE['name'];
```

如果设置了cookie前缀的话，相当于

```
$value = $_COOKIE['前缀+name'];
```

如果要获取所有的cookie，可以使用：

```
$value = cookie();
```

该用法相当于

```
$value = $_COOKIE;
```

> 注意，该用法会返回所有的cookie而无论是否当前的前缀。

Cookie删除

删除某个cookie的值，使用：

```
cookie('name',null);
```

要删除所有的Cookie值，可以使用：

```
cookie(null); // 清空当前设定前缀的所有cookie值
cookie(null,'think_'); //  清空指定前缀的所有cookie值
```

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。

要启用多语言功能，需要配置开启多语言行为，在应用的配置目录下面的行为定义文件tags.php中，添加：

```
return array(
    // 添加下面一行定义即可
    'app_begin' => array('Behavior\CheckLangBehavior'),
);
```

表示在 app_begin 标签位置执行多语言检测行为。

要开启语言包功能，需要开启

```
'LANG_SWITCH_ON' => true,   // 开启语言包功能
```

其他的配置参数包括：

```
'LANG_AUTO_DETECT' => true, // 自动侦测语言 开启多语言功能后有效
'LANG_LIST'        => 'zh-cn', // 允许切换的语言列表 用逗号分隔
'VAR_LANGUAGE'     => 'l', // 默认语言切换变量
```

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和英文语言包en-us.php ，如果要增加繁体中文zh-tw或者其他语言支持，只要增加相应的语言定义文件。

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 语言配置变量的话，在程序中的写法就要改为：

```
throw_exception（L('ADD_USER_ERROR')）;
```

也就是说，字符串信息要改成L方法和语言定义来表示。

应用语言包文件位于应用公共模块下的Lang目录，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。

具体的语言包文件命名和位置如下：

语言包	语言文件位置
应用语言包	应用公共目录/Lang/语言文件.php
模块语言包	模块目录/Lang/语言文件.php
控制器语言包	模块目录/Lang/语言目录/控制器名（小写）.php

以当前模块为Home、当前语言为zh-cn为例，我们可以读取语言包的顺序如下（如果没有定义则不读取）：

```
ThinkPHP/Lang/zh-cn.php 框架底层语言包
Application/Common/Lang/zh-cn.php 应用公共语言包
Application/Home/Lang/zh-cn.php Home模块语言包
Application/Home/Lang/zh-cn/user.php Home模块的User控制器语言包
```

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，那可以定义应用的公共语言文件即可，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',
);
```

也可以在程序里面动态设置语言定义的值，使用下面的方式：

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

通常多语言的使用是在控制器里面，但是模型类的自动验证功能里面会用到提示信息，这个部分也可以使用多语言的特性。例如原来的方式是把提示信息直接写在模型里面定义：

```
array('title','require','标题必须！',1),
```

如果使用了多语言功能的话（假设，我们在当前语言包里面定义了' lang_var'=>'标题必须！'），就可以这样定义模型的自动验证

```
array('title','require','{%lang_var}',1),
```

如果要在模板中输出语言变量不需要在控制器中赋值，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.lang_var}
```

可以输出当前选择的语言包里面定义的 `lang_var` 语言定义。

变量传入支持

语言包定义的时候支持传入变量，例如：

```
'FILE_FORMAT'    =>    '文件格式: {$format},文件大小：{$size}',
```

在模板中输出语言字符串的时候传入变量值即可：

```
{:L('FILE_FORMAT',array('format' => 'jpeg,png,gif,jpg','maximum' => '2MB'))}
```

数据分页

通常在数据查询后都会对数据集进行分页操作，ThinkPHP也提供了分页类来对数据分页提供支持。 下面是数据分页的两种示例。

利用Page类和limit方法分页

```
$User = M('User'); // 实例化User对象
$count      = $User->where('status=1')->count();// 查询满足要求的总记录数
$Page       = new \Think\Page($count,25);// 实例化分页类 传入总记录数和每页显示的记录数(25)
$show       = $Page->show();// 分页显示输出
// 进行分页数据查询 注意limit方法的参数要使用Page类的属性
$list = $User->where('status=1')->order('create_time')->limit($Page->firstRow.','.$Page->listRows)->
select();
$this->assign('list',$list);// 赋值数据集
$this->assign('page',$show);// 赋值分页输出
$this->display(); // 输出模板
```

分页类和page方法的实现分页

```
$User = M('User'); // 实例化User对象
// 进行分页数据查询 注意page方法的参数的前面部分是当前的页数使用 $_GET[p]获取
$list = $User->where('status=1')->order('create_time')->page($_GET['p'].',25')->select();
$this->assign('list',$list);// 赋值数据集
$count      = $User->where('status=1')->count();// 查询满足要求的总记录数
$Page       = new \Think\Page($count,25);// 实例化分页类 传入总记录数和每页显示的记录数
$show       = $Page->show();// 分页显示输出
$this->assign('page',$show);// 赋值分页输出
$this->display(); // 输出模板
```

带入查询条件

如果是POST方式查询，如何确保分页之后能够保持原先的查询条件呢，我们可以给分页类传入参数，方法是给分页类的parameter属性赋值

```
$count      = $User->where($map)->count();// 查询满足要求的总记录数
$Page       = new \Think\Page($count,25);// 实例化分页类 传入总记录数和每页显示的记录数
//分页跳转的时候保证查询条件
foreach($map as $key=>$val) {
    $Page->parameter[$key]   =   urlencode($val);
}
$show       = $Page->show();// 分页显示输出
```

分页样式定制

我们可以对输出的分页样式进行定制，分页类Page提供了一个setConfig方法来修改默认的一些设置。例如：

```
$Page->setConfig('header','个会员');
```

setConfig方法支持的属性包括：

属性	描述
header	头部描述信息，默认值 “共 %TOTAL_ROW% 条记录”
prev	上一页描述信息，默认值 “<<”
next	下一页描述信息，默认值 “>>”
first	第一页描述信息，默认值 “1...”
last	最后一页描述信息，默认值 “...%TOTAL_PAGE%”
theme	分页主题描述信息，包括了上面所有元素的组合 ，设置该属性可以改变分页的各个单元的显示位置，默认值是 "%FIRST% %UP_PAGE% %LINK_PAGE% %DOWN_PAGE% %END%"

其中，显示位置的对应的关系为：

位置	说明
%FIRST%	表示第一页的链接显示
%UP_PAGE%	表示上一页的链接显示
%LINK_PAGE%	表示分页的链接显示
%DOWN_PAGE%	表示下一页的链接显示
%END%	表示最后一页的链接显示

除了改变显示信息外，你还可以使用样式来定义分页的显示效果。 这些样式class包括：first（第一页）、prev（上一页）、next（下一页）、end（最后一页）、num（其他页的数字）、current（当前页）。

文件上传

上传表单

在ThinkPHP中使用上传功能无需进行特别处理。例如，下面是一个带有附件上传的表单提交：

```
<form action="__URL__/upload" enctype="multipart/form-data" method="post" >
<input type="text" name="name" />
<input type="file" name="photo" />
<input type="submit" value="提交" >
</form>
```

> 注意，要使用上传功能 你的表单需要设置 enctype="multipart/form-data"

多文件上传支持

如果需要使用多个文件上传，只需要修改表单，把

```
<input type='file'  name='photo'>
```

改为

```
<input type='file'  name='photo1'>
<input type='file'  name='photo2'>
<input type='file'  name='photo3'>
```

或者

```
<input type='file'  name='photo[]'>
<input type='file'  name='photo[]'>
<input type='file'  name='photo[]'>
```

两种方式的多附件上传系统的文件上传类都可以自动识别。

上传操作

ThinkPHP文件上传操作使用`Think\Upload`类，假设前面的表单提交到当前控制器的upload方法，我们来看下upload方法的实现代码：

```
public function upload(){
    $upload = new \Think\Upload();// 实例化上传类
    $upload->maxSize   =    3145728 ;// 设置附件上传大小
    $upload->exts      =    array('jpg', 'gif', 'png', 'jpeg');// 设置附件上传类型
    $upload->rootPath  =    './Uploads/'; // 设置附件上传根目录
    $upload->savePath  =    ''; // 设置附件上传（子）目录
    // 上传文件
    $info   =   $upload->upload();
    if(!$info) {// 上传错误提示错误信息
        $this->error($upload->getError());
    }else{// 上传成功
        $this->success('上传成功！');
    }
}
```

> 上传类对图片文件的上传安全做了支持，如果企图上传非法的图像文件，系统会提示 `非法图像文件`。
> 为了更好的使用上传功能，建议你的服务器开启 `finfo` 模块支持

上传参数

在上传操作之前，我们可以对上传的属性进行一些设置，Upload类支持的属性设置包括：

属性	描述
maxSize	文件上传的最大文件大小（以字节为单位），0为不限大小
rootPath	文件上传保存的根路径
savePath	文件上传的保存路径（相对于根路径）
saveName	上传文件的保存规则，支持数组和字符串方式定义
saveExt	上传文件的保存后缀，不设置的话使用原文件后缀
replace	存在同名文件是否是覆盖，默认为false
exts	允许上传的文件后缀（留空为不限制），使用数组或者逗号分隔的字符串设置，默认为空
mimes	允许上传的文件类型（留空为不限制），使用数组或者逗号分隔的字符串设置，默认为空
autoSub	自动使用子目录保存上传文件 默认为true
subName	子目录创建方式，采用数组或者字符串方式定义
hash	是否生成文件的hash编码 默认为true
callback	检测文件是否存在回调，如果存在返回文件信息数组

上面的属性可以通过两种方式传入：

实例化传入

我们可以在实例化的时候直接传入参数数组，例如：

```
$config = array(
    'maxSize'    =>    3145728,
    'rootPath'   =>    './Uploads/',
    'savePath'   =>    '',
    'saveName'   =>    array('uniqid',''),
    'exts'       =>    array('jpg', 'gif', 'png', 'jpeg'),
    'autoSub'    =>    true,
    'subName'    =>    array('date','Ymd'),
);
$upload = new \Think\Upload($config);// 实例化上传类
```

> 关于saveName和subName的使用后面我们会有详细的描述。

动态赋值

支持在实例化后动态赋值上传参数，例如：

```
$upload = new \Think\Upload();// 实例化上传类
$upload->maxSize = 3145728;
$upload->rootPath = './Uploads/';
$upload->savePath = '';
$upload->saveName = array('uniqid','');
$upload->exts     = array('jpg', 'gif', 'png', 'jpeg');
$upload->autoSub  = true;
$upload->subName  = array('date','Ymd');
```

上面的设置和实例化传入的效果是一致的。

上传文件信息

设置好上传的参数后，就可以调用 Think\Upload 类的upload方法进行附件上传，如果失败，返回false，并且用getError方法获取错误提示信息；如果上传成功，就返回成功上传的文件信息数组。

```
$upload = new \Think\Upload();// 实例化上传类
$upload->maxSize   =    3145728 ;// 设置附件上传大小
$upload->exts      =    array('jpg', 'gif', 'png', 'jpeg');// 设置附件上传类型
$upload->rootPath  =    './Uploads/'; // 设置附件上传根目录
$upload->savePath  =    ''; // 设置附件上传（子）目录
// 上传文件
$info   =   $upload->upload();
if(!$info) {// 上传错误提示错误信息
    $this->error($upload->getError());
}else{// 上传成功 获取上传文件信息
    foreach($info as $file){
        echo $file['savepath'].$file['savename'];
    }
}
```

每个文件信息又是一个记录了下面信息的数组，包括：

属性	描述
key	附件上传的表单名称
savepath	上传文件的保存路径
name	上传文件的原始名称
savename	上传文件的保存名称
size	上传文件的大小
type	上传文件的MIME类型
ext	上传文件的后缀类型
md5	上传文件的md5哈希验证字符串 仅当hash设置开启后有效
sha1	上传文件的sha1哈希验证字符串 仅当hash设置开启后有效

文件上传成功后，就可以使用这些文件信息来进行其他的数据操作，例如保存到当前数据表或者单独的附件数据表。

例如，下面表示把上传信息保存到数据表的字段：

```
$model = M('Photo');
// 取得成功上传的文件信息
$info = $upload->upload();
// 保存当前数据对象
$data['photo'] = $info['photo']['savename'];
$data['create_time'] = NOW_TIME;
$model->add($data);
```

单文件上传

upload方法支持多文件上传，有时候，我们只需要上传一个文件，就可以使用Upload类提供的uploadOne方法上传单个文件，例如：

```
public function upload(){
    $upload = new \Think\Upload();// 实例化上传类
    $upload->maxSize   =    3145728 ;// 设置附件上传大小
    $upload->exts      =    array('jpg', 'gif', 'png', 'jpeg');// 设置附件上传类型
    $upload->rootPath  =    './Uploads/'; // 设置附件上传根目录
    // 上传单个文件
    $info   =   $upload->uploadOne($_FILES['photo1']);
    if(!$info) {// 上传错误提示错误信息
        $this->error($upload->getError());
    }else{// 上传成功 获取上传文件信息
         echo $info['savepath'].$info['savename'];
    }
}
```

> uploadOne方法上传成功后返回的文件信息和upload方法的区别是只有单个文件信息的一维数组。

上传文件的命名规则

上传文件的命名规则（saveName）用于确保文件不会产生冲突或者覆盖的情况。命名规则的定义可以根据你的业务逻辑来调整，不是固定的。例如，如果你采用时间戳的方式来定义命名规范，那么在同时上传多个文件的时候可能产生冲突（因为同一秒内可以上传多个文件），因此你需要根据你的业务需求来设置合适的上传命名规则。这里顺便来说下saveName参数的具体用法。

采用函数方式

如果传入的字符串是一个函数名，那么表示采用函数动态生成上传文件名（不包括文件后缀），例如：

```
// 采用时间戳命名
$upload->saveName = 'time';
// 采用GUID序列命名
$upload->saveName = 'com_create_guid';
```

也可以采用用户自定义函数

```
// 采用自定义函数命名
$upload->saveName = 'myfun';
```

> 默认的命名规则设置是采用uniqid函数生成一个唯一的字符串序列。

saveName的值支持数组和字符串两种方式，如果是只有一个参数或者没有参数的函数，直接使用字符串设置即可，如果需要传入额外的参数，可以使用数组方式，例如：

```
// 采用date函数生成命名规则 传入Y-m-d参数
$upload->saveName = array('date','Y-m-d');
// 如果有多个参数需要传入的话 可以使用数组
$upload->saveName = array('myFun',array('__FILE__','val1','val2'));
```

如果需要使用上传的原始文件名，可以采用FILE传入，所以上面的定义规则，最终的结果是 myFun('上传文件名','val1','val2') 执行的结果。

直接设置上传文件名

如果传入的参数不是一个函数名，那么就会直接当做是上传文件名，例如：

```
$upload->saveName = time().'_'.mt_rand();
```

表示上传的文件命名采用时间戳加一个随机数的组合字符串方式。

当然，如果觉得有必要，你还可以固定设置一个上传文件的命名规则，用于固定保存某个上传文件。

```
$upload->saveName = 'ThinkPHP';
```

保持上传文件名不变

如果你想保持上传的文件名不变，那么只需要设置命名规范为空即可，例如：

```
$upload->saveName = '';
```

> 一般来说不建议保持不变，因为会导致相同的文件名上传后被覆盖的情况。

子目录保存

saveName只是用于设置文件的保存规则，不涉及到目录，如果希望对上传的文件分子目录保存，可以设置 autoSub 和 subName 参数来完成，例如：

```
// 开启子目录保存 并以日期（格式为Ymd）为子目录
$upload->autoSub = true;
$upload->subName = array('date','Ymd');
```

可以使用自定义函数来保存，例如：

```
// 开启子目录保存 并调用自定义函数get_user_id生成子目录
$upload->autoSub = true;
$upload->subName = 'get_user_id';
```

和saveName参数一样，subName的定义可以采用数组和字符串的方式。

> 注意：如果get_user_id函数未定义的话，会直接以get_user_id字符串作为子目录的名称保存。

子目录保存和文件命名规则可以结合使用。

上传驱动

上传类可以支持不同的环境，通过相应的上传驱动来解决，默认情况下使用本地（Local）上传驱动，当然，你还可以设置当前默认的上传驱动类型，例如：

```
'FILE_UPLOAD_TYPE'    =>    'Ftp',
'UPLOAD_TYPE_CONFIG'  =>    array(
        'host'     => '192.168.1.200', //服务器
        'port'     => 21, //端口
        'timeout'  => 90, //超时时间
        'username' => 'ftp_user', //用户名
        'password' => 'ftp_pwd', //密码 ),
```

表示当前使用Ftp作为上传类的驱动，上传的文件会通过FTP传到指定的远程服务器。

也可以在实例化上传类的时候指定，例如：

```
$config = array(
    'maxSize'    =>    3145728,
    'rootPath'   =>    './Uploads/',
    'savePath'   =>    '',
    'saveName'   =>    array('uniqid',''),
    'exts'       =>    array('jpg', 'gif', 'png', 'jpeg'),
    'autoSub'    =>    true,
    'subName'    =>    array('date','Ymd'),
);
$ftpConfig     =    array(
        'host'     => '192.168.1.200', //服务器
        'port'     => 21, //端口
        'timeout'  => 90, //超时时间
        'username' => 'ftp_user', //用户名
        'password' => 'ftp_pwd', //密码 );

$upload = new \Think\Upload($config,'Ftp',$ftpConfig);// 实例化上传类
```

> 目前已经支持的上传驱动包括Local、Ftp、Sae、Bcs、七牛和又拍云等。

验证码

Think\Verify类可以支持验证码的生成和验证功能。

生成验证码

下面是最简单的方式生成验证码：

```
$Verify = new \Think\Verify();
$Verify->entry();
```

上面的代码会生成默认的验证码图片并输出，显示如下：

生成的验证码信息会保存到session中，包含的数据有：

```
array('verify_code'=>'当前验证码的值','verify_time'=>'验证码生成的时间戳')
```

如果你需要在一个页面中生成多个验证码的话，entry方法需要传入可标识的信息，例如： 验证码1：

```
// 验证码1
$Verify = new \Think\Verify();
$Verify->entry(1);
```

验证码2：

```
// 验证码2
$Verify = new \Think\Verify();
$Verify->entry(2);
```

验证码参数

可以对生成的验证码设置相关的参数，以达到不同的显示效果。 这些参数包括：

参数	描述
expire	验证码的有效期（秒）
useImgBg	是否使用背景图片 默认为false
fontSize	验证码字体大小（像素） 默认为25
useCurve	是否使用混淆曲线 默认为true
useNoise	是否添加杂点 默认为true

参数	描述
imageW	验证码宽度 设置为0为自动计算
imageH	验证码高度 设置为0为自动计算
length	验证码位数
fontttf	指定验证码字体 默认为随机获取
useZh	是否使用中文验证码
bg	验证码背景颜色 rgb数组设置，例如 array(243, 251, 254)
seKey	验证码的加密密钥
codeSet	验证码字符集合
zhSet	验证码字符集合（中文）

参数设置使用两种方式。

实例化传入参数：

```
$config =    array(
    'fontSize'    =>    30,    // 验证码字体大小
    'length'      =>    3,     // 验证码位数
    'useNoise'    =>    false, // 关闭验证码杂点
);
$Verify =     new \Think\Verify($config);
$Verify->entry();
```

或者采用动态设置的方式，如：

```
$Verify =     new \Think\Verify();
$Verify->fontSize = 30;
$Verify->length   = 3;
$Verify->useNoise = false;
$Verify->entry();
```

生成的验证码如图所示：

验证码字体

默认情况下，验证码的字体是随机使用 `ThinkPHP/Library/Think/Verify/ttfs/` 目录下面的字体文件，我们可以指定验证码的字体，例如：

```
$Verify =     new \Think\Verify();
// 验证码字体使用 ThinkPHP/Library/Think/Verify/ttfs/5.ttf
$Verify->fontttf = '5.ttf';
$Verify->entry();
```

背景图片

支持验证码背景图片功能，可以如下设置：

```
$Verify =     new \Think\Verify();
// 开启验证码背景图片功能 随机使用 ThinkPHP/Library/Think/Verify/bgs 目录下面的图片
$Verify->useImgBg = true;
$Verify->entry();
```

效果如图所示：

中文验证码

如果要使用中文验证码，可以设置：

```
$Verify =     new \Think\Verify();
// 验证码字体使用 ThinkPHP/Library/Think/Verify/ttfs/5.ttf
$Verify->useZh = true;
$Verify->entry();
```

显示效果如图：

> 如果无法正常显示，请确认你的 ThinkPHP/Library/Think/Verify/zhttfs/ 目录下面存在中文字体文件。

指定验证码字符

3.2.1版本以上，我们可以指定验证码的字符，通过重新设置codeSet参数即可，例如：

```
$Verify =     new \Think\Verify();
// 设置验证码字符为纯数字
$Verify->codeSet = '0123456789';
$Verify->entry();
```

如果是中文验证码，可以使用zhSet参数设置，例如：

```
$Verify =     new \Think\Verify();
$Verify->useZh = true;
// 设置验证码字符
$Verify->zhSet = '们以我到他会作时要动国产的一是工就年阶义发成部民可出能方进在了不和有大这';
$Verify->entry();
```

验证码检测

可以用Think\Verify类的 check 方法检测验证码的输入是否正确，例如，下面是封装的一个验证码检测的函数：

```
// 检测输入的验证码是否正确，$code为用户输入的验证码字符串
function check_verify($code, $id = ''){
    $verify = new \Think\Verify();
    return $verify->check($code, $id);
}
```

图像处理

使用Think\Image类进行图像处理功能，支持Gd库和Imagick库，包括对GIf图像处理的支持。

实例化类库

```
$image = new \Think\Image();
```

默认使用GD库进行图像操作，如果需要使用Imagick库操作的话，需要改成：

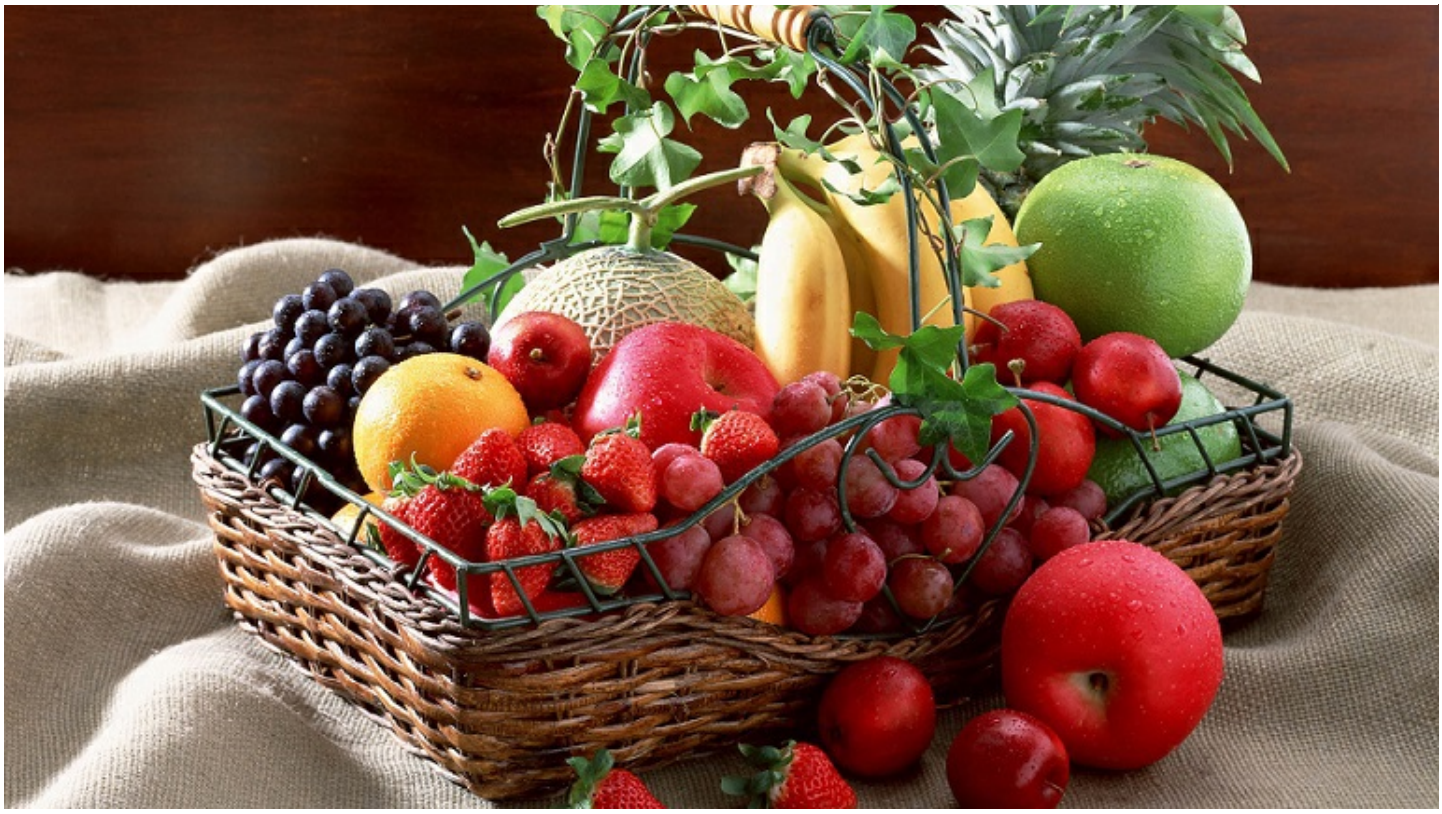
```
$image = new \Think\Image(\Think\Image::IMAGE_IMAGICK);
// 或者采用
$image = new \Think\Image('Imagick');
```

图像操作

下面来看下基础的图像操作功能的使用方法。

打开图像文件

假设当前入口文件目录下面有一个1.jpg文件，如图所示：



使用open方法打开图像文件进行相关操作：

```
$image = new \Think\Image();
$image->open('./1.jpg');
```

也可以简化成下面的方式：

```
$image = new \Think\Image(\Think\Image::IMAGE_GD,'./1.jpg'); // GD库
// 或者
$image = new \Think\Image(\Think\Image::IMAGE_IMAGICK,'./1.jpg');  // imagick库
```

获取图像信息

可以获取打开图片的信息，包括图像大小、类型等，例如：

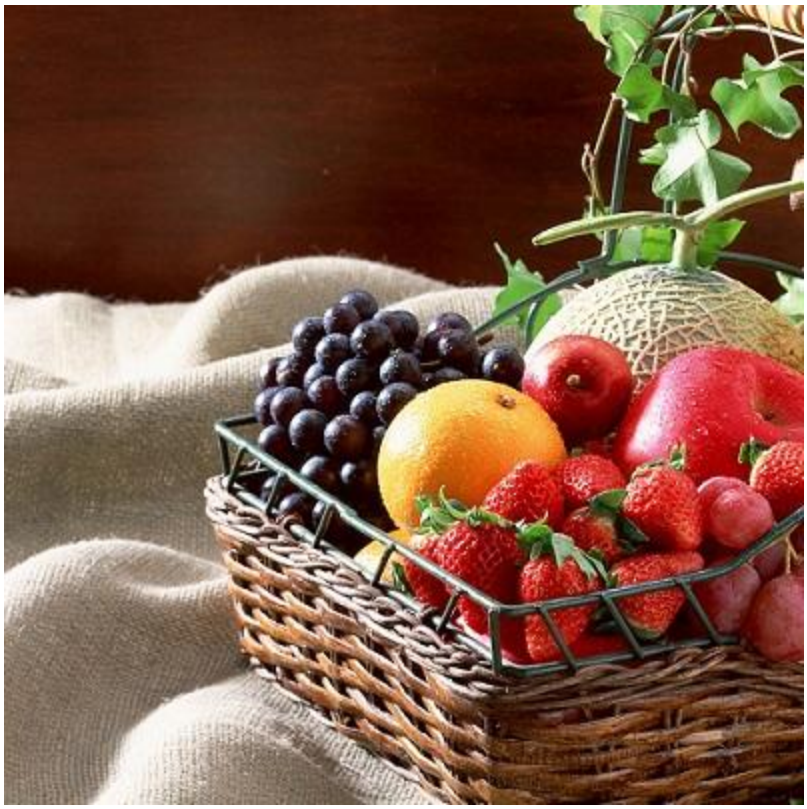
```
$image = new \Think\Image();
$image->open('./1.jpg');
$width = $image->width(); // 返回图片的宽度
$height = $image->height(); // 返回图片的高度
$type = $image->type(); // 返回图片的类型
$mime = $image->mime(); // 返回图片的mime类型
$size = $image->size(); // 返回图片的尺寸数组 0 图片宽度 1 图片高度
```

裁剪图片

使用crop和save方法完成裁剪图片功能。

```
$image = new \Think\Image();
$image->open('./1.jpg');
//将图片裁剪为400x400并保存为corp.jpg
$image->crop(400, 400)->save('./crop.jpg');
```

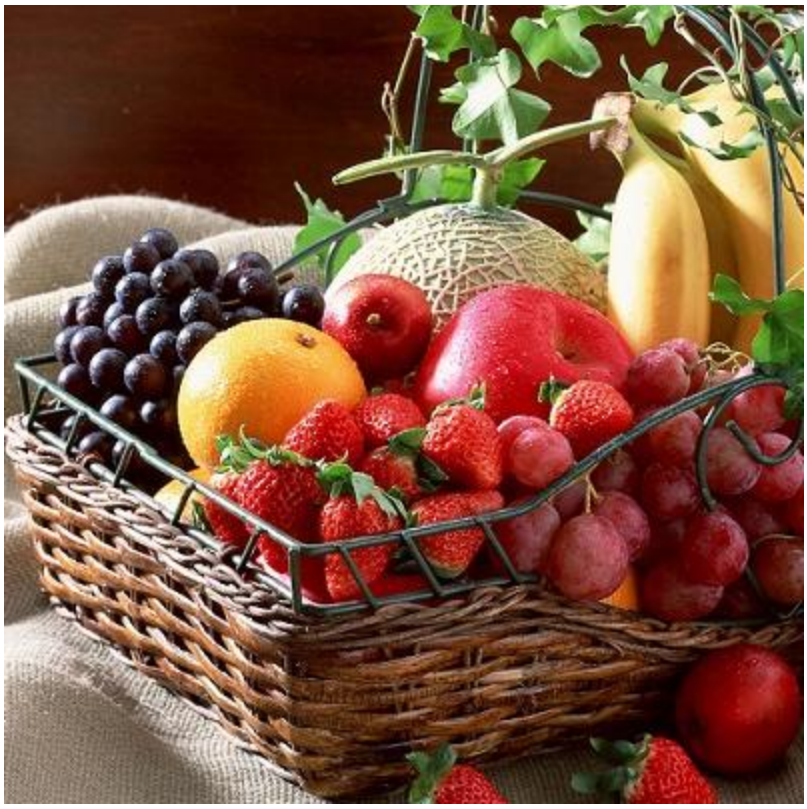
生成的图片如图：



支持从某个坐标开始裁剪，例如下面从（100，30）开始裁剪：

```
$image = new \Think\Image();
$image->open('./1.jpg');
//将图片裁剪为400x400并保存为corp.jpg
$image->crop(400, 400,100,30)->save('./crop.jpg');
```

生成的图片如图：



生成缩略图

使用thumb方法生成缩略图

```
$image = new \Think\Image();
$image->open('./1.jpg');
// 按照原图的比例生成一个最大为150*150的缩略图并保存为thumb.jpg
$image->thumb(150, 150)->save('./thumb.jpg');
```

生成的缩略图如图所示：

我们看到实际生成的缩略图并不是150*150，因为默认采用原图等比例缩放的方式生成缩略图，最大宽度是150。

可以支持其他类型的缩略图生成，设置包括Think\Image的下列常量或者对应的数字：

```
IMAGE_THUMB_SCALE    =  1 ; //等比例缩放类型
IMAGE_THUMB_FILLED   =  2 ; //缩放后填充类型
IMAGE_THUMB_CENTER   =  3 ; //居中裁剪类型
IMAGE_THUMB_NORTHWEST =  4 ; //左上角裁剪类型
IMAGE_THUMB_SOUTHEAST =  5 ; //右下角裁剪类型
IMAGE_THUMB_FIXED    =  6 ; //固定尺寸缩放类型
```

例如：

居中裁剪

```
$image = new \Think\Image();
$image->open('./1.jpg');
// 生成一个居中裁剪为150*150的缩略图并保存为thumb.jpg
$image->thumb(150, 150,\Think\Image::IMAGE_THUMB_CENTER)->save('./thumb.jpg');
```

居中裁剪后生成的缩略图效果如图：

左上角剪裁

```
$image = new \Think\Image();
$image->open('./1.jpg');
// 生成一个左上角裁剪为150*150的缩略图并保存为thumb.jpg
$image->thumb(150, 150,\Think\Image::IMAGE_THUMB_NORTHWEST)->save('./thumb.jpg');
```

左上角裁剪后生成的缩略图效果如图：

缩放填充

```
$image = new \Think\Image();
$image->open('./1.jpg');
// 生成一个缩放后填充大小150*150的缩略图并保存为thumb.jpg
$image->thumb(150, 150,\Think\Image::IMAGE_THUMB_FILLED)->save('./thumb.jpg');
```

缩放填充后生成的缩略图效果如图：

固定大小

```
$image = new \Think\Image();
$image->open('./1.jpg');
// 生成一个固定大小为150*150的缩略图并保存为thumb.jpg
$image->thumb(150, 150,\Think\Image::IMAGE_THUMB_FIXED)->save('./thumb.jpg');
```

采用固定大小的缩略图可能会有所变形，生成的缩略图效果如图：

添加图片水印

```
$image = new \Think\Image();
$image->open('./1.jpg');
//将图片裁剪为440x440并保存为corp.jpg
$image->crop(440, 440)->save('./crop.jpg');
// 给裁剪后的图片添加图片水印（水印文件位于./logo.png），位置为右下角，保存为water.gif
$image->water('./logo.png')->save("water.gif");
// 给原图添加水印并保存为water_o.gif（需要重新打开原图）
$image->open('./1.jpg')->water('./logo.png')->save("water_o.gif");
```

water方法的第二个参数表示水印的位置，可以传入下列Think\Image类的常量或者对应的数字：

```
IMAGE_WATER_NORTHWEST =  1 ; //左上角水印
IMAGE_WATER_NORTH     =  2 ; //上居中水印
IMAGE_WATER_NORTHEAST =  3 ; //右上角水印
IMAGE_WATER_WEST      =  4 ; //左居中水印
IMAGE_WATER_CENTER    =  5 ; //居中水印
IMAGE_WATER_EAST      =  6 ; //右居中水印
IMAGE_WATER_SOUTHWEST =  7 ; //左下角水印
IMAGE_WATER_SOUTH     =  8 ; //下居中水印
IMAGE_WATER_SOUTHEAST =  9 ; //右下角水印
```

例如：

```
$image = new \Think\Image();
// 在图片左上角添加水印（水印文件位于./logo.png） 并保存为water.jpg
$image->open('./1.jpg')->water('./logo.png',\Think\Image::IMAGE_WATER_NORTHWEST)->save("water.jpg");
```

生成的图片效果如下：

还可以支持水印图片的透明度（0~100，默认值是80），例如：

```
$image = new \Think\Image();
// 在图片左上角添加水印（水印文件位于./logo.png） 水印图片的透明度为50 并保存为water.jpg
$image->open('./1.jpg')->water('./logo.png',\Think\Image::IMAGE_WATER_NORTHWEST,50)->save("water.jpg");
```

生成的效果如下：

也可以支持给图片添加文字水印（假设在入口文件的同级目录下存在1.ttf字体文件），例如：

```
$image = new \Think\Image();
// 在图片右下角添加水印文字 ThinkPHP 并保存为new.jpg
$image->open('./1.jpg')->text('ThinkPHP','./1.ttf',20,'#000000',\Think\Image::IMAGE_WATER_SOUTHEAST)->save("new.jpg");
```

生成的图片效果：

RESTFul

REST介绍

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

1、网络上的所有事物都被抽象为资源（resource）；

2、每个资源对应一个唯一的资源标识（resource identifier）；

3、通过通用的连接器接口（generic connector interface）对资源进行操作；

4、对资源的各种操作不会改变资源标识；

5、所有的操作都是无状态的（stateless）。

需要注意的是，REST是设计风格而不是标准。REST通常基于使用HTTP，URI，和XML以及HTML这些现有的广泛流行的协议和标准。

传统的请求模式和REST模式的请求模式区别：

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

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

RESTFul支持

3.2的RESTFul支持更为灵活，你只需要把控制器继承Think\Controller\RestController即可。 继承RestController控制器后你的访问控制器就可以支持下面的一些功能：

> - 支持资源类型自动检测；
> - 支持请求类型自动检测；
> - RESTFul方法支持；
> - 可以设置允许的请求类型列表；
> - 可以设置允许请求和输出的资源类型；

- 可以设置默认请求类型和默认资源类型；

例如：

```
namespace Home\Controller;
use Think\Controller\RestController;
class BlogController extends RestController{
}
```

REST参数

继承了RestController后，你可以在你的控制器里面设置rest相关的属性参数，包括：allowMethod，defaultMethod，allowType，defaultType以及allowOutputType。

属性名	说明	默认值
allowMethod	REST允许的请求类型列表	array('get','post','put','delete')
defaultMethod	REST默认请求类型	get
allowType	REST允许请求的资源类型列表	array('html','xml','json','rss')
defaultType	REST默认的资源类型	html
allowOutputType	REST允许输出的资源类型列表	array('xml' => 'application/xml', 'json' => 'application/json','html' => 'text/html',)

REST方法

RESTFul方法的操作方法定义主要区别在于，需要对请求类型和资源类型进行判断，大多数情况下，通过路由定义可以把操作方法绑定到某个请求类型和资源类型。如果你没有定义路由的话，需要自己在操作方法里面添加判断代码，示例：

```
namespace Home\Controller;
use Think\Controller\RestController;
Class InfoController extends RestController {
   Public function rest() {
     switch ($this->_method){
      case 'get': // get请求处理代码
           if ($this->_type == 'html'){
           }elseif($this->_type == 'xml'){
           }
           break;
      case 'put': // put请求处理代码
           break;
      case 'post': // post请求处理代码
           break;
     }
   }
}
```

> 在Rest操作方法中，可以使用$this->_type获取当前访问的资源类型，用$this->_method获取当前的请求类型。

REST控制器类还提供了response方法用于REST输出： 用法如下：

```
$this->response($data,'json');
```

Response方法会自动对data数据进行输出类型编码，目前支持的包括 xml/json/html 。

除了普通方式定义Restful操作方法外，系统还支持另外一种自动调用方式，就是根据当前请求类型和资源类型自动调用相关操作方法。系统的自动调用规则是：

定义规范	说明
操作名*提交类型*资源后缀	标准的Restful方法定义，例如 read_get_pdf
操作名_资源后缀	当前提交类型和defaultMethod属性相同的时候，例如read_pdf
操作名_提交类型	当前资源后缀和defaultType属性相同的时候，例如read_post

要使用这种方式的前提就是不能为当前操作定义方法，这样在空操作的检查之前系统会首先按照上面的定义规范顺序检查是否存在方法定义，如果检测到相关的restful方法则不再检查后面的方法规范，例如我们定义了InfoController如下：

```
namespace Home\Controller;
use Think\Controller\RestController;
Class InfoController extends RestController {
    protected $allowMethod    = array('get','post','put'); // REST允许的请求类型列表
    protected $allowType      = array('html','xml','json'); // REST允许请求的资源类型列表

    Public function read_get_html(){
        // 输出id为1的Info的html页面
    }

    Public function read_get_xml(){
        // 输出id为1的Info的XML数据
    }
    Public function read_xml(){
        // 输出id为1的Info的XML数据
    }
    Public function read_json(){
        // 输出id为1的Info的json数据
    }
}
```

如果我们访问的URL是：

```
http://www.domain.com/Info/read/id/1.xml
```

假设我们没有定义路由，这样访问的是Info控制器的read操作，那么上面的请求会调用InfoController类的 `read_get_xml` 方法，而不是 `read_xml` 方法，但是如果访问的URL是：

```
http://www.domain.com/Info/read/id/1.json
```

那么则会调用read_json方法。

如果我们访问的URL是

```
http://www.domain.com/Info/read/id/1.rss
```

由于我们不允许rss资源类型的访问，所以，调用的方法其实是read_html方法。

REST路由

我们可以借助3.2的路由参数功能，来解决REST的路由定义问题。 例如，

```
'blog/:id'=>array('blog/read','status=1',array('ext'=>'xml','method'=>'get')),
```

上面的路由定义，把blog/5路由到了blog/read/id/5 并且，约束了后缀是xml 请求类型是get。 我们还可

以定义其他的路由参数，例如：

```
'blog/:id'=>array('blog/update','',array('ext'=>'xml','method'=>'put')),
```

为了确保定义不冲突，REST路由定义我们通常改成下面的定义方式：

```
array('blog/:id','blog/read','status=1',array('ext'=>'xml','method'=>'get')),
array('blog/:id','blog/update','',array('ext'=>'xml','method'=>'put')),
```

> 这样就可以给相同的路由规则定义不同的参数支持。定义了REST路由后，你的rest方法定义就不受任何约束，当然，如果路由定义的操作方法不存在的时候，系统默认的rest方法规范仍然会有效。

RPC

RPC（Remote Procedure Call Protocol）远程过程调用协议，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。RPC协议假定某些传输协议的存在，如TCP或UDP，为通信程序之间携带信息数据。在OSI网络通信模型中，RPC跨越了传输层和应用层。RPC使得开发包括网络分布式多程序在内的应用程序更加容易。

RPC采用客户机/服务器模式。请求程序就是一个客户机，而服务提供程序就是一个服务器。首先，客户机调用进程发送一个有进程参数的调用信息到服务进程，然后等待应答信息。在服务器端，进程保持睡眠状态直到调用信息的到达为止。当一个调用信息到达，服务器获得进程参数，计算结果，发送答复信息，然后等待下一个调用信息，最后，客户端调用进程接收答复信息，获得进程结果，然后调用执行继续进行。

ThinkPHP支持广泛的RPC协议，包括PHPRPC、HPRose、JsonRPC以及Yar。

PHPRPC支持

PHPRPC 是一个轻型的、安全的、跨网际的、跨语言的、跨平台的、跨环境的、跨域的、支持复杂对象传输的、支持引用参数传递的、支持内容输出重定向的、支持分级错误处理的、支持会话的、面向服务的高性能远程过程调用协议。目前该协议的最新版本为 3.0。详细的资料可以参考phprpc官网（http://www.phprpc.org/zh_CN/）

ThinkPHP提供了对PHPRpc的服务端和客户端调用的支持（客户端是跨平台跨语言的，可以支持任何语言的调用）。

服务器端的实现非常简单，你只需要把控制器继承Think\Controller\RpcController类即可。

例如：

```
namespace Home\Controller;
use Think\Controller\RpcController;
class ServerController extends RpcController{
    public function test1(){
        return 'test1';
    }
    public function test2(){
        return 'test2';
    }
    private function test3(){
        return 'test3';
    }
    protected function test4(){
        return 'test3';
    }
}
```

这样，ServerController控制器就变成了一个PHPRpc服务端，请求地址为：

```
http://serverName/index.php/Home/Server
```

rpc客户端可以调用 `test1` 和 `test2` 方法。方法的返回值可以支持数组等PHPRpc支持的格式。

> 注意：如果设置了不同的URL模式的话，服务器端请求地址需要相应调整。

可以使用allowMethodList属性设置允许访问的方法列表，例如：

```
namespace Home\Controller;
use Think\Controller\RpcController;
class ServerController extends RpcController{
    protected $allowMethodList =    array('test1','test2');
    public function test1(){
        return 'test1';
    }
    public function test2(){
        return 'test2';
    }
    public function test3(){
        return 'test3';
    }
}
```

上面的设置表示只允许 `test1` 和 `test2` 方法被rpc客户端调用。

> 你可以采用多个控制器进行不同的PHPRpc Server端用于不同的需要。

如果要在ThinkPHP里面进行客户端调用，可以使用下面的代码：

```
namespace Home\Controller;
use Think\Controller;
class IndexController extends Controller {
    public function index(){
        Vendor('phpRPC.phprpc_client');
        $client = new \PHPRPC_Client('http://serverName/index.php/Home/Server');
        // 或者采用
        //$client = new \PHPRPC_Client();
        //$client->useService('http://serverName/index.php/Home/Server');
        $result = $client->test1();
    }
}
```

其中test1就是服务器端定义的方法。其他使用和PHP的方法调用一致。

Hprose支持

Hprose (High Performance Remote Object Service Engine) 是一个MIT开源许可的新型轻量级跨语言跨平台的面向对象的高性能远程动态通讯中间件。它支持众多语言，例如nodeJs, C++, .NET, Java, Delphi, Objective-C, ActionScript, JavaScript, ASP, PHP, Python, Ruby, Perl 等语言，通过 Hprose 可以在这些语言之间实现方便且高效的互通。

你可以认为它是 PHPRPC 的商业版本，但是它跟 PHPRPC 完全不同，hprose 协议是全新设计的，比 PHPRPC 更加高效，实现也完全是全部从头开始的，比 PHPRPC 更加易用。更多信息可以参考（http://www.hprose.com/ ）

ThinkPHP同样也提供了对Hprose的服务端和客户端调用的支持。

服务器端的使用和PHPRPC的区别只是把控制器继承Think\Controller\HproseController类即可，其他用法基本一致，例如：

```
namespace Home\Controller;
use Think\Controller\HproseController;
class ServerController extends HproseController{
    public function test1(){
        return 'test1';
    }
    public function test2(){
        return 'test2';
    }
}
```

我们可以进行一些hprose服务器端的参数设置，包括debug、crossDomain、P3P和get，设置方法如下：

```
namespace Home\Controller;
use Think\Controller\HproseController;
class ServerController extends HproseController{
    protected $crossDomain =    true;
    protected $P3P         =    true;
    protected $get         =    true;
    protected $debug       =    true;

    public function test1(){
        return 'test1';
    }
    public function test2(){
        return 'test2';
    }
}
```

采用ThinkPHP的Hprose的客户端调用示例如下：

```
namespace Home\Controller;
use Think\Controller;
class IndexController extends Controller {
    public function index(){
        vendor('Hprose.HproseHttpClient');
        $client = new \HproseHttpClient('http://serverName/index.php/Home/Server');
        // 或者采用
        //$client = new \HproseHttpClient();
        //$client->useService('http://serverName/index.php/Home/Server');
        $result = $client->test1();
    }
}
```

JsonRPC支持

json-rpc是基于json的跨语言远程调用协议，比xml-rpc、webservice等基于文本的协议传输数据格小；相对hessian、java-rpc等二进制协议便于调试、实现、扩展，是非常优秀的一种远程调用协议。

ThinkPHP3.2提供了对JsonRPC的服务器端和客户端调用支持，服务器端实现示例：

```
namespace Home\Controller;
use Think\Controller\JsonRpcController;
class ServerController extends JsonRpcController {
    public function index(){
        return 'Hello, JsonRPC!';
    }
    // 支持参数传入
    public function test($name=''){
        return "Hello, {$name}!";
    }
}
```

所有的public方法都可以用于远程调用，客户端调用方式如下：

```
namespace Home\Controller;
use Think\Controller;
class IndexController extends Controller {
    public function index(){
        vendor('jsonRPC.jsonRPCClient');
        $client = new \jsonRPCClient('http://serverName/index.php/Home/Server');
        $result = $client->index();
        var_dump($result); // 结果：Hello, JsonRPC!
        $result = $client->test('ThinkPHP');
        var_dump($result); // 结果：Hello, ThinkPHP!
    }
}
```

Yar支持

Yar (yet another RPC framework) 是一个PHP扩展的RPC框架, 和现有的RPC框架(xml-rpc, soap)不同, 这是一个轻量级的框架, 支持多种打包协议(msgpack, json, php), 并且最重要的一个特点是, 它是可并行化的。

要使用Yar支持首先需要安装Yar扩展，扩展下载地址： http://pecl.php.net/package/yar

Yar说明文档： http://hk2.php.net/manual/zh/book.yar.php

安装好扩展后，使用ThinkPHP就可以开发服务器端，示例如下：

```
namespace Home\Controller;
use Think\Controller\YarController;
class ServerController extends YarController {
    public function index(){
        return 'Hello, Yar RPC!';
    }
    public function hello($name=''){
        return 'Hello, {$name}!';
    }
}
```

> Yar除了并行，还有一个亮点，通过GET方式可以查看到接口列表及注释。

客户端调用示例如下：

```
namespace Home\Controller;
use Think\Controller;
class IndexController extends Controller {
    public function index(){
        $client = new \Yar_client('http://serverName/index.php/Home/Server');
        $result = $client->index();
        var_dump($result); // 结果：Hello, Yar!
    }
}
```

SAE

SAE介绍

Sina App Engine（简称SAE）是新浪研发中心开发的国内首个公有云计算平台，是新浪云计算战略的核心组成部分，作为一个简单高效的分布式Web服务开发、运行平台越来越受开发者青睐。

SAE环境和普通环境有所不同，它是一个分布式服务器集群，能让你的程序同时运行在多台服务器中。并提供了很多高效的分布式服务。SAE为了提升性能和安全，禁止了本地IO写操作，使用MemcacheX、Storage等存储型服务代替传统IO操作，效率比传统IO读写操作高，有效解决因IO瓶颈导致程序性能低下的问题。

> 正是因为SAE和普通环境的不同，使得普通程序不能直接放在SAE上，需要经过移植才能放在SAE上运行。也使得很多能在SAE上运行的程序不能在普通环境下运行。

ThinkPHP3.2核心内置了对SAE平台的支持（采用了应用模式的方式），具有自己的独创特性，能够最大程度的使用ThinkPHP的标准特性，让开发人员感受不到SAE和普通环境的差别。甚至可以不学习任何SAE知识，只要会ThinkPHP开发，就能将你的程序运行在SAE上。

SAE版ThinkPHP具有以下特性：

- 横跨性：能让同样的代码既能在SAE环境下运行，也能在普通环境下运行。解决了使用SAE不能在本地调试代码的问题。
- 平滑性：我们还是按照以前一样使用ThinkPHP，但是您已经不知不觉的使用了SAE服务，不用特意学习SAE服务，降低学习成本。比如你不用特意的去学习KVDB服务，你在SAE环境下使用ThinkPHP的F函数就已经使用了KVDB的服务。
- 完整性：SAE开发下面功能没有任何删减，支持ThinkPHP标准模式的所有功能。

> 大多SAE移植程序都是使用Wrappers实现，SAE版ThinkPHP没有使用Wrappers，使用SAE的原始服务接口，运行效率比用Wrappers更高。

3.2版本中你无需考虑SAE平台的部署，在本地采用标准模式开发完成后，直接部署到SAE平台后，系统会自动转换为SAE模式运行。

> 在本地开发完成后，上传到SAE平台需要做一些初始化工作，例如初始化Mysql，Memcache，KVDB服务。 SAE平台不支持IO写操作，所以你不能在SAE上首次运行入口文件生成项目目录。你可以在本地运行入口文件，本地生成好项目目录后再提交到SAE上。
>
> 注意：如果要部署到SAE平台的话，你的应用代码不要直接进行文件读写操作，而采用ThinkPHP封装的方法或者函数进行操作即可。

SAE配置

SAE引擎运行时拥有SAE自己的惯例配置和专有配置，因此配置文件加载顺序为：

```
惯例配置->项目配置->SAE惯例配置->SAE专有配置
```

> SAE惯例配置和SAE专有配置中的配置项将会覆盖项目配置。

SAE惯例配置：位于系统目录的 /Mode/Sae/convention.php，其中定义了程序在SAE上运行时固定的数据库连接配置项。

SAE专有配置：位于应用的公共Conf目录下，文件名为config_sae.php，大家可以将针对SAE的配置写到其中。

> 注：SAE惯例配置和SAE专有配置是针对SAE环境的独有配置，在本地运行时将不会加载。

数据库

开发者不需要在应用配置文件(config.php)中定义和SAE相关的数据库配置项，只需要定义本地调试时连接的数据库即可。代码提交到SAE时无需修改任何配置项也能运行，因为SAE惯例配置会自动覆盖你的项目配

置文件中的数据库配置。

> 代码在SAE上运行时会进行分布式数据库连接，并读写分离。

缓存

在SAE开发过程中，你仍然可以使用ThinkPHP内置的缓存方法进行处理。下面是SAE引擎使用不同的缓存方法在本地和SAE平台下的区别（注意这个区别SAE引擎会自动判断处理）：

缓存方法	本地运行	SAE平台
S缓存	默认使用File方式实现	固定使用Memcache实现
F缓存	使用File实现	使用KVDB实现
静态缓存	生成静态Html文件	静态文件存入KVDB中
SQL队列	支持File、Xcache和APC方式	使用KVDB存储

你无需单独为SAE平台写日志功能，一切都是框架的Log类自动处理的。ThinkPHP在SAE平台的日志写入是调用了sae_debug方法，具体工作由`Think\Log\Driver\Sae`类完成。

在本地运行会将日志记录到项目的项目的`Runtime/Logs`文件夹下，而在SAE上运行会将日志记录到SAE平台的日志中心： http://sae.sina.com.cn/?m=applog

请在搜索框选择中的下拉菜单处选择“debug”进行查看。

文件上传

文件上传仍然使用`Think\Upload`扩展类库上传文件，使用方法不变。同样的代码在本地运行时将会上传到指定的目录，在SAE上运行时就会自动使用Storage服务，将文件上传到指定的Storage中。

首先你需要在SAE平台上创建一个Storage的domain用于存放上传的文件： http://sae.sina.com.cn/?m=storage

这里可以建立多个domain。而我们的文件会上传到哪个domain，是由上传路径的第一个目录名称决定的。如：

```
$upload->rootPath = './Public/';
$upload->savePath =  'Uploads/';
```

会上传到名为Public的domain。

> 你也不用在这个domain下创建Uploads文件夹，SAE的Storage服务会为你自动创建。

图片地址的问题： 我们使用UploadFile类上传图片，在本地和在SAE下图片的浏览地址是不一样的。比如有张图片地址为 "/Public/upload/1.jpg"， /Public 是一个模板替换变量， 他会被替换为Public文件夹所

在目录的地址， 我们可以通过浏览器的查看源代码功能查看被替换后是什么效果。 可以看见， 替换后为"/Public/upload/1.jpg" 。 但是在SAE上图片并没有在Public/upload目录下，而是在storage中。我们需要将 /Public/替换为storage的域名，在SAE上才能正常显示。

我们在SAE专有配置 Conf/config_sae.php 文件中定义如下代码：

```
<?php
return array(
  'TMPL_PARSE_STRING'=>array(
    '/Public/upload' => $st->getUrl('public','upload'),
  )
);
```

这样，在SAE上会把 /Public/upload 替换为storage的地址， 在SAE上图片也能正常显示。

隐藏index.php

SAE不支持 .htaccess 文件，但我们可以使用SAE提供的AppConfig服务实现伪静态。

在你项目的根目录建立 config.yaml 文件，代码为：

```
handle:
 - rewrite: if(!is_dir() && !is_file() && path~"^(.*)$") goto "index.php/$1"
```

这样就可以隐藏入口了。

比如这样的地址 http://serverName/index.php/Blog/read/id/1 也能通过 http://serverName/Blog/read/id/1 访问。

代码横跨性建议

ThinkPHP的SAE模式，是具有横跨性的，请不要破坏它的横跨性。比如，不要在项目配置文件中写和SAE数据库相关配置项。自己写代码时，也要尽量做到横跨性，这样就可以让同样的代码既能在SAE下运行，也能在普通环境下运行，使你在本地调试完后上传到SAE也不用修改任何代码就能运行。

下面是一些保持代码横跨性的建议：

（1）尽量少使用原生的SAE服务

能使用ThinkPHP自带函数替代的，尽量使用ThinkPHP自带函数。比如要使用SAE的KVDB服务，在ThinkPHP中完全可以用F函数代替。如果要使用SAE的Memcache服务，都使用S函数实现。这样就不会导致你的代码从SAE转移到普通环境后性能很低。个别SAE服务无法使用ThinkPHP自带函数代替的，才考虑使用原生的SAE服务。

（2）利用SAE专有配置

当遇到SAE和普通环境配置需要不一样时，你可以把普通环境的配置写到项目配置文件Conf/config.php中，而将SAE需要用的配置写到SAE专有配置Conf/config_sae.php中。（大多数差异化的配置，SAE惯例配置文件已经内置处理了）

IP获取和定位

系统内置了 get_client_ip 方法用于获取客户端的IP地址，使用示例：

```
$ip = get_client_ip();
```

如果要支持IP定位功能，需要使用扩展类库 Org\Net\IpLocation ，并且要配合IP地址库文件一起使用，例如：

```
$Ip = new \Org\Net\IpLocation('UTFWry.dat'); // 实例化类 参数表示IP地址库文件
$area = $Ip->getlocation('203.34.5.66'); // 获取某个IP地址所在的位置
```

如果传入的参数为空，则会自动获取当前的客户端IP地址，要正确输出位置，必须配合UTF8编码的ip地址库文件，否则可能还需要进行编码转换。

> IP地址库文件和IpLocation类库位于同一目录即可。

附录

常量参考

预定义常量

预定义常量是指系统内置定义好的常量，不会随着环境的变化而变化，包括：

```
URL_COMMON    普通模式 URL（0）
URL_PATHINFO  PATHINFO URL（1）
URL_REWRITE   REWRITE URL（2）
URL_COMPAT    兼容模式 URL（3）
EXT           类库文件后缀（.class.php）
THINK_VERSION 框架版本号
```

路径常量

系统和应用的路径常量用于系统默认的目录规范，可以通过重新定义改变，如果不希望定制目录，这些常量一般不需要更改。

```
THINK_PATH 框架系统目录
APP_PATH 应用目录（默认为入口文件所在目录）
LIB_PATH 系统类库目录（默认为 THINK_PATH.'Library/'）
CORE_PATH 系统核心类库目录（默认为 LIB_PATH.'Think/'）
MODE_PATH 系统应用模式目录（默认为 THINK_PATH.'Mode/'）
BEHAVIOR_PATH 行为目录（默认为 LIB_PATH.'Behavior/'）
COMMON_PATH 公共模块目录（默认为 APP_PATH.'Common/'）
VENDOR_PATH 第三方类库目录（默认为 LIB_PATH.'Vendor/'）
RUNTIME_PATH 应用运行时目录（默认为 APP_PATH.'Runtime/'）
HTML_PATH 应用静态缓存目录（默认为 APP_PATH.'Html/'）
CONF_PATH 应用公共配置目录（默认为 COMMON_PATH.'Conf/'）
LANG_PATH 公共语言包目录（默认为 COMMON_PATH.'Lang/'）
LOG_PATH 应用日志目录（默认为 RUNTIME_PATH.'Logs/'）
CACHE_PATH 项目模板缓存目录（默认为 RUNTIME_PATH.'Cache/'）
TEMP_PATH 应用缓存目录（默认为 RUNTIME_PATH.'Temp/'）
DATA_PATH 应用数据目录（默认为 RUNTIME_PATH.'Data/'）
ADDON_PATH 插件控制器目录（默认为 APP_PATH.'Addon'） 3.2.3新增
```

系统常量

系统常量会随着开发环境的改变或者设置的改变而产生变化。

IS_CGI 是否属于 CGI模式
IS_WIN 是否属于Windows 环境
IS_CLI 是否属于命令行模式
__ROOT__ 网站根目录地址
__APP__ 当前应用（入口文件）地址
__MODULE__ 当前模块的URL地址
__CONTROLLER__ 当前控制器的URL地址
__ACTION__ 当前操作的URL地址
__SELF__ 当前URL地址
__INFO__ 当前的PATH_INFO字符串
__EXT__ 当前URL地址的扩展名
MODULE_NAME 当前模块名
MODULE_PATH 当前模块路径
CONTROLLER_NAME 当前控制器名
CONTROLLER_PATH 当前控制器路径 3.2.3新增
ACTION_NAME 当前操作名
APP_DEBUG 是否开启调试模式
APP_MODE 当前应用模式名称
APP_STATUS 当前应用状态
STORAGE_TYPE 当前存储类型
MODULE_PATHINFO_DEPR 模块的PATHINFO分割符
MEMORY_LIMIT_ON 系统内存统计支持
RUNTIME_FILE 项目编译缓存文件名
THEME_NAME 当前主题名称
THEME_PATH 当前模板主题路径
LANG_SET 当前浏览器语言
MAGIC_QUOTES_GPC MAGIC_QUOTES_GPC
NOW_TIME 当前请求时间（时间戳）
REQUEST_METHOD 当前请求类型
IS_GET 当前是否GET请求
IS_POST 当前是否POST请求
IS_PUT 当前是否PUT请求
IS_DELETE 当前是否DELETE请求
IS_AJAX 当前是否AJAX请求
BIND_MODULE 当前绑定的模块（3.2.1新增）
BIND_CONTROLLER 当前绑定的控制器（3.2.1新增）
BIND_ACTION 当前绑定的操作（3.2.1新增）
CONF_EXT 配置文件后缀（3.2.2新增）
CONF_PARSE 配置文件解析方法（3.2.2新增）
TMPL_PATH 用于改变全局视图目录（3.2.3新增）

配置参考

惯例配置

应用设定

```
'APP_USE_NAMESPACE'     =>  true,    // 应用类库是否使用命名空间 3.2.1新增
'APP_SUB_DOMAIN_DEPLOY' =>  false,   // 是否开启子域名部署
'APP_SUB_DOMAIN_RULES'  =>  array(), // 子域名部署规则
'APP_DOMAIN_SUFFIX'     =>  '', // 域名后缀 如果是com.cn net.cn 之类的后缀必须设置
'ACTION_SUFFIX'         =>  '', // 操作方法后缀
'MULTI_MODULE'          =>  true, // 是否允许多模块 如果为false 则必须设置 DEFAULT_MODULE
'MODULE_DENY_LIST'      =>  array('Common','Runtime'), // 禁止访问的模块列表
'MODULE_ALLOW_LIST'     =>  array(),    // 允许访问的模块列表
'CONTROLLER_LEVEL'      =>  1,
'APP_AUTOLOAD_LAYER'    =>  'Controller,Model', // 自动加载的应用类库层（针对非命名空间定义类库）
 3.2.1新增
'APP_AUTOLOAD_PATH'     =>  '', // 自动加载的路径（针对非命名空间定义类库） 3.2.1新增
```

默认设定

```
'DEFAULT_M_LAYER'       =>  'Model', // 默认的模型层名称
'DEFAULT_C_LAYER'       =>  'Controller', // 默认的控制器层名称
'DEFAULT_V_LAYER'       =>  'View', // 默认的视图层名称
'DEFAULT_LANG'          =>  'zh-cn', // 默认语言
'DEFAULT_THEME'         =>  '', // 默认模板主题名称
'DEFAULT_MODULE'        =>  'Home',  // 默认模块
'DEFAULT_CONTROLLER'    =>  'Index', // 默认控制器名称
'DEFAULT_ACTION'        =>  'index', // 默认操作名称
'DEFAULT_CHARSET'       =>  'utf-8', // 默认输出编码
'DEFAULT_TIMEZONE'      =>  'PRC',  // 默认时区
'DEFAULT_AJAX_RETURN'   =>  'JSON',  // 默认AJAX 数据返回格式,可选JSON XML ...
'DEFAULT_JSONP_HANDLER' =>  'jsonpReturn', // 默认JSONP格式返回的处理方法
'DEFAULT_FILTER'        =>  'htmlspecialchars', // 默认参数过滤方法 用于I函数...
```

Cookie设置

```
'COOKIE_EXPIRE'         =>  0,    // Cookie有效期
'COOKIE_DOMAIN'         =>  '',      // Cookie有效域名
'COOKIE_PATH'           =>  '/',     // Cookie路径
'COOKIE_PREFIX'         =>  '',      // Cookie前缀 避免冲突
'COOKIE_HTTPONLY'       =>  '',     // Cookie的httponly属性 3.2.2新增
```

数据库设置

```
'DB_TYPE'               =>  '',     // 数据库类型
'DB_HOST'               =>  '', // 服务器地址
'DB_NAME'               =>  '',          // 数据库名
'DB_USER'               =>  '',      // 用户名
'DB_PWD'                =>  '',          // 密码
'DB_PORT'               =>  '',        // 端口
'DB_PREFIX'             =>  '',    // 数据库表前缀
'DB_FIELDTYPE_CHECK'    =>  false,       // 是否进行字段类型检查 3.2.3版本废弃
'DB_FIELDS_CACHE'       =>  true,        // 启用字段缓存
'DB_CHARSET'            =>  'utf8',      // 数据库编码默认采用utf8
'DB_DEPLOY_TYPE'        =>  0, // 数据库部署方式:0 集中式(单一服务器),1 分布式(主从服务器)
'DB_RW_SEPARATE'        =>  false,       // 数据库读写是否分离 主从式有效
'DB_MASTER_NUM'         =>  1, // 读写分离后 主服务器数量
'DB_SLAVE_NO'           =>  '', // 指定从服务器序号
'DB_SQL_BUILD_CACHE'    =>  false, // 数据库查询的SQL创建缓存 3.2.3版本废弃
'DB_SQL_BUILD_QUEUE'    =>  'file',   // SQL缓存队列的缓存方式 支持 file xcache和apc 3.2.3版本废弃
'DB_SQL_BUILD_LENGTH'   =>  20, // SQL缓存的队列长度 3.2.3版本废弃
'DB_SQL_LOG'            =>  false, // SQL执行日志记录 3.2.3版本废弃
'DB_BIND_PARAM'         =>  false, // 数据库写入数据自动参数绑定
'DB_DEBUG'              =>  false,  // 数据库调试模式 3.2.3新增
'DB_LITE'               =>  false,  // 数据库Lite模式 3.2.3新增
```

数据缓存设置

```
'DATA_CACHE_TIME'       =>  0,      // 数据缓存有效期 0表示永久缓存
'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_PREFIX'     =>  '',     // 缓存前缀
'DATA_CACHE_TYPE'       =>  'File',  // 数据缓存类型,支持:File|Db|Apc|Memcache|Shmop|Sqlite|Xcache|A
pachenote|Eaccelerator
'DATA_CACHE_PATH'       =>  TEMP_PATH,// 缓存路径设置 (仅对File方式缓存有效)
'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)
'DATA_PATH_LEVEL'       =>  1,        // 子目录缓存级别
```

错误设置

```
'ERROR_MESSAGE'         =>  '页面错误！请稍后再试～',//错误显示信息,非调试模式有效
'ERROR_PAGE'            =>  '', // 错误定向页面
'SHOW_ERROR_MSG'        =>  false,    // 显示错误信息
'TRACE_MAX_RECORD'      =>  100,    // 每个级别的错误信息 最大记录数
```

日志设置

```
'LOG_RECORD'            =>  false,   // 默认不记录日志
'LOG_TYPE'              =>  'File', // 日志记录类型 默认为文件方式
'LOG_LEVEL'             =>  'EMERG,ALERT,CRIT,ERR',// 允许记录的日志级别
'LOG_EXCEPTION_RECORD'  =>  false,    // 是否记录异常信息日志
```

SESSION设置

```
'SESSION_AUTO_START'    =>  true,    // 是否自动开启Session
'SESSION_OPTIONS'       =>  array(), // session 配置数组 支持type name id path expire domain 等参数
'SESSION_TYPE'          =>  '', // session hander类型 默认无需设置 除非扩展了session hander驱动
'SESSION_PREFIX'        =>  '', // session 前缀
```

模板引擎设置

```
'TMPL_CONTENT_TYPE'     =>  'text/html', // 默认模板输出类型
'TMPL_ACTION_ERROR'     =>  THINK_PATH.'Tpl/dispatch_jump.tpl', // 默认错误跳转对应的模板文件
'TMPL_ACTION_SUCCESS'   =>  THINK_PATH.'Tpl/dispatch_jump.tpl', // 默认成功跳转对应的模板文件
'TMPL_EXCEPTION_FILE'   =>  THINK_PATH.'Tpl/think_exception.tpl',// 异常页面的模板文件
'TMPL_DETECT_THEME'     =>  false,       // 自动侦测模板主题
'TMPL_TEMPLATE_SUFFIX'  =>  '.html',     // 默认模板文件后缀
'TMPL_FILE_DEPR'        =>  '/', //模板文件CONTROLLER_NAME与ACTION_NAME之间的分割符
'TMPL_ENGINE_TYPE'      =>  'Think',     // 默认模板引擎 以下设置仅对使用Think模板引擎有效
'TMPL_CACHFILE_SUFFIX'  =>  '.php',      // 默认模板缓存后缀
'TMPL_DENY_FUNC_LIST'   =>  'echo,exit',    // 模板引擎禁用函数
'TMPL_DENY_PHP'         =>  false, // 默认模板引擎是否禁用PHP原生代码
'TMPL_L_DELIM'          =>  '{',            // 模板引擎普通标签开始标记
'TMPL_R_DELIM'          =>  '}',            // 模板引擎普通标签结束标记
'TMPL_VAR_IDENTIFY'     =>  'array',     // 模板变量识别。留空自动判断,参数为'obj'则表示对象
'TMPL_STRIP_SPACE'      =>  true,       // 是否去除模板文件里面的html空格与换行
'TMPL_CACHE_ON'         =>  true,        // 是否开启模板编译缓存,设为false则每次都会重新编译
'TMPL_CACHE_PREFIX'     =>  '',         // 模板缓存前缀标识，可以动态改变
'TMPL_CACHE_TIME'       =>  0,         // 模板缓存有效期 0 为永久，(以数字为值，单位:秒)
'TMPL_LAYOUT_ITEM'      =>  '{__CONTENT__}', // 布局模板的内容替换标识
'LAYOUT_ON'             =>  false, // 是否启用布局
'LAYOUT_NAME'           =>  'layout', // 当前布局名称 默认为layout
```

URL设置

```
'URL_CASE_INSENSITIVE'  =>  true,   // 默认false 表示URL区分大小写 true则表示不区分大小写
'URL_MODEL'             =>  1,       // URL访问模式,可选参数0、1、2、3,代表以下四种模式：
// 0 (普通模式); 1 (PATHINFO 模式); 2 (REWRITE  模式); 3 (兼容模式)  默认为PATHINFO 模式
'URL_PATHINFO_DEPR'     =>  '/',    // PATHINFO模式下，各参数之间的分割符号
'URL_PATHINFO_FETCH'    =>  'ORIG_PATH_INFO,REDIRECT_PATH_INFO,REDIRECT_URL', // 用于兼容判断PATH_INFO 参数的SERVER替代变量列表
'URL_REQUEST_URI'       =>  'REQUEST_URI', // 获取当前页面地址的系统变量 默认为REQUEST_URI
'URL_HTML_SUFFIX'       =>  'html',  // URL伪静态后缀设置
'URL_DENY_SUFFIX'       =>  'ico|png|gif|jpg', // URL禁止访问的后缀设置
'URL_PARAMS_BIND'       =>  true, // URL变量绑定到Action方法参数
'URL_PARAMS_BIND_TYPE'  =>  0, // URL变量绑定的类型 0 按变量名绑定 1 按变量顺序绑定
'URL_404_REDIRECT'      =>  '', // 404 跳转页面 部署模式有效
'URL_ROUTER_ON'         =>  false,   // 是否开启URL路由
'URL_ROUTE_RULES'       =>  array(), // 默认路由规则 针对模块
'URL_MAP_RULES'         =>  array(), // URL映射定义规则
```

系统变量名称设置

```
'VAR_MODULE'          =>  'm',    // 默认模块获取变量
'VAR_CONTROLLER'      =>  'c',    // 默认控制器获取变量
'VAR_ACTION'          =>  'a',    // 默认操作获取变量
'VAR_AJAX_SUBMIT'     =>  'ajax',// 默认的AJAX提交变量
'VAR_JSONP_HANDLER'   =>  'callback',
'VAR_PATHINFO'        =>  's',    // 兼容模式PATHINFO获取变量例如 ?s=/module/action/id/1 后面的参数取决于URL_PATHINFO_DEPR
'VAR_TEMPLATE'        =>  't',    // 默认模板切换变量
'VAR_ADDON'           =>  'addon',    // 默认的插件控制器命名空间变量 3.2.2新增
```

其他设置

```
'HTTP_CACHE_CONTROL'    =>  'private',// 网页缓存控制
'CHECK_APP_DIR'         =>  true,       // 是否检查应用目录是否创建
'FILE_UPLOAD_TYPE'      =>  'Local',    // 文件上传方式
'DATA_CRYPT_TYPE'       =>  'Think',    // 数据加密方式
```

升级指导

本章旨在帮助使用3.1版本的用户更方便的升级到3.2版本，给出了升级步骤和建议。

升级须知

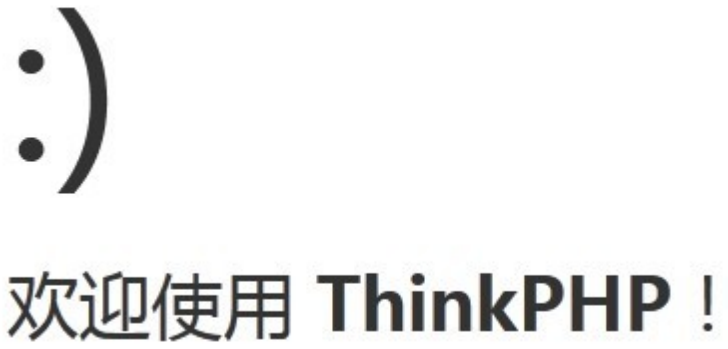
如果从3.1版本升级到3.2版本，需要注意如下的升级须知并按照升级指导的操作步骤进行。

- 3.2版本要求PHP5.3.0以上，如果环境低于该版本，将无法升级；
- 本升级指导用于指导开发人员从3.1版本升级到3.2版本；
- 如果你的项目对框架核心进行过较大的改动的话不建议升级；
- 本指导手册不确保你的项目顺利升级，不对因升级带来的任何后果负责；
- 升级项目之前请做好各项备份工作。

准备工作

- 从官网或者github下载最新版本的ThinkPHP3.2；
- 把下载的ThinkPHP3.2解压缩，得到Application、Public和ThinkPHP目录，以及一个入口文件index.php；
- 备份你的项目文件（包括ThinkPHP核心目录）到安全的位置；
- 删除项目的Runtime目录；
- 把原来的ThinkPHP系统目录更名为ThinkPHP_old；

- 把原来的index.php入口文件更名为index_old.php；
- 如果原来的项目目录为Application更名为App；
- 把解压后的Application、ThinkPHP目录，以及index.php放入你的网站目录；
- 运行新的入口文件index.php，如果显示



则准备工作已经完成，下面开始进行应用目录的调整工作。

应用目录调整

应用目录的调整分三种不同的情况：未分组/普通分组/独立分组，请根据自己的情况选择目录调整的方式。

未分组

如果你的项目未进行任何分组，请按照如下的方式调整目录结构：

删除Application/Common目录，在你的原有项目目录（假设为App）下面的Common、Conf和Lang目录移动到Application/Home目录下面，并把其中的Common/common.php文件改名为function.php，移动前后的位置类似于：

```
App/Common/common.php        =>    Application/Home/Common/function.php
App/Common/extend.php        =>    Application/Home/Common/extend.php（假设存在定义的话）
App/Conf/Config.php          =>    Application/Home/Conf/config.php
App/Lang/zh-cn/common.php    =>    Application/Home/Lang/zh-cn.php（假设存在的话）
```

把项目目录下面的Lib目录下面的所有子目录移动到Application/Home目录下面，类似于：

```
App/Lib/Action    =>    Application/Home/Action
App/Lib/Model     =>    Application/Home/Model
```

把项目目录下面的Tpl目录移动到Application/Home目录下面，并更名为View，类似于：

```
App/Tpl           =>  Application/Home/View
```

调整后的目录结构类似于：

```
Application
├─Home
│  ├─Conf       配置文件目录
│  ├─Common     公共函数目录
│  ├─Action     控制器目录
│  ├─Model      模型目录
│  └─View       模版文件目录
```

普通分组

如果你的项目采用了普通分组，则按照下面的方式进行目录调整（以Home分组为例，其他分组参考调整）：

项目公共函数目录下面的目录和文件作如下调整，类似于：

```
App/Common/common.php          => Application/Common/Common/function.php
App/Common/Home/function.php   => Application/Home/Common/function.php
```

项目目录下面的Conf目录如如下调整，类似于：

```
App/Conf/Config.php          =>    Application/Common/Conf/config.php
App/Conf/Home/config.php     =>    Application/Home/Conf/config.php
```

如果采用了语言包功能，目录如如下调整：

```
App/Lang/zh-cn/common.php       =>    Application/Common/Lang/zh-cn.php
App/Lang/zh-cn/Home/lang.php    =>    Application/Home/Lang/zh-cn.php
```

控制器目录调整如下，类似于：

```
App/Lib/Action/Home     =>    Application/Home/Action
App/Lib/Action/Admin    =>    Application/Admin/Action
```

模型目录调整如下，类似于：

```
App/Lib/Model         =>    Application/Common/Model
App/Lib/Model/Home    =>    Application/Home/Model（如果有定义）
```

模版目录调整如下，类似于：

```
App/Tpl/Home    =>  Application/Home/View
```

调整后的目录结构如下：

```
Application
├─Common        应用公共模块
│  ├─Common     应用公共函数目录
│  └─Conf       应用公共配置文件目录
├─Home          Home模块
│  ├─Action     模块控制器目录
│  ├─Common     模块函数公共目录
│  ├─Conf       模块配置文件目录
│  ├─Lang       模块语言包目录
│  ├─Model      模块模型目录
│  └─View       模块视图文件目录
```

> 其他分组参考Home分组进行调整即可。

独立分组

如果采用的是独立分组，公共函数目录作如下调整，类似于：

```
App/Common/common.php => Application/Common/Common/function.php
```

把项目目录下面的Conf、Lang移动到Application/Common目录下面，类似于：

```
App/Conf/Config.php         =>   Application/Common/Conf/config.php
App/Lang/zh-cn/common.php   =>   Application/Common/Lang/zh-cn.php
```

把独立分组目录（假设你的独立分组目录为App/Modules）下面的子目录都移动到原来的项目目录下面，类似于：

```
App/Modules/Home    => Application/Home
```

并且把Home目录下面的Tpl目录更改为View。

调整后的目录结构如下：

```
Application
├─Common        应用公共模块
│  ├─Common     应用公共函数目录
│  └─Conf       应用公共配置文件目录
├─Home          Home模块
│  ├─Action     模块控制器目录
│  ├─Common     模块函数公共目录
│  ├─Conf       模块配置文件目录
│  ├─Lang       模块语言包目录
│  ├─Model      模块模型目录
│  └─View       模块视图文件目录
```

> 其他分组参考Home分组进行调整即可。

配置调整

编辑 Application/Common/config.php（没有则创建一个新的文件），添加下面的配置参数：

```
'DEFAULT_C_LAYER'       =>  'Action', // 默认的控制器层名称
'MODULE_ALLOW_LIST'     =>  array('Home','Admin',...)， // 配置你原来的分组列表
'DEFAULT_MODULE'        =>  'Home'， // 配置你原来的默认分组
```

未分组的情况下，再添加如下配置参数：

```
'MULTI_MODULE'          =>  false, // 单模块访问
'DEFAULT_MODULE'        =>  'Home'， // 默认访问模块
```

数据库连接配置参数DB_HOST如果原来配置的是localhost或者域名，请修改为ip地址，否则会导致数据库连接缓慢（这是PHP5.3的机制问题 非TP问题），例如之前如果是配置的：

```
'DB_HOST'=>'localhost'
```

建议改为：

```
'DB_HOST'=>'127.0.0.1'
```

原来的配置参数中废弃的参数包括（增补中）：

```
APP_GROUP_LIST
APP_GROUP_MODE
APP_AUTOLOAD_PATH
APP_TAGS_ON
APP_GROUP_PATH
DEFAULT_APP
DEFAULT_GROUP
VAR_GROUP
LOG_DEST
LOG_EXTRA
```

调整的配置参数包括：

```
DEFAULT_MODULE => DEFAULT_CONTROLLER
```

别名定义调整

如果你在项目中定义了自己的别名定义文件，需要在别名定义中使用命名空间，例如：

```
'Think\Page' => CORE_PATH.'Page'.EXT,
'Think\Auth' => CORE_PATH.'Auth'.EXT,
```

行为定义调整

如果在项目中自定义了行为定义文件，那么需要修改行为定义为命名空间方式，例如：

```
'app_begin'=>array('Behavior\Cron','Behavior\BrowserCheck'),
```

路由定义调整

如果你的项目使用了路由功能，请参考下面的建议进行调整。

3.2版本的路由定义是针对模块的，所以路由定义需要放到模块配置文件中，把 `Application/Common/config.php` 中的路由定义相关的配置参数 `URL_ROUTER_ON` 和 `URL_ROUTE_RULES` 移动到相关模块的配置文件中并作适当的调整。

> 新版中路由定义规则中不需要添加模块名，如果要在URL中隐藏模块名请参考模块部署章节内容。

命名空间调整

把项目的Application/Home/Action目录下面的所有文件，头部添加如下代码（必须是除注释以外的第一行）：

```
namespace Home\Action;
use Think\Action;
```

如果你的项目使用了控制器分层的话，需要对每个分层的类库文件添加类似的代码，例如有定义Event分层的话，需要在头部添加：

```
namespace Home\Event;
use Think\Action;
```

把项目的Application/Home/Model目录下面的所有文件，头部添加如下代码（必须是除注释以为的第一行）：

```
namespace Home\Model;
use Think\Model;
```

如果你的项目使用了模型分层的话，需要对每个分层的类库文件添加类似的代码，例如如果你有Service分层，需要在头部添加：

```
namespace Home\Service;
use Think\Model;
```

对类库中的代码实现中实例化对象（包括系统内置类和自定义类）的部分调整为命名空间调用的方式，例如：

```
new Page(...)      => new \Think\Page(...)
new Pdo(...)       => new \Pdo(...)
new UserModel(...) => new \Home\Model\UserModel(...)
```

> 用ThinkPHP内置的A/D/M方法实例化的对象代码无需调整。

如果你升级的版本是3.2.1版本，应用类库的命名空间可以无需定义，但调用系统核心类库的时候仍然需要使用命名空间的方式。

在3.2.1版本中，可以在应用配置文件中设置：

```
'APP_USE_NAMESPACE'     =>   false, // 关闭应用的命名空间定义
'APP_AUTOLOAD_LAYER'    =>  'Action,Model', // 模块自动加载的类库后缀
```

设置后，应用类库无需再使用命名空间定义，只需要改成：

```
class UserAction extends Think\Action{
}
class UserModel extends Think\Model{
}
```

模型调整

如果在模型类的自动验证或者自动完成定义中使用了下面的常量，需要进行调整：

原来方式	新版方式
MODEL_INSERT	self::MODEL_INSERT
MODEL_UPDATE	self::MODEL_UPDATE
MODEL_BOTH	self::MODEL_BOTH
MUST_VALIDATE	self::MUST_VALIDATE
EXISTS_VALIDATE	self::EXISTS_VALIDATE
VALUE_VALIDATE	self::VALUE_VALIDATE

原来的 halt 函数和 _404 函数已经废除， ThrowException 也不建议使用，统一使用E函数替代。

如果你之前的项目定义了 common.php 函数文件，需要并入 Common\function.php 函数文件中。

方法调整

控制器类Think\Controller或者Think\Action的下列方法已经废除：

废除方法	替代方法
_get('id')	I('get.id')
_post('id')	I('post.id')
_put('id')	I('put.id')
_param('id')	I('id')
_request('id')	I('request.id')
_cookie('id')	I('cookie.id')
_server('id')	I('server.id')
_globals('id')	I('globals.id')

下列常量已经废除：

```
APP_NAME // 3.2版本中无需再定义该常量
__GROUP__ // 3.2版本中可以用__MODULE__ 表示模块的URL地址
GROUP_NAME // 3.2版本中可以用 MODULE_NAME 获取当前模块名
MODE_NAME  // 3.2版本中模式扩展已经废弃，参考下面的模式调整部分
```

模式调整

如果你使用了ThinkPHP的模式扩展，那么抱歉地通知您，原来的模式扩展已经废弃，命令行模式不需要单独开发，新版框架可以直接切换到命令行模式访问。如果使用了PHPRPC或者REST模式的话，请参考专题中的RPC和RESTFul部分修改。如果你使用了SAE引擎扩展的话，新版在标注模式下面可以直接部署到SAE环境，无需更改。参考SAE部分说明。

下面的模式暂时不提供支持：

```
Lite
Thin
Amf
```

自定义驱动调整

如果你在项目中自定义了相关驱动，包括数据库、标签库等，那么请参考驱动扩展部分进行调整。

模板调整

默认的模板替换行为只支持下列替换规则：

```
'__ROOT__'      =>  __ROOT__,       // 当前网站地址
'__APP__'       =>  __APP__,        // 当前应用地址
'__MODULE__'    =>  __MODULE__,
'__ACTION__'    =>  __ACTION__,     // 当前操作地址
'__SELF__'      =>  __SELF__,       // 当前页面地址
'__CONTROLLER__'=>  __CONTROLLER__,
'__URL__'       =>  __CONTROLLER__,
'__PUBLIC__'    =>  __ROOT__.'/Public',// 站点公共目录
```

对于废除的替换规则你可以在模块的配置文件中自行添加，例如：

```
'TMPL_PARSE_STRING'=>array(
  '../Public'=> MODULE_PATH.'View/Public/',
  '__TMPL__' => MODULE_PATH.'View/default/'
)
```

入口文件调整

如果你的原来项目的入口文件中（之前备份的index_old.php）还有其他代码，请调整合并到新的入口文件中，然后建议你开启调试模式后运行新的入口文件，如果仍然有错误发生，请根据错误提示进行下一步的调整或者到官网讨论区给我们反馈。

升级成功后记得删除ThinkPHP_old和原来的项目目录。

希望您的项目能够升级顺利！

鸣谢

在ThinkPHP3.2手册的编写过程中，要感谢ThinkPHP文档小组成员、社区核心团队成员和官方QQ群活跃成员、论坛活跃用户的参与和反馈，由于人数众多，不再一一列出他们的名字， 谨对他们的工作和付出表示感谢！

> 参与本文档编写的人员包括流年、Misn、麦当苗儿、luofei、deeka、yangweijie、vus520。