ThinkPHP
中文WEB应用开发框架

完全开发手册

ThinkPHP 2.0
中文WEB应用开发框架

2009 年

上海顶想信息科技有限公司

版权申明

目 录

1 简介

ThinkPHP 是一个**免费开源的，快速、简单的面向对象的轻量级 PHP 开发框架**，遵循 Apache2 开源协议发布，是为了敏捷 WEB 应用开发和简化企业级应用开发而诞生的。拥有众多的优秀功能和特性，经历了三年多发展的同时，在社区团队的积极参与下，在易用性、扩展性和性能方面不断优化和改进，众多的典型案例确保可以稳定用于商业以及门户级的开发。

ThinkPHP 借鉴了国外很多优秀的框架和模式，使用面向对象的开发结构和 MVC 模式，采用单一入口模式等，融合了 Struts 的 Action 思想和 JSP 的 TagLib（标签库）、RoR 的 ORM 映射和 ActiveRecord 模式，封装了 CURD 和一些常用操作，在项目配置、类库导入、模版引擎、查询语言、自动验证、视图模型、项目编译、缓存机制、SEO 支持、分布式数据库、多数据库连接和切换、认证机制和扩展性方面均有独特的表现。

使用 ThinkPHP，你可以更方便和快捷的开发和部署应用。当然不仅仅是企业级应用，任何 PHP 应用开发都可以从 ThinkPHP 的简单和快速的特性中受益。ThinkPHP 本身具有很多的原创特性，并且倡导**大道至简，开发由我**的开发理念，用最少的代码完成更多的功能，宗旨就是让 WEB 应用开发更简单、更快速。为此 ThinkPHP 会不断吸收和融入更好的技术以保证其新鲜和活力，提供 WEB 应用开发的最佳实践！

ThinkPHP 遵循 Apache2 开源许可协议发布，意味着你可以免费使用 ThinkPHP，甚至允许把你基于 ThinkPHP 开发的应用**开源或商业产品发布/销售**。

2 入门基础

2.1 基础概念

在学习和掌握 ThinkPHP 开发之前，我们有必要了解一些相关的基础概念，这样会更加便于后面内容的理解和掌握。

2.1.1 LAMP

LAMP 是基于 **Linux，Apache，MySQL 和 PHP** 的开放资源网络开发平台，PHP 是一种有时候用 Perl 或 Python 可代替的编程语言。这个术语来自欧洲，在那里这些程序常用来作为一种标准开发环境。名字来源于每个程序的第一个字母。每个程序在所有权里都符合开放源代码标准：Linux 是开放系统；Apache 是最通用的网络服务器；MySQL 是带有基于网络管理附加工具的关系数据库；PHP 是流行的对象脚本语言，它包含了多数其它语言的优秀特征来使得它的网络开发更加有效。开发者在 Windows 操作系统下使用这些 Linux 环境里的工具称为使用 WAMP。

虽然这些开放源代码程序本身并不是专门设计成同另外几个程序一起工作的，但由于它们都是影响较大的开源软件，拥有很多共同特点，这就导致了这些组件经常在一起使用。在过去的几年里，这些组件的兼容性不断完善，在一起的应用情形变得更加普遍。并且它们为了改善不同组件之间的协作，已经创建了某些扩展功能。目前，几乎在所有的 Linux 发布版中都默认包含了这些产品。Linux 操作系统、Apache 服务器、MySQL 数据库和 Perl、PHP 或者 Python 语言，这些产品共同组成了一个强大的 Web 应用程序平台。

随着开源潮流的蓬勃发展，开放源代码的 LAMP 已经与 J2EE 和.Net 商业软件形成三足鼎立之势，并且该软件开发的项目在软件方面的投资成本较低，因此受到整个 IT 界的关注。从网站的流量上来说，70%以上的访问流量是 LAMP 来提供的，LAMP 是最强大的网站解决方案．

2.1.2 OOP

面向对象编程（**Object Oriented Programming**，OOP，面向对象程序设计）是一种计算机编程架构。OOP 的一条基本原则是计算机程序是由单个能够起到子程序作用的单元或对象组合而成。OOP 达到了软件工程的三个主要目标：重用性、灵活性和扩展性。为了实现整体运算，每个对象都能够接收信息、处理数据和向其它对象发送信息。OOP 主要有以下的概念和组件：

组件 - 数据和功能一起在运行着的计算机程序中形成的单元，组件在 OOP 计算机程序中是模块和结构化的基础。

抽象性 - 程序有能力忽略正在处理中信息的某些方面，即对信息主要方面关注的能力。

封装 - 也叫做信息封装：确保组件不会以不可预期的方式改变其它组件的内部状态；只有在那些提供了内部状态改变方法的组件中，才可以访问其内部状态。每类组件都提供了一个与其它组件联系的接口，并规定了其它组件进行调用的方法。

多态性 - 组件的引用和类集会涉及到其它许多不同类型的组件，而且引用组件所产生的结果得依据实际调用的类型。

继承性 - 允许在现存的组件基础上创建子类组件，这统一并增强了多态性和封装性。典型地来说就是用类来对组件进行分组，而且还可以定义新类为现存的类的扩展，这样就可以将类组织成树形或网状结构，这体现了动作的通用性。

由于抽象性、封装性、重用性以及便于使用等方面的原因，以组件为基础的编程在脚本语言中已经变得特别流行。

2.1.3 MVC

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

视图 ：视图是用户看到并与之交互的界面。对老式的 Web 应用程序来说，视图就是由 HTML 元素组成的界面，在新式的 Web 应用程序中，HTML 依旧在视图中扮演着重要的角色，但一些新的技术已层出不穷，它们包括 Adobe Flash 和象 XHTML，XML/XSL，WML 等一些标识语言和 Web services。如何处理应用程序的界面变得越来越有挑战性。MVC 一个大的好处是它能为你的应用程序处理很多不同的视图。在视图中其实没有真正的处理发生，不管这些数据是联机存储的还是一个雇员列表，作为视图来讲，它只是作为一种输出数据并允许用户操纵的方式。

模型 ：模型表示企业数据和业务规则。在 MVC 的三个部件中，模型拥有最多的处理任务。例如它可能用象 EJBs 和 ColdFusion Components 这样的构件对象来处理数据库。被模型返回的数据是中立的，就是说模型与数据格式无关，这样一个模型能为多个视图提供数据。由于应用于模型的代码只需写一次就可以被多个视图重用，所以减少了代码的重复性。

控制器 ：控制器接受用户的输入并调用模型和视图去完成用户的需求。所以当单击 Web 页面中的超链接和发送 HTML 表单时，控制器本身不输出任何东西和做任何处理。它只是接收请求并决定调用哪个模型构件去处理请求，然后确定用哪个视图来显示模型处理返回的数据。

现在我们总结 MVC 的处理过程，首先控制器接收用户的请求，并决定应该调用哪个模型来进行处理，然后模型用业务逻辑来处理用户的请求并返回数据，最后控制器用相应的视图格式化模型返回的数据，并通过表示层呈现给用户。

2.1.4 ORM

对象-关系映射（**Object/Relation Mapping**，简称 ORM），是随着面向对象的软件开发方法发展而产生的。面向对象的开发方法是当今企业级应用开发环境中的主流开发方法，关系数据库是企业级应用环境中永久存放数据的主流数据存储系统。对象和关系数据是业务实体的两种表现形式，业务实体在内存中表现为对象，在数据库中表现为关系数据。内存中的对象之间存在关联和继承关系，而在数据库中，关系数据无法直接表达多对多关联和继承关系。因此，对象-关系映射(ORM)系统一般以中间件的形式存在，主要实现程序对象到关系数据库数据的映射。

面向对象是从软件工程基本原则(如耦合、聚合、封装)的基础上发展起来的，而关系数据库则是从数学理论发展而来的，两套理论存在显著的区别。为了解决这个不匹配的现象,对象关系映射技术应运而生。

2.1.5 CURD

CURD 是一个数据库技术中的缩写词，一般的项目开发的各种参数的基本功能都是 CURD。它代表创建（**Create**）、更新（**Update**）、读取（**Read**）和删除（**Delete**）操作。CURD 定义了用于处理数据的基本原子操作。之所以将 CURD 提升到一个技术难题的高度是因为完成一个涉及在多个数据库系统中进行 CURD 操作的汇总相关的活动，其性能可能会随数据关系的变化而有非常大的差异。

CURD 在具体的应用中并非一定使用 create、update 、read 和 delete 字样的方法，但是他们完成的功能是一致的。例如，ThinkPHP 就是使用 add、save、select 和 delete 方法表示模型的 CURD 操作。

2.1.6 ActiveRecord

ActiveRecord 也属于 ORM 层，由 Rails 最早提出，遵循标准的 ORM 模型：表映射到记录，记录映射到对象，字段映射到对象属性。配合遵循的命名和配置惯例，能够很大程度的快速实现模型的操作，而且简洁易懂。

ActiveRecord 的主要思想是：

1. 每一个数据库表对应创建一个类，类的每一个对象实例对应于数据库中表的一行记录；通常表的每个字段在类中都有相应的 Field；

2. ActiveRecord 同时负责把自己持久化，在 ActiveRecord 中封装了对数据库的访问，即 CURD;；

3. ActiveRecord 是一种领域模型(Domain Model)，封装了部分业务逻辑；

ActiveRecord 比较适用于：

1. 业务逻辑比较简单，当你的类基本上和数据库中的表一一对应时, ActiveRecord 是非常方便的，即你的业务逻辑大多数是对单表操作；

2. 当发生跨表的操作时, 往往会配合使用事务脚本(Transaction Script)，把跨表事务提升到事务脚本中；

3. ActiveRecord 最大优点是简单, 直观。 一个类就包括了数据访问和业务逻辑. 如果配合代码生成器使用就更方便了；

这些优点使 ActiveRecord 特别适合 WEB 快速开发。

2.1.7 单一入口

单一入口通常是指一个项目或者应用具有一个统一（但并不一定是唯一）的入口文件，也就是说项目的所有功能操作都是通过这个入口文件进行的，并且往往入口文件是第一步被执行的。

单一入口的好处是项目整体比较规范，因为同一个入口，往往其不同操作之间具有相同的规则。另外一个方面就是单一入口带来的好处是控制较为灵活，因为拦截方便了，类似如一些权限控制、用户登录方面的判断和操作可以统一处理了。

或者有些人会担心所有网站都通过一个入口文件进行访问，是否会造成太大的压力，其实这是杞人忧天的想法。

2.2 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。

最新的下载版本可以在http://thinkphp.cn/Down 下载到。

你还还可以通过 SVN 获取最新的更新版本。

SVN 地址：

完整版本http://thinkphp.googlecode.com/svn/trunk

核心版本http://thinkphp.googlecode.com/svn/trunk/ThinkPHP

更多的 ThinkPHP 相关资源：

Google 项目地址：http://code.google.com/p/thinkphp/

SF 项目地址：http://sourceforge.net/projects/thinkphp

ThinkPHP 无需任何安装，直接拷贝到你的电脑或者服务器目录下面即可。没有入口文件的调用，ThinkPHP 不会执行任何操作。

2.3 关于版本

本完全手册的内容主要针对最新的 ThinkPHP 2.0 版本，尽管有些功能在之前的版本上面也能使用，但是我们不建议在使用 1.5 版本或者更早版本开发的过程中参考（事实上，1.*以后的每个发布版本都有详细的文档）。2.*版本的体系架构和 2.0 版本是保持一致的，因此对于以后的 2.*版本，本手册中涉及的内容基本上可以适用。如有变更，会在最新的发布版本中注明。

2.4 环境要求

ThinkPHP 可以支持 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，需要 **PHP5.0 以上版本**支持，支持 Mysql、MsSQL、PgSQL、Sqlite、Oracle、Ibase 以及 PDO 等多种数据库和连接。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。

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（http://www.wampserver.com/en/ 是一个集成了 Apache、PHP 和 MySQL 的开发套件，而且可以支持不同 PHP 版本的切换）来使用 ThinkPHP 进行本地开发和测试。

2.5 许可协议

ThinkPHP 遵循 **Apache2 开源协议**发布。Apache Licence 是著名的非盈利开源组织 Apache 采用的协议。该协议和 BSD 类似，鼓励代码共享和尊重原作者的著作权，同样允许代码修改，再作为**开源或商业软件**发布。需要满足的条件：

1．需要给代码的用户一份 Apache Licence ；

2．如果你修改了代码，需要在被修改的文件中说明；

3．在延伸的代码中（修改和有源代码衍生的代码中）需要带有原来代码中的协议，商标，专利声明和其他原来作者规定需要包含的说明；

4．如果再发布的产品中包含一个 Notice 文件，则在 Notice 文件中需要带有 Apache Licence。你可以在 Notice 中增加自己的许可，但不可以表现为对 Apache Licence 构成更改。

具体的协议参考：http://www.apache.org/licenses/LICENSE-2.0。

3 架构设计

ThinkPHP 遵循了**简洁实用**的设计原则，兼顾开发速度和执行速度的同时，也注重易用性。新版在性能提升 100%的同时，还保留了足够的扩展机制。下面这部分内容会对 ThinkPHP 框架的整体思想和架构体系作简要的描述说明。

3.1 系统特性

ThinkPHP 是一个性能卓越并且功能丰富的轻量级 PHP 开发框架，本身具有很多的原创特性，并且倡导**大道至简，开发由我**的开发理念，用最少的代码完成更多的功能，宗旨就是让 WEB 应用开发更简单、更快速。从 1.*版本开始就放弃了对 PHP4 的兼容，因此整个框架的架构和实现能够得以更加灵活和简单。2.0 版本更是在之前的基础上，经过全新的重构和无数次的完善以及改进，达到了一个新的阶段，足以达到企业级和门户级的开发标准。

ThinkPHP 值得推荐的特性包括：

- **类库导入**：ThinkPHP 是首先采用基于类库包和命名空间的方式导入类库，让类库导入看起来更加简单清晰，而且还支持冲突检测和别名导入。为了方便项目的跨平台移植，系统还可以严格检查加载文件的大小写。
- **URL 模式**：系统支持普通模式、PATHINFO 模式、REWRITE 模式和兼容模式的 URL 方式，支持不同的服务器和运行模式的部署，配合 URL 路由功能，让你随心所欲的构建需要的 URL 地址和进行 SEO 优化工作。
- **编译机制**：独创的核心编译和项目的动态编译机制，有效减少 OOP 开发中文件加载的性能开销。ALLINONE 模式更是让你体验飞一般的感觉。

✧ **ORM**：简洁轻巧的 ORM 实现，配合简单的 CURD 以及 AR 模式，让开发效率无处不在。

✧ **查询语言**：内建丰富的查询机制，包括组合查询、复合查询、区间查询、统计查询、定位查询、动态查询和原生查询，让你的数据查询简洁高效。

✧ **动态模型**：无需创建任何对应的模型类，轻松完成 CURD 操作，支持多种模型之间的动态切换，让你领略数据操作的无比畅快和最佳体验。

✧ **高级模型**：可以轻松支持序列化字段、文本字段、只读字段、延迟写入、乐观锁、数据分表等高级特性。

✧ **视图模型**：轻松动态地创建数据库视图，多表查询不再烦恼。

✧ **关联模型**：让你以出乎意料的简单、灵活的方式完成多表的关联操作。

✧ **分组模块**：不用担心大项目的分工协调和部署问题，分组模块帮你解决跨项目的难题。

✧ **模板引擎**：系统内建了一款卓越的基于 XML 的编译型模板引擎，支持两种类型的模板标签，融合了 Smarty 和 JSP 标签库的思想，支持标签库扩展。通过驱动还可以支持 Smarty、EaseTemplate、TemplateLite、Smart 等第三方模板引擎。

✧ **AJAX 支持**：内置 AJAX 数据返回方法，支持 JSON、XML 和 EVAL 格式返回客户端，并且系统不绑定任何 AJAX 类库，可随意使用自己熟悉的 AJAX 类库进行操作。

✧ **多语言支持**：系统支持语言包功能，项目和模块都可以有单独的语言包，并且可以自动检测浏览器语言自动载入对应的语言包。

✧ **模式扩展**：除了标准模式外，系统内置了 Lite、Thin 和 Cli 模式，针对不同级别的应用开发提供最佳核心框架，还可以自定义模式扩展。

✧ **自动验证和完成**：自动完成表单数据的验证和过滤，生成安全的数据对象。

✧ **字段类型检测**：字段类型强制转换，确保数据写入和查询更安全。

✧ **数据库特性**：系统支持多数据库连接和动态切换机制，支持分布式数据库。犹如企业开发的一把利刃，跨数据库应用和分布式支持从此无忧。

✧ **缓存机制**：系统支持包括文件方式、APC、Db、Memcache、Shmop、Eaccelerator 和 Xcache 在内的多种动态数据缓存类型，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。

✧ **扩展机制**：系统支持包括类库扩展、驱动扩展、应用扩展、模型扩展、控制器扩展、标签库扩展、模板引擎扩展、Widget 扩展、行为扩展和模式扩展在内的强大灵活的扩展机制，让你不再受限于核心的不足和无所适从，随心 DIY 自己的框架和扩展应用。

3.2 目录结构

新版的目录结构在原来的基础上进行了调整，更加清晰。

一、**系统目录**（ThinkPHP 框架目录）

ThinkPHP.php 框架的公共入口文件

Common 包含框架的一些公共文件、系统定义、系统函数和惯例配置等

Lang 系统语言文件

Lib 系统基类库目录

Tpl 系统模板目录

Mode 框架模式扩展目录

Vendor 第三方类库目录

二、**应用目录**（项目目录）

index.php 项目入口文件（可以使用其他名称或者放置于其他位置）

Common 项目公共文件目录，一般放置项目的公共函数

Conf 项目配置目录，所有的配置文件都放在这里。

Lang 项目语言包目录（可选）

Lib 项目类库目录，通常包括 Action 和 Model 子目录

Tpl 项目模板目录，支持模板主题

Runtime 项目运行时目录，包括 Cache（模板缓存）、Temp（数据缓存）、Data（数据目录）和 Logs（日志文件）子目录

上面的只是默认方式，项目下面的目录名称和结构是可以重新定义的。其实**项目目录并不需要开发人员手动创建**，只需要定义好项目的入口文件之后，系统会在第一次执行的时候自动生成项目必须的所有目录结构（前提是项目目录具有可写权限，这点在 Linux 环境下面需要注意）。

可以看出新版的目录结构更加便于部署和配置，因为只有 Runtime 目录才是需要具备可写权限的，在 Linux 环境下面可以更加快速的部署和配置目录权限。

三、**部署目录**

当我们实际部署网站的时候，目录结构往往由于项目的复杂而变得复杂。我们推荐的部署目录结构如下：

ThinkPHP 系统目录（下面的目录结构同上面的系统目录）

Home 项目目录（下面的目录结构同上面的应用目录）

Admin 后台管理项目目录

…… 更多的项目目录

index.php 网站的入口文件

admin.php 网站的后台入口文件

如果采用分组模块的话 可以简化为一个项目目录

ThinkPHP 系统目录（下面的目录结构同上面的系统目录）

App 项目目录

Public 网站公共目录

index.php 网站的入口文件

项目的模板文件还是放到项目的 Tpl 目录下面，只是将外部调用的资源文件， 包括图片 JS 和 CSS 统一放到网站的公共目录 Public 下面，分 Images、Js 和 Css 子目录存放，如果有可能的话，甚至也可以把这些资源文件单独放一个外部的服务器远程调用，并进行优化。

这样部署的好处是系统目录和项目目录可以放到非 WEB 访问目录下面，网站目录下面可以只需要放置 Public 公共目录和 index.php 入口文件（如果是多个项目的话，每个项目的入口文件都需要放到 WEB 目录下面），从而提高网站的安全性。

3.3 MVC 分层

MVC 是一种将应用程序的逻辑层和表现层进行分离的方法。ThinkPHP 也是基于 MVC 设计模式的。MVC 只是一个抽象的概念，并没有特别明确的规定，ThinkPHP 中的 MVC 分层大致体现在：

模型（M）：模型的定义由 Model 类来完成。

控制器（C）：应用控制器（核心控制器 App 类）和 Action 控制器都承担了控制器的角色，Action 控制器完成业务过程控制，而应用控制器负责调度控制。

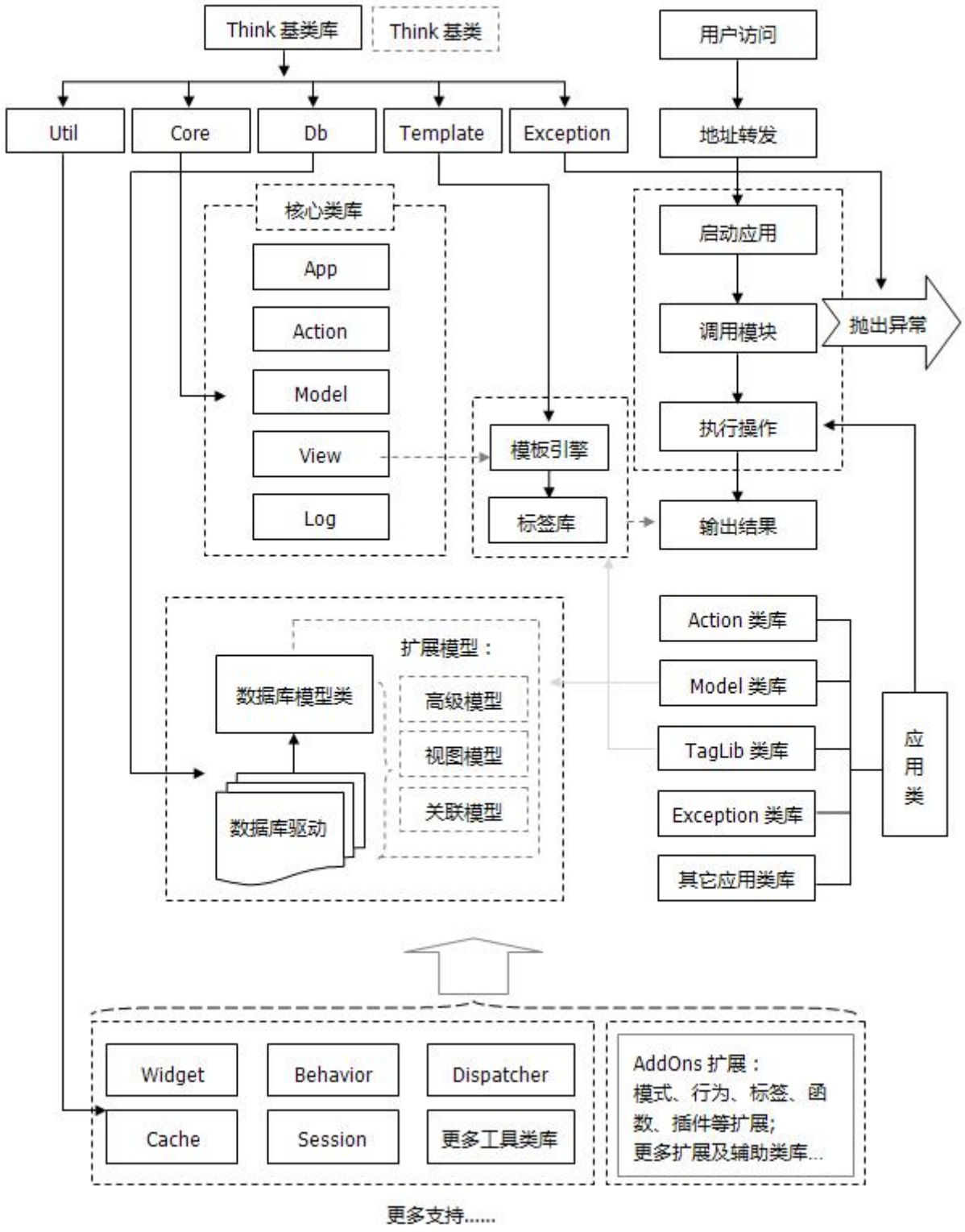
视图（V）：由 View 类和模板文件组成，模板做到了 100%分离，可以独立预览和制作。

有些时候，ThinkPHP 并不依赖 M 或者 V ，也就是说没有模型或者视图也一样可以工作。甚至也不依赖 C，这是因为 ThinkPHP 在 Action 之上还有一个总控制器，即 App 控制器，负责应用的总调度。在没有 C 的情况下，必然存在视图 V，否则就不再是一个完整的应用。

总而言之，ThinkPHP 的 MVC 模式只是提供了一种敏捷开发的手段，而不是拘泥于 MVC 本身。

3.4 执行流程

基于 ThinkPHP 框架的应用程序组成和执行过程，如图所示：



3.5 命名规范

框架必然有其自身的一定规范，在 ThinkPHP 中亦然。下面是使用 ThinkPHP 应该尽量遵循的命名规范：

- 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。
- 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）。
- 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 UserAction 类的文件命名是 UserAction.class.php， InfoModel 类的文件名是 InfoModel.class.php，
- 函数的命名使用小写字母和下划线的方式，例如 get_client_ip
- Action 控制器类以 Action 为后缀，例如 UserAction、InfoAction
- 模型类以 Model 为后缀，例如 UserModel、InfoModel
- 方法的命名使用驼峰法，并且首字母小写，例如 getUserName
- 属性的命名使用驼峰法，并且首字母小写，例如 tableName
- 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_。
- 数据表和字段采用小写加下划线方式命名，例如 think_user 和 user_name

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，ADSL 方法等等，他们有着特殊的含义，后面会有所了解。

另外一点，ThinkPHP 默认使用 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）。

3.6 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，一个项目只有一个统一（但不一定是唯一）的入口。并且所有的项目的入口文件是类似的，入口文件主要完成的作用是：

✧ 路径定义 项目名称定义（可选）

✧ 额外参数定义（可选）

✧ 载入框架入口文件（**必须**）

✧ 实例化一个 App 应用（**必须**）

下面是一个标准的入口文件的写法：

```
<?php
// 定义 ThinkPHP 框架路径(相对于入口文件)
define('THINK_PATH', '../ThinkPHP');
//定义项目名称和路径
define('APP_NAME', 'Myapp');
define('APP_PATH', '.');
```

```
// 加载框架入口文件
require(THINK_PATH."/ThinkPHP.php");
//实例化一个网站应用实例
App::run();
?>
```

3.7 项目编译

ThinkPHP 正式版本开始引入了新的项目编译机制，所谓的项目编译机制是指系统第一次运行的时候会自动生成核心缓存文件~runtime.php 和项目编译缓存文件~app.php，这些编译缓存文件把核心和项目必须的文件打包到一个文件中，并且去掉所有空白和注释代码，因为存在一个预编译的过程，所以还会进行一些相关的目录检测，对于不存在的目录可以自动生成，这个自动生成机制后面还会提到。当第二次执行的时候就会直接载入编译过的缓存文件，从而省去很多 IO 开销，加快执行速度。项目编译机制对运行没有任何影响，预编译操作和其他的目录检测机制只会执行一次，因此无论在预编译过程中做了多少复杂的操作，对后面的执行没有任何效率的缺失。

编译缓存文件，默认是自动生成在项目目录下面的 Runtime 目录下面。如果希望自己设置目录，可以在入口文件里面设置 RUNTIME_PATH 进行更改，例如

```
define('RUNTIME_PATH','./MyApp/temp/');
```

注意在 Linux 环境下面需要对 RUNTIME_PATH 目录设置可写权限。

核心编译缓存文件~runtime.php 包含的文件由系统的 **core.php** 文件决定，如果是采用了模式扩展的话，就由**模式扩展入口**文件决定。默认的核心模式下面包含了下面的一些文件：系统定义文件

defines.php、系统函数库 functions.php、系统基类 Think、异常基类 ThinkException、日志类 Log、应用类 App、控制器基类 Action、视图类 View。

其他类库可以在操作方法中使用系统导入机制或者自动加载机制完成加载。

项目编译缓存文件~app.php 通常包含了下面的一些文件：项目配置文件（由惯例配置、项目配置合并而成）、项目公共函数文件 common.php。每个项目还可以单独添加自己的项目编译文件列表，只需要在项目配置目录下面定义 **app.php** 文件，返回需要额外添加到项目编译缓存的文件列表数组即可。

注意在调试模式下面不会生成项目编译缓存，但是依然会生成核心缓存。如果不希望生成核心缓存文件的话，可以在项目入口文件里面设置 NO_CACHE_RUNTIME，例如：

```
define('NO_CACHE_RUNTIME',True);
```

以及设置对编译缓存的内容是否进行去空白和注释，例如：

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

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

3.8 URL 访问

ThinkPHP 框架基于模块和操作的方式进行访问，由于 ThinkPHP 框架的应用采用单一入口文件来执行，因此网站的所有的模块和操作都通过 URL 的参数来访问和执行。这样一来，传统方式的文件入口访问会变成由 URL 的参数来统一解析和调度。

ThinkPHP 强大的 URL 解析、调度以及路由功能为这个功能实现提供了有力的保证，并且可以在绝大多数的服务器环境里面部署成功。

ThinkPHP 支持的 URL 模式包括普通模式、PATHINFO 模式、REWRITE 模式和兼容模式，并且都提供路由支持。默认为 PATHINFO 模式，提供最好的用户体验和搜索引擎友好支持。

例如普通模式下面的 URL 为：

http://localhost/appName/index.php?m=moduleName&a=actionName&id=1

如果使用 PATHINFO 模式的话，URL 成为：

http://localhost/appName/index.php/moduleName/actionName/id/1/

PATHINFO 模式对以往的编程方式没有影响，GET 和 POST 方式传值依然有效，因为系统会对 PATHINFO 方式自动处理，例如上面 URL 地址中的 id 的值可以通过 $_GET['id'] 的方式正常获取到。

如果使用 REWRITE 模式，通过配置 URL 可以成为：

http://localhost/appName/moduleName/actionName/id/1/

例如上面生成的 myApp 项目如果我们通过下面的 URL 访问：

http://localhost/myApp/

其实是定位到 myApp 项目的 Index 模块的 index 操作，因为系统在没有指定模块和操作的时候，会执行**默认的模块和操作**，这个在 ThinkPHP 的惯例配置里面是 Index 模块和 index 操作。因此下面的 URL 和上面的结果是相同的：

http://localhost/myApp/index.php/Index/index/

通过项目配置参数，我们可以改变这个默认配置。

系统还支持分组模式和 URL 路由的功能，这些都能够带来 URL 的不同体验。

3.9 控制器

ThinkPHP 的控制器就是模块类，通常位于项目的 Lib\Action 目录下面。类名就是模块名加上 Action 后缀，例如 IndexAction 类就表示了 Index 模块。控制器类必须继承系统的 Action 基础类，这样才能确保使用 Action 类内置的方法。而 index 操作其实就是 IndexAction 类的一个公共方法，所以我们在浏览器里面输入 URL：

http://localhost/myApp/index.php/Index/index/

其实就是执行了 IndexAction 类的 index（公共）方法。

每个模块的操作并非一定需要有定义操作方法，如果我们只是希望输出一个模板，既没有变量也没有任何的业务逻辑，那么只需要按照规则定义好操作对应的模板文件即可，而不需要定义操作方法。例如，我们在 IndexAction 中如果没有定义 help 方法，但是存在对应的 Index/help.html 模板文件，那么下面的 URL 访问依然可以正常运作：

http://localhost/myApp/index.php/Index/help/

因为系统找不到 IndexAction 类的 help 方法，会自动定位到 Index 模块的模板目录中查找 help.html 模板文件，然后直接渲染输出。

控制器中还设计了模块分组、空操作、空模块、前置和后置操作、操作链等功能，后面会有详细的描述。

3.10 模型

在 ThinkPHP 中基础的模型类就是 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类中，例如 AdvModel 高级模型类完成了一些包括文本

字段、只读字段、序列化字段、乐观锁、多数据库连接等模型的高级特性，ViewModel 视图模型类完成了模型的视图操作，RelationModel 关联模型类完成了模型的关联操作。

基础模型类 Model 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。

新版实现了动态模型的设计，可以从基础模型类切换到其他模型类进行方法操作而不会丢失现有的数据属性。这是一个真正的按需加载的思想，而不再是必须要事先继承需要操作的模型类。

3.11 数据库抽象层

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，而使用了统一的操作接口。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适配器来处理。目前支持 Mysql、MsSQL、PgSQL、Sqlite、Oracle、Ibase 以及 PDO 等多种数据库和连接。

数据库抽象层也支持分布式数据库的连接，包括对等和主从方式两种的支持，而且也支持多数据库连接和切换，为企业级应用保驾护航。

3.12 视图

ThinkPHP 的视图主要由 View 视图类和模板文件构成。视图类负责 Action 控制器类和模板文件之间沟通，Action 类把数据通过 View 类传递到模板文件，而模板文件把接收到的数据转换成相应的数据格式显示。在特殊的情况下面，视图类会缓存模板文件的输出结果，这个时候缓存文件也纳入了视图层的概念之中了。

如果模板文件使用了某些模板引擎进行标签定义，而不是使用原生的 PHP 语法，那么在模板输出的过程中还需要引入模板解析，如果是编译型的模板引擎例如 ThinkPHP 内置的模板引擎和 Smarty 之类的，那么模板文件会有一个编译的过程，通常编译后的模板文件会生成一个编译后的模板缓存文件，第二次输出模板文件的时候就是直接输出编译后的模板缓存。如果是解释型的模板引擎，就会在每次输出模板的过程中进行解析操作。

无论如何，视图应该仅仅是进行数据的输出显示，通常在视图渲染过程是不会改变数据本身的，而只是进行格式化输出和显示。

3.13 模板引擎

ThinkPHP 内置了一个基于 XML 的性能卓越的模板引擎 ThinkTemplate，这是一个专门为 ThinkPHP 服务的内置模板引擎，无论在功能或是性能还有易用性方面都比 Smarty 优秀。ThinkTemplate 是一个使用了 XML 标签库技术的编译型模板引擎，使用了动态编译和缓存技术，支持两种类型的模板标签，支持 PHP 原生代码和模板标签的混合使用。而且支持自定义标签库,在基于内置模板引擎的基础上，扩展更多更强大更适合自己项目所使用的模板标签，任何想达到的功能皆有可能。

3.14 函数库

函数无论在 PHP 还是在框架中都起到了非常重要的作用，是我们完成快速开发的有效辅助。

3.14.1 系统函数库

系统函数库位于系统的 Common 目录下面，函数库文件名为 functions.php，该文件会在执行过程自动加载，系统函数库中的大部分方法是核心所依赖或者经常被使用的，因此系统函数库的所有函数都可以在任何时候直接使用。

除了系统函数库外，系统还内置了一个扩展函数库 extend.php，供项目开发的过程中加载调用，扩展函数库中的函数通常是核心不依赖的，但却有很好的辅助作用，能够为应用开发提供进一步的方便。需要使用扩展函数库中的方法，可以直接拷贝到你的项目函数库中。

3.14.2 快捷方法

ThinkPHP 为一些常用的操作定义了快捷方法，这些方法以单字母命名，具有比较容易记忆的特点。非常有意思的是，这些快捷方法的字母包含了 ADSL 字母，所以我们称之为 ADSL 方法，但是并不局限于 ADSL 四个方法，包括下面的：

A 快速实例化 Action 类库

B 执行行为类

C 配置参数存取方法

D 快速实例化 Model 类库

F 快速简单文本数据存取方法

L 语言参数存取方法

M 快速高性能实例化模型

R 快速远程调用 Action 类方法

S 快速缓存存取方法

U URL 动态生成和重定向方法

W 快速 Widget 输出方法

由上可知，快捷方法的命名方式，一般是以该方法所对应的符合其功能意义的英文单词首字母进行命名，至于每个快捷方法的详细使用，我们会在具体的章节中有针对的描述或者参考附录部分。

3.14.3 项目函数库

项目函数库通常位于项目的 Common 目录下面，文件名为 common.php，该文件会在执行过程中自动加载，并且合并到项目编译统一缓存，如果使用了分组部署方式，并且该目录下存在"分组名称/function.php"文件，也会根据当前分组执行时对应进行自动加载，因此项目函数库的所有函数也都可以无需手动载入而直接使用。

3.15 类库

3.15.1 基类库

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

基类库目录位于系统目录下面的 Lib 目录，框架内置的有 Think 核心类库，还可以扩展 ORG 、Com 扩展类库。核心基类库的作用是完成框架的通用性开发而必须的基础类和常用工具类等，包含有：

Think.Core 核心类库包

Think.Db 数据库类库包

Think.Exception 异常处理类库包

Think.Template 内置模板引擎类库包

Think.Util 系统工具类库包

3.15.2 扩展类库

官方网站额外提供了很多的基类库扩展，可以直接带路径拷贝类库文件到系统的基类库目录就可以使用了。例如，我们要使用扩展类库的 ORG/Util/Page.class.php 的话，把 Page 类库拷贝到系统目录下面的 Lib/ORG/Util/目录即可。

目前可以支持的扩展类库包，包括 ORG 和 Com。所有扩展类库必须放置于上面两个类库包之下管理。

3.15.3 应用类库

应用类库是指项目中自己定义或者使用的类库，这些类库也是遵循 ThinkPHP 的命名规范。应用类库目录位于项目目录下面的 Lib 目录。应用类库的范围很广，包括 Action 类库、Model 类库或者其他的工具类库。

3.15.4 类库导入

ThinkPHP 模拟了 Java 的类库导入机制，统一采用 import 方法进行类文件的加载。import 方法是 ThinkPHP 内建的类库和文件导入方法，提供了方便和灵活的文件导入机制，完全可以替代 PHP 的 require 和 include 方法。例如：

```
import("Think.Util.Session");
import("App.Model.UserModel");
```

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

```
import("Think.Util.Array");
```

```
import("ORG.Util.Array");
```

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

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，否则会引入文件失败。

对于 import 方法，系统会自动识别导入类库文件的位置，ThinkPHP 的约定是 **Think**、**ORG**、**Com** 包的导入以**系统基类库**为相对起始目录，否则就认为是项目应用类库为起始目录。

```
import("Think.Util.Session");
import("ORG.Util.Page");
```

上面两个方法分别导入了系统目录下的 Lib/Think/Util/Session.class.php 和 Lib/ORG/Util/Page.class.php 类文件。

要导入项目的应用类库文件也很简单，使用下面的方式就可以了，和导入基类库的方式看起来差不多：

```
import("MyApp.Action.UserAction");
import("MyApp.Model.InfoModel");
```

上面的方式分别表示导入 MyApp 项目下面的 Lib/Action/UserAction.class.php 和 Lib/Model/InfoModel.class.php 类文件。通常我们都是在当前项目里面导入所需的类库文件，所以，我们可以使用下面的方式来简化代码

```
import("@.Action.UserAction");
import("@.Model.InfoModel");
```

除了看起来简单一些外，还可以方便项目类库的移植。

如果要在当前项目下面导入其他项目的类库，必须保证两个项目的目录是平级的，否则无法使用

```
import("OtherApp.Model.GroupModel");
```

的方式来加载其他项目的类库。

我们知道，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。

对于 import 方法，系统会自动识别导入类库文件的位置，如果是其它情况的导入，需要指定 baseUrl 参数，也就是 import 方法的第二个参数。例如，要导入当前文件所在目录下面的 RBAC/AccessDecisionManager.class.php 文件，可以使用：

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

3.15.5 导入第三方类库

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

例如，我们把 Zend 的 Filter\Dir.php 放到 Vendor 目录下面，这个时候 Dir 文件的路径就是

Vendor\Zend\Filter\Dir.php，我们使用 vendor 方法导入只需要使用：

Vendor('Zend.Filter.Dir');

就可以导入 Dir 类库了。

3.15.6 别名导入

新版 ThinkPHP 引入了别名导入功能，可以预先定义好相关类库的路径，在需要使用的时候根据定义的别名进行快速导入。别名导入功能已经和 import 方法整合，所以我们可以统一使用 import 方法进行导入，例如：

import('AdvModel');

如果有定义 AdvModel 别名，则 import 方法会自动加载定义的别名导入。

系统默认的别名定义文件位于系统的 Common\alias.php，每个模式和项目都可以定义自己的别名定义文件。

3.15.7 自动加载

在很多情况下，我们可以利用框架的自动加载功能，完成类库的加载工作，而无需我们手动导入所需要使用的类库。这些情况包括：

✧ 系统和项目中已经定义的别名导入；

✧ 当前项目下面的 Action 类库和 Model 类库文件；

✧ 自动加载路径中的类库文件；

这里的自动加载路径，是指 ThinkPHP 的配置参数 **APP_AUTOLOAD_PATH** 所定义的路径。

APP_AUTOLOAD_PATH 参数是用于设置框架的自动导入的搜索路径的，默认的配置是Think.Util.，因此才会实现自动导入 Think.Util 工具类库。例如，我们需要增加 ORG.Util.路径作为类库搜索路径，可以使用：

'APP_AUTOLOAD_PATH'=> 'Think.Util.,ORG.Util.',

多个搜索路径之间用逗号分割，并且注意定义的顺序代表了搜索的顺序。

3.16 扩展

新版在保证核心简洁高效的同时保留了足够的扩展机制，让开发人员可以更好的扩展开发以满足项目或者自身的特殊需要。

目前可以支持的扩展包括：类库扩展、模型扩展、控制器扩展、应用扩展、标签库扩展、模板引擎扩展、模式扩展、行为扩展、Widget 扩展。

4 构建应用

ThinkPHP 具有项目目录自动创建功能，因此构建项目应用程序非常简单，您只需要定义好项目的入口文件，在第一次访问入口文件的时候，系统就会自动根据您在入口文件中所定义的目录路径，迅速为您创建好项目的相关目录结构。由于新版无需创建单独的模型类，所以要创建一个基于数据库的应用，是如此的轻松和简单。

这里以一个简单的数据库应用为例，讲解下如何使用 ThinkPHP 快速构建应用。

4.1 开发流程

使用 ThinkPHP 创建应用的一般开发流程是：

✧ 创建数据库和数据表；（没有数据库操作可略过）

✧ 项目命名并创建项目入口文件；

✧ 完成项目配置；（无需额外配置可以忽略）

✧ 创建控制器类；

✧ 创建模型类；（如果只是简单的模型类可以不必创建）

✧ 创建模板文件；

✧ 运行和调试。

为了顺利完成下面的操作，我们首先在数据库创建一个测试表，以 MySQL 为例：

```
CREATE TABLE `think_demo` (
  `id` int(11) unsigned NOT NULL auto_increment,
  `title` varchar(255) NOT NULL default '',
```

```
  `content` longtext NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 ;
```

4.2 入口文件

我们给项目命名为 Myapp，并且在 WWW 目录下面创建一个 Myapp 目录（项目目录），并且把下载的 ThinkPHP 核心目录放到该目录下面。

然后在 Myapp 目录下面创建一个入口文件 index.php，其中内容如下：

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

注意，APP_PATH 的路径指的是项目目录所在路径，而不是项目入口文件所在的路径。APP_NAME 通常都必须和项目目录名称一致。

如果你的项目入口文件放到项目目录下面的话，可以无需定义 APP_NAME 和 APP_PATH，系统可以自动识别。THINK_PATH 通常也不是必须的。

因为我们的入口文件位于项目目录下面，因此，上面的入口文件可以简化为：

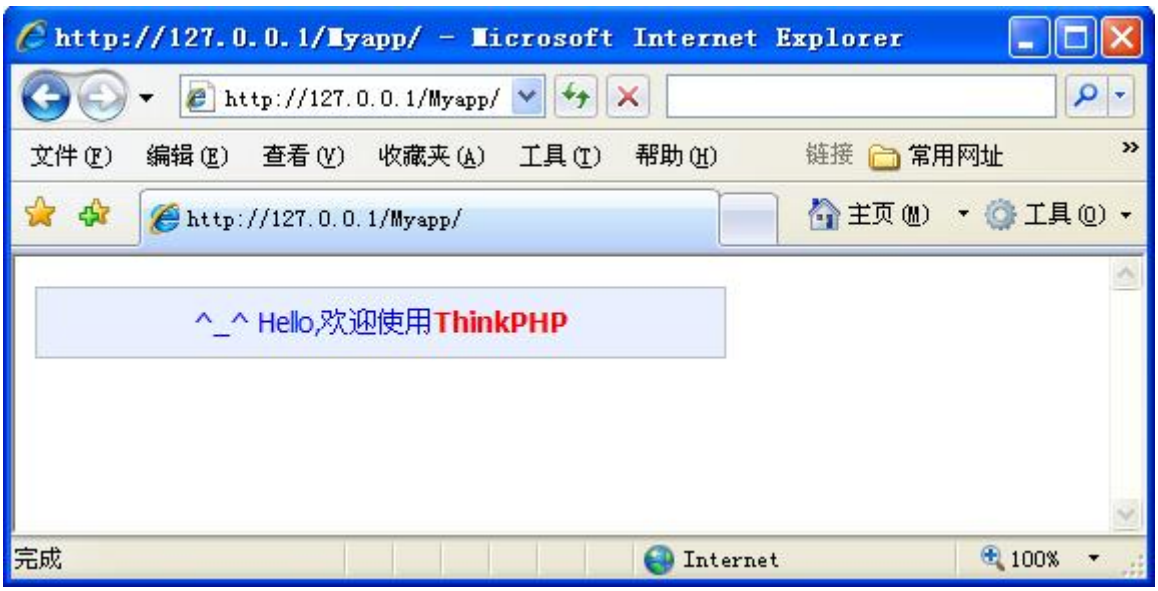
```
<?php
// 加载框架入口文件
require(" ./ThinkPHP/ThinkPHP.php");
//实例化一个网站应用实例
App::run();
?>
```

4.3 自动生成

ThinkPHP 具备项目目录自动生成功能，并且不需要使用任何命令行工具。我们只需要简单的浏览器里面访问刚才创建的应用入口文件。

打开浏览器，访问该项目的入口文件：http://127.0.0.1/Myapp/index.php

这时可以看到项目构建成功后的提示画面，并且在 Myapp 目录下，已为您构建好了项目目录。



注意：ThinkPHP 框架的所有文件都是采用 UTF-8 编码保存，但是这不影响你的项目中使用其他编码开发和浏览。请注意确保文件保存的时候去掉 UTF-8 的 BOM 头信息，防止因产生隐藏的输出而导致程序运行不正常。

注意：如果你是在 Linux 环境下，要确保项目目录的自动生成，请设置 Myapp 目录的权限为可写，否则请自行创建相关目录。然后设置 Runtime 目录为可写权限（通常都是设置目录属性为 777）。

4.4 项目配置

自动生成的项目目录下面已经为我们创建了一个空的项目配置文件，位于项目的 Conf 目录下面，名称是 config.php。我们打开这个配置文件，加入我们的数据库配置信息。

```
<?php
return array(
    'APP_DEBUG' => true, // 开启调试模式
   'DB_TYPE'=> 'mysql',    // 数据库类型
   'DB_HOST'=> 'localhost', // 数据库服务器地址
```

```
    'DB_NAME'=>'demo',  // 数据库名称
    'DB_USER'=>'root', // 数据库用户名
    'DB_PWD'=>'', // 数据库密码
    'DB_PORT'=>'3306', // 数据库端口
    'DB_PREFIX'=>'think_', // 数据表前缀
);?>
```

根据你本地的数据库连接信息修改上面的配置内容，修改完成后，保存项目配置文件。

4.5 业务逻辑

接下来，我们需要实现一个数据添加和查询操作的简单应用，来领略下 ThinkPHP 的快速开发。

在项目的 Lib\Action 目录下面找到自动生成的 IndexAction.class.php 文件，这个文件就是 ThinkPHP 的控制器，也就是 Index 模块的实现。删除 IndexAction 类默认生成的 index 方法。添加新的 insert 方法和 index 方法，代码如下：

```
// 数据写入操作
public function insert() {
    $Demo = new Model('Demo');   // 实例化模型类
    $Demo->Create(); // 创建数据对象
    $result = $Demo->add(); // 写入数据库
    $this->redirect('index'); // 成功后重定向到 index 操作页面
 }
```

```
// 数据查询操作
public function index() {
    $Demo = new Model('Demo'); // 实例化模型类
    $list = $Demo->select(); // 查询数据
    $this->assign('list',$list); // 模板变量赋值
    $this->display(); // 输出模板
}
```

以上定义后，Index 模块就具有了 insert 和 index 两个操作，操作方法的定义不需要使用任何参数，而且必须定义为 public 类型，否则无法访问。

由于只是简单的数据操作应用，所以我们根本不需要创建任何的模型类也同样可以进行 CURD 操作，这就是新版的魅力所在。^_^

4.6 模板定义

控制器和操作方法已经创建完毕，接下来就是定义模板文件了。

项目的自动生成已经为我们生成了 Tpl/default 目录，我们只需要在 default 目录下面创建 Index 目录，表示存放 Index 模块的模板文件。由于 insert 操作是后台操作，并不涉及模板输出，因此不需要定义模板文件，所以我们只要为 index 操作定义模板即可，内容如下：

```
<!--数据新增表单-->
 <form method="post"  action="__URL__/insert" >
标题：<input type="text" name="title"><br />
```

内容：<textarea name="content" rows="5" cols="25"></textarea>

<input type="submit" value="新增数据">

</form>

<!--循环输出查询结果数据集-->

<volist name='list' id='vo' >

编号:{$vo.id}

标题： {$vo.title}

内容： {$vo.content}<hr>

</volist>

把上面的内容保存为 Tpl/default/Index/index.html 即可。

action="__URL__/insert" 表示提交表单到当前模块的 insert 操作。

4.7 运行应用

模板定义完成后，我们就可以运行应用了。我们在浏览器里面输入：

http://localhost/Myapp/ 就可以看到页面的表单输出了。

由于我们开启了调试模式，所以在页面的最下面还会看到一些额外的调试信息，并且可以很清楚的看到当前页面的请求信息和执行时间、SQL 日志，最后还有加载的文件列表，事实上，页面 Trace 信息的显示完全是可以定制的，而这些内容不需要在模板里面定义。

在 ThinkPHP 中，我们称之为页面 Trace 信息，这是为了在开发过程中调试用的，关闭调试模式后，这些信息会自动消失。另外在调试模式下面，由于开启了日志记录，并且关闭了所有缓存，所以执行效率会有一定影响，但是关闭调试模式后，效率会有非常显著的提高。

可以尝试在页面新增数据，会看到页面下面有列表数据输出。到目前为止，我们已经完成了一个完整的数据操作应用了。

5 开发指南

5.1 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、项目配置、调试配置和模块配置，并且会自动生成配置缓存文件，无需重复解析的开销。对于有些简单的应用，你无需配置任何配置文件，而对于复杂的要求，你还可以增加模块配置文件，另外 ThinkPHP 的动态配置使得你在开发过程中可以灵活的动态调整配置参数。

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

惯例配置→ 项目配置→ 调试配置→ 分组配置→模块配置→ 操作（动态）配置

以上是配置文件的加载顺序，但是因为后面的配置会覆盖之前的配置（在没有生效的前提下），所以优先顺序从右到左。系统的配置参数是通过静态变量全局存取的，存取方式非常简单高效。

5.1.1 配置格式

ThinkPHP 框架中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：

```
<?php return array(
    'APP_DEBUG' => true,
    'URL_MODEL' => 2,
     // 更多的配置参数
     // ……
);?>
```

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：

```
<?php return array(
```

```
    'app_debug' => true,
    'url_model' => 2,
);?>
```

但是习惯上保持大写定义的原则。

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：

```
<?php return array(
    'APP_DEBUG' => true,
   'USER_CONFIG' => array(
            'USER_AUTH' => true,
            'USER_TYPE' => 2,
   ),
);?>
```

系统目前最多支持二维数组的配置级别，每个项目配置文件除了定义 ThinkPHP 所需要的配置参数之外，开发人员可以在里面添加项目需要的一些配置参数，用于自己的应用。项目配置文件的位置默认位于项目的 Conf 目录。

5.1.2 惯例配置

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

（如果需要了解惯例配置中的详细配置列表请参考附录的配置参考部分。）

5.1.3 项目配置

这里的项目配置指的是项目的全局配置，因为一个项目除了可以定义项目配置文件之外，还可以定义模块配置文件用于针对某个特定的模块进行特殊的配置。他们的定义格式都是一致的，区别只是配置文件命名的不同。系统会自动在不同的阶段读取配置文件。

项目配置文件位于项目的配置文件目录（默认是 Conf）下面，文件名是 config.php。

在项目配置文件里面除了添加内置的参数配置外，还可以额外添加项目需要的配置参数。

后面的开发指南中提及的配置参数设置如未特别说明，都是指在项目配置文件中定义。

5.1.4 调试配置

如果启用了调试模式的话，那么会导入框架默认的调试配置文件，默认的调试配置文件位于 Think\Common\debug.php，如果没有检测到项目的调试配置文件，就会直接使用默认的调试配置参数。项目定义了自身的调试配置文件的话，则会和默认的调试配置文件合并，也就是说，项目配置文件也只需要配置和默认调试配置不同的参数或者新增的参数。

调试配置文件也位于项目配置目录下面，文件名是 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。系统默认的调试配置文件中设置了：

- ✧ 开启日志记录
- ✧ 关闭模板缓存
- ✧ 记录 SQL 日志

- ✧ 关闭字段缓存
- ✧ 开启运行时间详细显示（包括内存、缓存情况）
- ✧ 开启页面 Trace 信息显示
- ✧ 严格检查文件大小写（即使是 Windows 平台）

由于以上的设置涉及到较多的文件 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，不过不用担心，一旦关闭调试模式，性能即可恢复理想的效果。

5.1.5 分组配置

分组配置用于系统启用了分组模式的情况之下，对于每个分组可以单独定义自己的配置文件。

分组配置文件位于：项目配置目录/分组名称/config.php

分组配置的定义格式和项目配置是一样的。分组名称区分大小写。

5.1.6 模块配置

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

项目配置目录/模块名(小写)_config.php // 用于不使用分组的情况

或者

项目配置目录/分组名/模块名(小写)_config.php // 用于使用分组的情况

模块配置文件的定义格式和项目配置相同。需要注意的是，有些配置参数在读取模块配置之前已经生效，因此可能会发生定义后不起作用的情况。

5.1.7 读取配置

定义了配置文件之后，可以使用系统提供的 C 方法来读取已有的配置：

```
C('参数名称') // 获取已经设置的参数值
```

例如，C('APP_DEBUG') 可以读取到系统的调试模式的设置值，同样，由于配置参数不区分大小写，因此 C('app_debug') 是等效的，但是建议使用大写方式的规范。

如果 APP_DEBUG 尚未存在设置，则返回 NULL。

C 方法同样可以用于读取二维配置：

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

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，哪怕某个设置参数已经生效过期了。后面我们还会了解到 C 方法同样还具有给配置参数赋值的作用。（如果对 C 方法的命名比较奇怪的话，可以借助 Config 单词来帮助记忆）

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

设置新的值：

```
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

```
C(' DATA_CACHE_TIME','60');
```

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。因此掌握 C 方法的使用对于掌握配置有着关键的作用。

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');
```

5.1.9 扩展配置

新版的配置文件都具有扩展能力，以往的项目配置文件只有一个配置文件（调试配置和模块配置文件除外），但是新版可以增加任何需要的配置文件定义，在真正执行的过程中会自动汇总到项目配置缓存里面去，而且都可以通过 C 方法来调用。

通常扩展配置文件的定义是为了某个特殊的需要，而分离出来的配置文件，这样的目的是为了便于维护和便于管理。系统也内置了一些扩展配置文件的定义，其中包括标签库定义，路由定义，静态定义，扩展模块定义，扩展操作定义，标签定义。惯例配置如下：

```
'APP_CONFIG_LIST' => array('taglibs','routes','htmls','modules','actions','tags'),
```

对于已经定义好的扩展配置文件系统会自动导入，并加入项目配置的缓存文件里面。例如：

路由配置文件 routes.php 的定义会自动并入：

```
C('_routes_');
```

后面怎么用这个扩展配置，就完全看应用自己的需要了，扩展配置对于扩展配置文件的某个配置项的获取，使用下面的方式：

```
C('_扩展配置名称_.configName');

// 例如
```

C('_modules_.extend');

如果需要增加额外的扩展配置文件，只需要在项目的配置文件里面增加额外的配置文件名称即可，例如：

'APP_CONFIG_LIST' => array('taglibs','routes','htmls','modules','actions','tags','myconfig')

注意事项：

✧ 扩展配置文件更改后，需要删除项目编译缓存文件才会生效；

✧ 对于没有定义的扩展配置文件系统不会自动加载；

✧ 注意扩展配置文件里面的配置参数的获取方式有别于一般的项目配置参数。

✧ 考虑到扩展配置的特殊需要，扩展配置里面的设置项是有大小写区分的。

5.2 控制器

5.2.1 模块和操作

ThinkPHP 采用模块和操作的方式来执行，首先，用户的请求会通过入口文件生成一个应用实例，应用控制器（我们称之为核心控制器）会管理整个用户执行的过程，并负责模块的调度和操作的执行，并且在最后销毁该应用实例。任何一个 WEB 行为都可以认为是一个模块的某个操作，系统会根据当前的 URL 来分析要执行的模块和操作。这个分析工作由 URL 调度器来实现，官方内置了 Dispatcher 类来完成该调度。

在 Dispatcher 调度器中，会根据

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

来获取当前需要执行的项目（appName）、模块（moduleName）和操作（actionName），在某些情况下，appName 可以不需要（通常是网站的首页，因为项目名称可以在入口文件中指定，这种情况下，appName 就会被入口文件替代）。在复杂一点的情况下面，可能还会出现分组（groupName）。

每个模块是一个 Action 文件，类似于我们平常所说的控制器，系统会自动寻找项目类库 Action 目录下面的相关类，如果没有找到，则会定位到空模块，否则抛出异常。

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

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

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

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

如果我们要执行下面的 URL

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

你需要增加一个 add 方法就可以了，例如

```
Class UserAction extends Action{
  // 定义一个 add 操作方法，注意操作方法不需要任何参数
  Public function add(){
    // add 操作方法的逻辑实现
    // ……
```

```
        $this->display(); // 输出模板页面
    }
}
```

操作方法必须定义为 Public 类型，否则会报错。并注意操作方法的命名不要和内置的 Action 类的方法重复。系统会自动定位当前操作的模板文件，而默认的模板文件应该位于项目目录下面的 Tpl\default\User\add.html。

5.2.2 默认模块和操作

如果使用 http://<serverName>/index.php，没有带任何模块和操作的参数，系统就会寻找默认模块和默认操作，通过 **DEFAULT_MODULE** 和 **DEFAULT_ACTION** 来定义，系统的默认模块设置是 Index 模块，默认操作设置是 index 操作。也就是说：

http://<serverName>/index.php 和

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

http://<serverName>/index.php/Index/index 等效。

可以在项目配置文件中修改默认模块和默认操作的名称。

5.2.3 模块分组

模块分组功能是为了更好的组织已有的模块，并且增加项目容量的一个有效机制。分组功能可以把以往的多项目合并到一个项目中去，这样一来，之前需要采用跨项目操作的地方，现在因为在一个项目中从而免去了不少麻烦，并且公共文件的重用也方便了，并且每个分组都可以有自己独立的配置文件、公共文件、语言包，在 URL 的访问上面也非常清晰。

要启用分组模块非常简单，配置下 **APP_GROUP_LIST** 参数和 **DEFAULT_GROUP** 参数即可。

例如我们把当前的项目分成 Home 和 Admin 两个组，分别表示前台和后台功能，那么只需要进行下面的配置：

'APP_GROUP_LIST'=>'Admin,Home',

'DEFAULT_GROUP'=>'Home',

需要注意的是，一定要把上面的配置参数放入项目的配置文件，而不是项目的分组配置或者模块配置文件。多个分组之间用逗号分隔即可，默认分组只允许设置一个。

在我们启用项目分组之前，由于使用的两个项目，所以 URL 地址分别是：

http://<serverName>/index.php/Index/index Home 项目地址

http://<serverName>/Admin/index.php/Index/index Admin 项目地址

采用了分组模式后，URL 地址变成：

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

如果 Home 是默认分组的话 还可以变成 http://<serverName>/index.php/Index/index

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

如果设置了隐藏 index.php 的话，两者的 URL 表现效果基本上是一致的，但是从管理和公共调用的角度来看，确实方便了不少。当使用分组模式时，目录结构只是做了一点小小的扩展，主要区别在于项目类库目录和模板目录下面多了一层分组目录。

如果不使用分组模式的话，Action 目录下面应该是所有的 Action 类库，现在我们可以在 Action 目录下面创建自己的分组目录，例如我们把当前项目分成了 Home 和 Admin 两个组，那么就需要在 Action 目录下面创建 Home 和 Admin 目录，然后把属于各自的 Action 类库放到对应的目录下面。如果某个 Action

类库是每个分组都需要使用或者公共继承的话，可以把这个公共 Action 类库放到分组目录之外，并且利用 ThinkPHP 的自动加载机制无需手动引入。

使用了模块分组后，如果需要实例化其他分组的模块类，可以使用：

A('Home.User')；// 实例化 Home 分组的 UserAction 类

对于分组模式下面的 Model 类库是否需要分组完全看项目的需要，由于通常不同的分组对应的数据表是相同的，因此，我们推荐 Model 类库不分组存放，仍然保留之前的方式，无论是什么分组都公共调用 Model 类库。如果确实需要分组的话，仍然可以按照 Action 的方式，在 Model 目录下面创建 Home 和 Admin 目录，然后放入对应的 Model 类库，采用这种方式的话，模型类的调用方法有所区别。

如果模型类也分组存放，在使用 D 方法调用的时候需要使用：

$User = D('Home.User')；// 实例化 Home 分组下面的 UserModel 类

模板文件的分组和 Action 类库分组也基本类似，在原来的模板主题目录下面增加一个分组目录即可。

例如：

Tpl/default/Home/Index/index.html

Tpl/default/Admin/User/index.html

相比之前的模板文件位置就是多了一个分组目录 Home 和 Admin，如果觉得目录结构太深了，可以配置 TMPL_FILE_DEPR 参数 来减少目录层次，该参数默认是 “/” ，如果改成

'TMPL_FILE_DEPR'=>'_'

那么分组的模板文件就变成了

Tpl/default/Home/Index_index.html

Tpl/default/Admin/User_index.html

分组模块的概念，并不局限于将项目区分为前台和后台。你可以按自己所需类型，进行明确细致的区分，这样非常方便于项目管理和开发部署。

分组模块下面的具体模块和之前的模块功能没有任何区别，已有的 URL 和模块功能都可以很好的支持，例如空模块、空操作、伪静态等等。

更多的关于分组模式下面 URL 方面的区别可以查看 URL 生成部分的 U 方法的使用。

5.2.4 URL 模式

我们在上面的执行过程里面看到的 URL 是默认情况下，其实 ThinkPHP 支持四种 URL 模式，可以通过设置 URL_MODEL 参数来定义，包括普通模式、PATHINFO、REWRITE 和兼容模式。

一、普通模式 ：设置 URL_MODEL 为 0

采用传统的 URL 参数模式

http://<serverName>/appName/?m=module&a=action&id=1

普通 URL 模式和在关闭 URL_DISPATCH_ON 的情况下面的效果是一样的，只是普通 URL 模式还具有路由功能。如果你并不需要使用路由功能，而且还在使用普通模式的话，建议直接关闭 URL_DISPATCH_ON，效率会更高。

二、**PATHINFO 模式** ：设置 URL_MODEL 为 1

默认情况使用 PATHINFO 模式，ThinkPHP 内置强大的 PATHINFO 支持，提供灵活和友好 URL 支持。PATHINFO 模式根据不同的设置还包括普通模式和智能模式两种：

普通模式 设置 **URL_PATHINFO_MODEL** 参数为 1

该模式下面 URL 参数没有顺序，例如

http://<serverName>/appName/m/module/a/action/id/1

http://<serverName>/appName/a/action/id/1/m/module

以上 URL 等效

智能模式 设置 **URL_PATHINFO_MODEL** 参数为 2 （系统默认的模式）

自动识别模块和操作，例如

http://<serverName>/appName/module/action/id/1/ 或者

http://<serverName>/appName/module,action,id,1/

在智能模式下面，第一个参数会被解析成模块名称（或者路由名称，下面会有描述），第二个参数会被解析成操作（在第一个参数不是路由名称的前提下），后面的参数是显式传递的，而且必须成对出现，例如：

http://<serverName>/appName/module/action/year/2008/month/09/day/21/

其中参数之间的分割符号由 **URL_PATHINFO_DEPR** 参数设置，默认为"/"，例如我们设置 URL_PATHINFO_DEPR 为 "-" 的话，就可以使用下面的 URL 访问

http://<serverName>/appName/module-action-id-1/

注意不要使用":" 和"&"符号进行分割，该符号有特殊用途。

略加修改，就可以展示出富有诗意的 URL，呵呵～

如果想要简化 URL 的形式可以通过路由功能（后面会有描述）以及空模块和空操作。

在 PATH_INFO 模式下面，会把相关参数转换成 GET 变量，以及并入 REQUEST 变量，因此不妨碍 URL 里面的 GET 和 REQUEST 变量获取。

三、**REWRITE 模式：** 设置 URL_MODEL 为 2

该 URL 模式和 PATHINFO 模式功能一样，除了可以不需要在 URL 里面写入口文件，和可以定义.htaccess 文件外。在开启了 Apache 的 URL_REWRITE 模块后，就可以启用 REWRITE 模式了，具体参考下面的 URL 重写部分。

四、**兼容模式：** 设置 URL_MODEL 为 3

兼容模式是普通模式和 PATHINFO 模式的结合，并且可以让应用在需要的时候直接切换到 PATHINFO 模式而不需要更改模板和程序。**兼容模式 URL 可以支持任何的运行环境。**

兼容模式的效果是：

http://<serverName>/appName/?s=/module/action/id/1/

并且也可以支持参数分割符号的定义，例如在 URL_PATHINFO_DEPR 为~的情况下，下面的 URL 有效：

http://<serverName>/appName/?s=module~action~id~1

其实是利用了 VAR_PATHINFO 参数，用普通模式的实现模拟了 PATHINFO 的模式。但是兼容模式并不需要自己传 s 变量，而是由系统自动完成 URL 部分。正是由于这个特性，兼容模式可以和 PATHINFO 模式之间直接切换，而不需更改模板文件里面的 URL 地址连接。

某些服务器环境不能良好的支持 PATHINFO，或者需要进行额外的配置才可以支持，如果你确认你的服务器环境不支持 PATHINFO，可以选择普通模式或者兼容模式 URL 运行。

5.2.5 URL 路由

ThinkPHP 支持 URL 路由功能，要启用路由功能，需要设置 URL_ROUTER_ON 参数为 true。开启路由功能后，系统会自动进行路由检测，如果在路由定义里面找到和当前 URL 匹配的路由名称，就会进行

路由解析和重定向。路由功能需要定义路由定义文件，位于项目的配置目录下面，文件名为 **routes.php**，定义格式：

```
return array(
  // 第一种方式 常规路由
  'RouteName'=>array('模块名称', '操作名称', '参数定义', '额外参数'),
  // 第二种方式 泛路由
  'RouteName@'=>array(
  array('路由匹配正则', '模块名称', '操作名称', '参数定义', '额外参数'),
  ),
  ...更多的路由名称定义
)
```

系统在执行 Dispatch 解析的时候，会判断当前 URL 是否存在定义的路由名称，如果有就会按照定义的路由规则来进行 URL 解析。例如，我们启用了路由功能，并且定义了下面的一个路由规则：

```
'blog'=>array('Blog', 'archive', 'year,month,day', 'userId=1&status=1')
```

那么我们在执行

http://<serverName>/appName/blog/2009/10/1/ 的时候就会实际执行 Blog 模块的 archive 操作，后面的参数/2009/10/1/ 就会依次按照 year/month/day 来解析，并且会隐含传入 userId=1 和 status=1 两个参数。

另外一种路由参数的传入是

http://<serverName>/appName/?r=blog&year=2009&month=10&day=1，会执行上面相同的路由解析，该方式主要是提供不支持 PATHINFO 方式下面的兼容实现。其中 r 由 VAR_ROUTER 参数定义，默认是 r。

如果需要路由到分组模块的话，可以定义成

'blog'=>array('**Home.Blog**', 'archive', 'year,month,day', 'userId=1&status=1')

就可以指定路由到 Home 分组的 Blog 模块。

泛路由支持

泛路由指的是对同一个路由名称提供了多个规则的支持，使得 URL 的设置更加灵活，例如，我们对 Blog 路由名称需要有多个规则的路由：

```
'Blog@'=>array(
        array('/^\/(\d+)(\/p\/\d)?$/','Blog','read','id'),
        array('/^\/(\d+)\/(\d+)/','Blog','archive','year,month'),
        ),
```

第一个路由规则表示解析 Blog/123 这样的 URL 到 Blog 模块的 read 操作

第二个路由规则表示解析 Blog/2009/10 这样的 URL 到 Blog 模块的 archive 操作

泛路由的定义难度就在路由正则的定义上面，其它参数和常规路由的使用一致。

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

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

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

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

接下来我们来定义路由文件，实现类似于

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

这样的解析，路由文件名称是

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

这样，URL 里面所有的 City 模块都会被路由到 City 模块的 city 操作，而后面的第二个参数会被解析成 $_GET['name']

接下来，我们就可以在浏览器里面输入

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

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

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

会看到依次输出的结果是：

当前城市:beijing

当前城市:shanghai

当前城市:shenzhen

5.2.6 URL 伪静态

系统支持伪静态 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 为 .shtml 的话，我们可以把下面的 URL

http://<serverName>/Blog/read/id/1

变成

http://<serverName>/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。注意配置设置时要包含后缀中的“.”。

伪静态设置后，如果需要动态生成一致的 URL，可以使用 U 方法在模板文件里面生成 URL。

关于 U 方法的使用请参考后面的 URL 生成部分。

5.2.7 URL 重写

通常的 URL 里面含有 index.php，为了达到更好的 SEO 效果可能需要去掉 URL 里面的 index.php ，通过 URL 重写的方式可以达到这种效果，通常需要服务器开启 URL_REWRITE 模块才能支持。

下面是 Apache 的配置过程，可以参考下：

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、确保 URL_MODEL 设置为 2

4、把.htaccess 文件放到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>

5.2.8 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。

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（'[项目://][路由@][分组名-模块/]操作? 参数 1=值 1[&参数 N=值 N]'）

或者用数组的方式传入参数

U（'[项目://][路由@][分组名-模块/]操作',array('参数 1'=>'值 1' [,'参数 N'=>'值 N'])）

如果不定义项目和模块的话 就表示当前项目和模块名称，下面是一些简单的例子：

U（'Myapp://User/add'）// 生成 Myapp 项目的 User 模块的 add 操作的 URL 地址

U（'Blog/read?id=1'）// 生成 Blog 模块的 read 操作 并且 id 为 1 的 URL 地址

U（'Admin-User/select'）// 生成 Admin 分组的 User 模块的 select 操作的 URL 地址

参数请确保使用 ?id=1&name=tp 或者数组的方式来定义，虽然有些情况下

U（'Blog/read/id/1'）和 U（'Blog/read?id=1'）的效果一样，但是在不同的 URL 设置情况下，会导致解析的错误。

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/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>/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>/Blog/read/id/1.html

U 方法还可以支持路由，如果我们定义了一个名称为 View 的路由，指向 Blog 模块的 read 操作，参数是 id，那么 U（`'View@?id=1'`）生成的 URL 地址是：

http://<serverName>/index.php/View/id/1

5.2.9 URL 大小写

我们知道，系统默认的规范是根据 URL 里面的 moduleName 和 actionName 来定位到具体的模块类，从而执行模块类的操作方法，如果在 Linux 环境下面，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，我们访问下面的 URL 是正常的：

http://<serverName>/index.php/User/add

但是，如果使用

http://<serverName>/index.php/user/add

就会出现 user 模块不存在的错误。因为，我们定义的模块类是 UserAction 而不是 userAction，但是后者显然不符合 ThinkPHP 的命名规范，显然这样的问题会造成用户体验的下降。

其实，系统本身已经提供了一个很好的解决方案，可以通过配置简单实现。

只要在项目配置中，增加：

'URL_CASE_INSENSITIVE' => true

就可以实现 URL 访问不再区分大小写了。

http://<serverName>/index.php/User/add

将等效于

http://<serverName>/index.php/user/add

这里需要注意一个地方，如果我们定义了一个 UserTypeAction 的模块类，那么 URL 的访问应该是：

http://<serverName>/index.php/user_type/list

而不是

http://<serverName>/index.php/usertype/list

如果设置

'URL_CASE_INSENSITIVE' => false

的话，URL 就又变成：

http://<serverName>/index.php/UserType/list

5.2.10 空操作

空操作是指系统在找不到指定的操作方法的时候，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，我们前面用 URL 路由实现了一个城市切换的功能，下面我们用空操作功能来重新实现。

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

```
Class CityAction extends Action{
    Public function _empty(){
        // 把所有城市的操作都解析到 city 方法
        $cityName = ACTION_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/City/beijing/

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

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

会看到依次输出的结果是：

当前城市:beijing

当前城市:shanghai

当前城市:shenzhen

可以看出来，和用 URL 路由实现的效果是一样的，而且不需要定义路由定义文件。

5.2.11 空模块

空模块的概念是指当系统找不到指定的模块名称的时候，系统会尝试定位空模块(EmptyAction)，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/City/shanghai/

变成

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

这样更加简单的方式，如果按照传统的模式，我们必须给每个城市定义一个 Action 类，然后在每个 Action 类的 index 方法里面进行处理。 可是如果使用空模块功能，这个问题就可以迎刃而解了。 我们可以给项目定义一个 EmptyAction 类

```
Class EmptyAction extends Action{
    Public function index(){
        // 根据当前模块名称来判断要执行哪个城市的操作
        $cityName = MODULE_NAME;
        $this->city($cityName);
    }
    Protected function city($name){
        // 和$name 这个城市相关的处理
        Echo ('当前城市: '.$name);
    }
}
```

接下来，我们就可以在浏览器里面输入

http://<serverName>/index.php/beijing/

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

http://<serverName>/index.php/shenzhen/

会看到依次输出的结果是：

当前城市:beijing

当前城市:shanghai

当前城市:shenzhen

5.2.12 前置和后置操作

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，例如，我们在UserAction 类里面定义了_before_insert() 和 _after_insert() 操作，那么执行 User 模块的 insert 操作的时候，会按照顺序执行下面的操作：

_before_insert

insert

_after_insert

特殊情况是，当前的 add 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_before_add 和 _after_add 方法的话，依然会生效，也会按照这个顺序来执行 add 操作。真正有模板输出的可能仅仅是当前的 add 操作，前置和后置操作一般情况是没有任何输出的。前置和后置操作的方法名是在要执行的方法前面加 **_before_**和**_after_**，例如：

```
Class CityAction extends Action{
    public function _before_index() {
        echo 'before';
    }
```

```
    public function index(){
        echo 'index';
    }
    public function _after_index() {
        echo 'after';
    }
}
```

执行结果会先输出 before 然后输出 index 最后输出 after。对于任何操作方法我们都可以按照这样的规则来定义前置和后置方法。

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 after 后置方法了。

5.2.13 操作链

ThinkPHP 支持使用操作链的方式，例如，我们访问下面的 URL：

http://serverName/appName/User/action1:action2:action3/

那么会依次执行 UserAction 的 action1 action2 action3 方法，并且当前操作名称是最后一个操作。在进行默认模板输出的时候会用到。如果确实需要在不同的操作方法中都进行输出，请确保在 Action 的 display 方法中指定需要渲染的模板文件名。否则，只能输出最后的操作模板。使用了操作链后，前置和后置方法会失效。

5.2.14 跨模块调用

在开发过程中经常会在当前模块调用其他模块的方法，这个时候就涉及到跨模块调用，我们还可以了解到 A 和 R 两个快捷方法的使用。

```
$User = A("User"); // 实例化 UserAction 控制器对象
$User->importUser(); // 调用 User 模块的 importUser 操作方法
```

这里的 A("User") 是一个快捷方法，和下面的代码等效：

```
import("@.Action.UserAction");
$User = new UserAction();
```

事实上，在这个例子里面还有比 A 方法更简单的调用方法，例如：

```
R("User","importUser"); // 远程调用 UserAction 控制器的 importUser 操作方法
```

上面只是在当前项目中调用，如果你有需要在多个项目之间调用方法，一样可以完成：

```
$User = A("User","App2"); // 实例化 App2 项目的 UserAction 控制器对象
$User->importUser();
// 远程调用 App2 项目的 UserAction 控制器的 importUser 操作方法
R("User","importUser","App2");
```

5.2.15 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 Action 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 ($result){
// 设置成功后的跳转页面地址 默认的返回页面是$_SERVER["HTTP_REFERER"]
```

```
$this->assign("jumpUrl","/User/list/");
    $this->success("新增成功！");
}else{
// 错误页面的默认跳转页面是返回上一页 通常可以不用设置
    $this->error("新增错误！");
}
```

Success 和 error 方法都有对应的模板，并且是可以设置的，默认的设置 Public:success 和 Public:error，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：

$msgTitle ：操作标题

$message ：页面提示信息

$status ：操作状态 1 表示成功 0 表示失败 具体还可以由项目本身定义规则

$waitSecond ：跳转等待时间 单位为妙

$jumpUrl ：跳转页面地址

如果是 AJAX 方式提交的话，success 和 error 方法会调用 ajaxReturn 方法返回信息，具体可以参考后面的 AJAX 返回部分。

5.2.16 重定向

Action 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（参考上面的 URL 生成部分），例如：

```
$this->redirect('User/list', array('cate_id'=>2), 5,'页面跳转中~')
```

上面的用法是停留 5 秒后跳转到 User 模块的 list 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。

5.2.17 AJAX 返回

系统支持任何的 AJAX 类库，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、XML 和 EVAL 三种方式给客户端接受数据，通过配置 **DEFAULT_AJAX_RETURN** 进行设置，在选择不同的 AJAX 类库的时候可以使用不同的方式返回数据。

要使用 ThinkPHP 的 ajaxReturn 方法返回数据的话，需要遵守一定的返回数据的格式规范。ThinkPHP 返回的数据格式包括：

status 操作状态

info 提示信息

data 返回数据

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 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，并且忽略 status 和 info 信息。

下面是一个简单的例子：

```
$User = M("User"); // 实例化 User 对象
$result = $User->add($data);
if ($result){
// 成功后返回客户端新增的用户 ID，并返回提示信息和操作状态
    $this->ajaxReturn($result,"新增成功！",1);
```

```
}else{
// 错误后返回错误的操作状态和提示信息
    $this->ajaxReturn(0,"新增错误！",0);
}
```

注意，确保你是使用 AJAX 提交才使用 ajaxReturn 方法。

在客户端接受数据的时候，根据使用的编码格式进行解析即可。

5.3 模型

5.3.1 定义和实例化

在 ThinkPHP2.0 版本中，**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，下面来分析下什么情况下用什么方法：

1、实例化基础模型（Model）类

在没有定义任何模型的时候，我们可以使用下面的方法实例化一个模型类来进行操作：

```
$User = new Model('User');
```

或者使用 M 快捷方法实例化是等效的

```
$User = M('User');
$User->select(); // 进行其他的数据操作
```

这种方法最简单高效，因为不需要定义任何的模型类，所以支持跨项目调用。缺点也是因为没有自定义的模型类，因此无法写入相关的业务逻辑，只能完成基本的 CURD 操作。

2、实例化其他模型类

第一种方式实例化因为没有模型类的定义，因此很难封装一些额外的逻辑方法，不过大多数情况下，也许只是需要扩展一些通用的逻辑，那么就可以尝试下面一种方法。

M 方法默认是实例化 Model 类，如果需要实例化其他模型类，可以使用

```
$User = M('User', 'CommonModel');
```

上面的方法等效于

```
$User = new CommonModel('User');
```

因为系统的模型类都能够自动加载，因此我们不需要在实例化之前手动进行类库导入操作。模型类 commonModel 必须继承 Model，如果没有定义别名导入的话，需要放在项目 Model 下。我们可以在 CommonModel 类里面定义一些通用的逻辑方法，就可以省去为每个数据表定义具体的模型类，如果你的项目已经有超过 100 个数据表了，而大多数情况都是一些基本的 CURD 操作的话，只是个别模型有一些复杂的业务逻辑需要封装，那么第一种方式和第二种方式的结合是一个不错的选择。

3、实例化用户定义的模型（×××Model）类

这种情况是使用的最多的，一个项目不可避免的需要定义自身的业务逻辑实现，就需要针对每个数据表定义一个模型类，例如 UserModel 、InfoModel 等等。

定义的模型类通常都是放到项目的 Lib\Model 目录下面。例如，

```
class UserModel extends Model{
    Public function myfun(){
    // 添加自己的业务逻辑
    // ……
```

```
    }
}
```

其实模型类还可以继承一个用户自定义的公共模型类，而不是只能继承 Model 类。

要实例化自定义模型类，可以使用下面的方式：

```
$User = new UserModel();
```

或者使用 D 快捷方法实例化是等效的

```
$User = D('User');
$User->select(); // 进行其他的数据操作
```

D 方法可以自动检测模型类，不存在时系统会抛出异常，同时对于已实例化过的模型，不会重复去实例化。默认的 D 方法只能支持调用当前项目的模型，如果需要跨项目调用，需要使用：

```
$User = D('User', 'Admin'); // 实例化 Admin 项目下面的 User 模型
$User->select();
```

如果启用了模块分组功能，可使用：

```
$User = D('Admin.User');
```

4、实例化空模型类

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

```
$Model = new Model();
// 或者使用 M 快捷方法实例化是等效的
// $Model = M();
$Model->query('SELECT * FROM think_user where status=1');
```

空模型类也支持跨项目调用。

在后面的内容中，针对 M 方法或者 D 方法将不再具体说明，请自行分析。

5.3.2 模型命名

当我们创建一个 UserModel 类的时候，其实已经遵循了系统的约定。ThinkPHP 要求数据库的表名和模型类的命名遵循一定的规范，首先数据库的表名和字段全部采用小写形式，模型类的命名规则是除去表前缀的数据表名称，并且首字母大写，然后加上模型类的后缀定义，例如：

UserModel 表示 User 数据对象，（假设数据库的前缀定义是 think_）其对应的数据表应该是 think_user

UserTypeModel 对应的数据表是 think_user_type

如果你的规则和系统的约定不符合，那么需要设置 Model 类的 tableName 属性。

在 ThinkPHP 的模型里面，有两个数据表名称的定义：

1、**tableName** 不包含表前后缀的数据表名称，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。

2、**trueTableName** 包含前后缀的数据表名称，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。

下面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_）。

```
protected $tableName = 'categories';
```

注意这个属性的定义不需要加表的前缀 think_

而对于另外一种特殊情况，数据库中有一个表（top_depts）的前缀和其它表前缀不同，不是 think_ 而是 top_，这个时候我们就需要定义 trueTableName 属性了

```
protected $trueTableName = 'top_depts';
```

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：

dbName 定义模型当前对应的数据库名称，只有当你当前的模型类对应的数据库名称和配置文件不同的时候才需要定义，例如：

```
protected $dbName = 'top';
```

另外，我们来了解下表后缀的含义。表后缀通常情况下用处不大，因为这个和表的设计有关。但是个别情况下也是有用，例如，我们在定义数据表的时候统一采用复数形式定义，下面是我们设计的几个表名 think_users、think_categories、think_blogs，我们定义的模型类分别是 UserModel 、CategoryModel 、BlogModel，按照上面的方式，我们必须给每个模型类定义 tableName 属性。其实我们可以通过设置表后缀的方式来实现相同的效果，我们可以设置 DB_SUFFIX 配置参数为 s，那么系统在获取真实的表名的时候就会自动加上这个定义的表后缀，我们就不必给每个模型类定义 tableName 属性了，而只是对 categories 这样的复数情况单独定义 trueTableName 属性就可以了。

5.3.3 获取字段

我们在 UserModel 类里面根本没有定义任何 User 表的字段信息，但是系统是如何做到属性对应数据表的字段呢？这是因为 ThinkPHP 可以在运行时自动获取数据表的字段信息（确切的说，是在第一次运行的时候，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），包括数据表的主键字段和是否自动增长等等，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。如果你在开发过程中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。

如果你没有定义模型类，进行相关操作的时候一样会生成字段缓存文件。

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，在模型类里面添加 fields 属性即可，定义格式如下：

```
class UserModel extends Model{
    protected $fields = array(
    'id',
    'username',
    'email',
    'age',
    '_pk'=>'id',
    '_autoinc'=>true
    )
}
```

其中_pk 表示主键字段名称 _autoinc 表示主键是否自动增长类型

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：

```
'DB_FIELDS_CACHE'=>false
```

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。ThinkPHP 的默认约定每个数据表的主键名采用统一的 id 作为标识，并且是自动增长类型的。系统会自动识别当前操作的数据表的字段信息和主键名称，所以即使你的主键不是 id，也无需进行额外的设置，系统会自动识别。要在外部获取当前数据对象的主键名称，请使用下面的方法：

```
$pk = $Model->getPk();
```

目前不支持联合主键的自动操作。

在个别情况下，可能不需要对当前操作的数据表进行字段缓存，或许是由于采用了动态方式或者当前模型根本没有任何相关的数据表，我们可以设置 **autoCheckFields** 属性来关闭某个模型类的字段获取和缓存。

使用 getDbFields 方法可以获取当前数据对象的全部字段信息：

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

5.3.4 属性访问

因为 Model 对象本身也是一个数据对象，所以属性的访问就显得非常直观和简单。

ThinkPHP 利用了 PHP5 的魔术方法机制来实现了属性的直观访问。这也是最常用的访问方式，通过数据对象访问，例如

```
$User = new Model('User');
$User->find(1);
```

```
// 获取 name 属性的值
echo $User->name;
// 设置 name 属性的值
$User->name = 'ThinkPHP'
```

还有一种属性的操作方式是通过返回数组的方式：

```
$User = D("User");
// 注意这里返回的 user 数据是一个数组
$user = $User->find(1);
// 获取 name 属性的值
echo $user['name'];
// 设置 name 属性的值
$user['name'] = 'ThinkPHP';
```

两种方式的属性区别是一个是对象的属性，一个是数组的索引名称。

5.3.5 跨库操作

ThinkPHP 可以支持模型的同一数据库服务器的跨库操作，跨库操作只需要简单配置一个模型所在的数据库名称即可，例如，假设 UserModel 对应的数据表在数据库 user 下面，而 InfoModel 对应的数据表在数据库 info 下面，那么我们只需要进行下面的设置即可。

```
class UserModel extends Model {
    protected $dbName = 'user';
}
class InfoModel extends Model {
    protected $dbName = 'info';
```

```
}
```

在进行查询的时候，系统能够自动添加当前模型所在的数据库名。

```
$User = D('User');
$User->select();
echo $User->getLastSql();
// 输出的 SQL 语句为 select * from user.think_user
```

模型的表前缀取的是项目配置文件定义的数据表前缀，如果跨库操作的时候表前缀不是统一的，那么我们可以在模型里面单独定义表前缀，例如：

```
protected $tablePrefix = 'other_';
```

5.3.6 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适配器来处理。目前的数据库包括 Mysql、MsSQL、PgSQL、Sqlite、Oracle、Ibase 以及 PDO 的支持，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：

第一种 在项目配置文件里面定义

```
return array(
  'DB_TYPE'=> 'mysql',
  'DB_HOST'=> 'localhost',
  'DB_NAME'=>'thinkphp',
  'DB_USER'=>'root',
  'DB_PWD'=>'',
  'DB_PORT'=>'3306',
```

```
    'DB_PREFIX'=>'think_',
    // 其他项目配置参数.....
);
```

系统推荐使用该种方式，因为一般一个项目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。

可以对每个项目定义不同的数据库连接信息，还可以在调试配置文件里面定义调试数据库的配置信息，如果在项目配置文件和调试模式配置文件里面同时定义了数据库连接信息，那么在调试模式下面后者生效，部署模式下面前者生效。

第二种 使用 DSN 方式在初始化 Db 类的时候传参数

```
$db_dsn = "mysql://username:passwd@localhost:3306/DbName";
$db = new Db($db_dsn);
```

该方式主要用于在控制器里面自己手动连接数据库的情况，或者用于创建多个数据库连接。

第三种 使用数组传参数

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

```
);
 $db = new Db($DSN);
```

该方式也是用于手动连接数据库的情况，或者用于创建多个数据库连接。

第四种 在模型类里面定义

```
protected $connection = array(
   'dbms'     => 'mysql',
   'username' => 'username',
   'password' => 'password',
   'hostname' => 'localhost',
   'hostport' => '3306',
   'database' => 'dbname'
);
// 或者使用下面的定义
protected $connection = ”mysql://username:passwd@localhost:3306/DbName”;
```

如果在某个模型类里面定义了 connection 属性，则在实例化模型对象的时候，会使用该数据库连接信息进行数据库连接。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。

ThinkPHP 并不是在一开始就会连接数据库，而是在有数据查询操作的时候才会去连接数据库。额外的情况是，在系统第一次操作模型的时候，框架会自动连接数据库获取相关模型类的数据字段信息，并缓存下来。

ThinkPHP 支持 PDO 方式，如果要使用 PDO 方式连接数据库，可以参考下面的设置。

我们以项目配置文件定义为例来说明：

```
return array(
    'DB_TYPE'=> 'pdo',
    // 注意 DSN 的配置针对不同的数据库有所区别 请参考 PHP 手册 PDO 类库部分
    'DB_DSN'=> 'mysql:host=localhost;dbname=think',
    'DB_USER'=>'root',
    'DB_PWD'=>'',
    'DB_PREFIX'=>'think_',
    // 其他项目配置参数.....
);
```

使用 PDO 方式的时候，要注意检查是否开启相关的 PDO 模块。DB_DSN 参数仅对 PDO 方式连接才有效。

5.3.7 主从数据库

ThinkPHP 的模型支持主从式数据库的连接，配置 DB_DEPLOY_TYPE 为 1 可以采用分布式数据库支持。如果采用分布式数据库，定义数据库配置信息的方式如下：

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

```
    'DB_PREFIX'=>'think_',
    ..... 其它项目配置参数
);
```

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

'DB_PORT'=>'3306,3306' 和 'DB_PORT'=>'3306' 等效

'DB_USER'=>'user1',

'DB_PWD'=>'pwd1',

和

'DB_USER'=>'user1,user1',

'DB_PWD'=>'pwd1,pwd1',

等效。

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

'DB_RW_SEPARATE'=>true,

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

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

5.3.8 创建数据

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
.....
```

然而 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();
// 把创建的数据对象写入数据库
$User->add();
```

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 = M("User");
$User->find(1);
$Member = M("Member");
$Member->create($User);
```

而事实上，create 方法所做的工作远非这么简单，在创建数据对象的同时，完成了一些很有意义的工作，包括：

- ✧ 支持多种数据源
- ✧ 令牌验证
- ✧ 数据自动验证
- ✧ 字段映射支持
- ✧ 字段类型检查
- ✧ 数据自动完成

因此，我们熟悉的令牌验证、自动验证和自动完成（我们会在后面看到相关的用法）功能，其实都必须通过 create 方法才能生效。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。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();
```

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。

5.3.9 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 TP 的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：

```
Class UserModel extends Model{
  protected $_map = array(
    'name'        =>'username',
    'mail'        =>'email',
  );
}
```

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。在保存的时候会字段转换成定义的字段映射。

5.3.10 连贯操作

ThinkPHP2.0 版本全面启用模型类的**连贯操作**方法，可以有效的提高数据存取的代码清晰度和开发效率。使用方面也比较简单， 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();
```

除了 select 方法必须放到最后一个外，其他的**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')->where('status=1')->limit(10)->select();
```

如果不习惯使用连贯操作的话，新版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 add、 save、delete 等方法都可以使用，例如：

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

原则上说，**所有的连贯操作都只有一个参数**，并且连贯操作的参数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，简而言之，连贯操作的结果不会带入以后的查询。下面总结下连贯操作的使用方法（更多的用法我们会在 CURD 操作的过程中详细描述）：

Where 方法：用于查询或者更新条件的定义

Where 方法的参数支持字符串、数组和对象。详细的使用请参考后面的查询语言部分。

Table 方法：定义要操作的数据表名称

可以动态改变当前操作的数据表名称，需要写数据表的全名，包含前缀，可以使用别名，例如：

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

Table 方法的参数支持字符串和数组，数组方式的用法：

$Model->Table(array('think_user'=>'user','think_group'=>'group'))->where('status>1')->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。

如果不定义 table 方法，默认会自动获取当前模型对应或者定义的数据表。

Data 方法：数据对象赋值

可以用于新增或者保存数据之前的数据对象赋值，例如：

$Model->data($data)->add();

$Model->data($data)->where('id=3')->save();

Data 方法的参数支持对象和数组，如果是对象会自动转换成数组。如果不定义 data 方法赋值，也可以使用 create 方法或者手动给数据对象赋值的方式。

Field 方法：定义要查询的字段

Field 方法的参数支持字符串和数组，例如，

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

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

如果不使用 field 方法指定字段的话，默认和使用 field('*')等效。

Order 方法：结果排序

例如：order('id desc')

排序方法支持对多个字段的排序

order('status desc,id asc')

order 方法的参数支持字符串和数组，数组的用法如下：

order(array('status'=>'desc','id'))

Limit 方法：结果限制

我们知道不同的数据库类型的 limit 用法是不尽相同的，但是在 ThinkPHP 的用法里面始终是统一的方法，也就是 limit('offset,length') ，无论是 Mysql、SqlServer 还是 Oracle 数据库，都是这样使用，系统的数据库驱动类会负责解决这个差异化。

例如：

limit('1,10')

如果使用 limit('10') 等效于 limit('0,10')

Page 方法：查询分页

Page 操作方法是新增的特性，可以更加快速的进行分页查询。

Page 方法的用法和 limit 方法类似，格式为：

Page('page[,listRows]')

Page 表示当前的页数，listRows 表示每页显示的记录数。例如：

Page('2,10')

表示每页显示 10 条记录的情况下面，获取第 2 页的数据。

listRow 如果不写的话，会读取 limit('length') 的值，例如：

limit(25)->page(3);

表示每页显示 25 条记录的情况下面，获取第 3 页的数据。

如果 limit 也没有设置的话，则默认为每页显示 20 条记录。

Group 方法：查询 Group 支持

例如：group('user_id')

Group 方法的参数只支持字符串

Having 方法：查询 Having 支持

例如：having('user_id>0')

having 方法的参数只支持字符串

Join 方法：查询 Join 支持

Join 方法的参数支持字符串和数组，并且 join 方法是连贯操作中唯一可以多次调用的方法。

例如：

$Model->join(' work ON artist.id = work.artist_id')->join('card ON artist.card_id = card.id')->select();

默认采用 LEFT 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();

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 = card.id'))

Distinct 方法：查询的 Disiinct 支持

查询数据的时候进行唯一过滤

$Model->Distinct(true)->select();

Relation 方法：关联查询支持

关联查询方法的详细用法请参考后面的关联模型部分。

Lock 方法：查询锁定

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。

5.3.11 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。下面来分析下各自的用法：

（下面的 CURD 操作我们均以 M 方法创建模型实例来说明，因为不涉及到具体的业务逻辑）

一、创建操作

在 ThinkPHP 使用 add 方法新增数据到数据库。

使用方法如下：

```
$User = M("User"); // 实例化 User 对象
$data['name'] = 'ThinkPHP';
$data['email'] = 'ThinkPHP@gmail.com';
$User->add($data);
```

或者使用 data 方法连贯操作

```
$User->data($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
$User->create();
$User->add(); // 根据条件保存修改的数据
```

如果你的主键是自动增长类型，并且如果插入数据成功的话，Add 方法的返回值就是最新插入的主键值，可以直接获取。

二、**读取数据**

在 ThinkPHP 中读取数据的方式很多，通常分为读取数据和读取数据集。

读取数据集使用 findall 或者 select 方法（findall 和 select 方法等效）：

```
$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();
```

select 方法的返回值是一个二维数组，如果没有查询到任何结果的话，也是返回一个空的数组。配合上面提到的连贯操作方法可以完成复杂的数据查询。而最复杂的连贯方法应该是 where 方法的使用，因为这部分涉及的内容较多，我们会在查询语言部分就如何进行组装查询条件进行详细的使用说明。基本的查询暂时不涉及关联查询部分，而是统一采用关联模型来进行数据操作，这一部分请参考关联模型部分。

读取数据使用 find 方法：

读取数据的操作其实和数据集的类似，select 可用的所有连贯操作方法也都可以用于 find 方法，区别在于 find 方法最多只会返回一条记录，因此 limit 方法对于 find 查询操作是无效的。

```
$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="think" ')->find();
```

即使满足条件的数据不止一条，find 方法也只会返回第一条记录。

如果要读取某个字段的值，可以使用 getField 方法，例如：

```
$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');
```

当只有一个字段的时候，始终返回一个值。

如果传入多个字段的话，可以返回一个关联数组：

```
$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');
```

返回的 list 是一个数组，键名是用户的 id， 键值是用户的昵称 nickname。

三、**更新数据**

在 ThinkPHP 中使用 save 方法更新数据库，并且也支持连贯操作的使用。

```
$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); // 根据条件保存修改的数据
```

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。

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
```

上面的情况，表单中必须包含一个以主键为名称的隐藏域，才能完成保存操作。

如果只是更新个别字段的值，可以使用 setField 方法：

```
$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
$User-> where('id=5')->setField(array('name','email'),array('ThinkPHP','ThinkPHP@gmail.com'));
```

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->setInc('score','id=5',3); // 用户的积分加 3
$User->setInc('score','id=5'); // 用户的积分加 1
$User->setDec('score','id=5',5); // 用户的积分减 5
$User->setDec('score','id=5'); // 用户的积分减 1
```

四、**删除数据**

在 ThinkPHP 中使用 delete 方法删除数据库中的记录。同样可以使用连贯操作进行删除操作。

```
$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->where('status=0')->delete(); // 删除所有状态为 0 的用户数据
```

delete 方法可以用于删除单个或者多个数据，主要取决于删除条件，也就是 where 方法的参数，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();
```

5.3.12 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
```

5.3.13 令牌验证

ThinkPHP 新版内置了表单令牌验证功能，可以有效防止表单的远程提交等安全防护。

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

'TOKEN_NAME'=>'__hash__',    // 令牌验证的表单隐藏字段名称
```

```
'TOKEN_TYPE'=>'md5',  //令牌哈希验证规则 默认为 MD5
```

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。如果在开启表单令牌验证的情况下，个别表单不需要使用令牌验证功能，可以在表单页面添加{__NOTOKEN__}，则系统会忽略当前表单的令牌验证。

如果页面中存在多个表单，建议添加{__TOKEN__}标识，并确保只有一个表单需要令牌验证。

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
}
```

5.3.14 类型检测

新版的 ThinkPHP 具有字段类型检测，对于不合法的字段数据会进行强制转换。字段类型检测可以用于数据写入和数据查询操作。

需要启用字段类型检测的话，需要在配置文件中开启 DB_FIELDTYPE_CHECK 参数：

'DB_FIELDTYPE_CHECK'=>true, // 开启字段类型验证

如果在非调试模式下面开启字段类型检测后，请清空字段缓存目录（位于Runtime/Data/_fields/），重新生成字段缓存的时候，会在缓存文件中记录字段的类型信息。这是后面进行字段类型检测的前提。

字段类型检测主要在两个阶段会自动处理：

一、在数据写入到数据库之前

例如：

```
$User = M("User"); // 实例化 User 对象
// 然后直接给数据对象赋值
$User->name = 'ThinkPHP';
$User->score = '2ThinkPHP';
// 把数据对象添加到数据库
$User->add();
```

由于用户表的 score 设计的是数字类型，所以实际写入数据库之前，score 属性的值已经被强制进行 intval 转换了，模型的 save 方法也会同样进行字段类型检查。虽然在很多情况下，数据库本身也会进行数据转换，但是对于某些数据库要求严格检查数据类型的情况会有帮助。

二、在使用数组方式的普通查询条件后

例如：

```
$User = M("User"); // 实例化 User 对象
```

```
$condition['id'] = '1 OR 1=1';
// 把查询条件传入查询方法
$User->where($condition)->select();
```

对于这样的一个查询条件，在进行数据库查询之前，会对查询的数组条件进行字段类型检查，直接就把 id 的值强制转换为 1 然后再进行查询操作。

即使不进行强制转换，系统也会进行安全过滤，把这样的非法数据进行转义，区别在于这样对于数据库更加安全，对于某些数据库要求严格检查数据类型的情况会有帮助。

5.3.15 自动验证

类型检查只是针对数据库级别的验证，所以系统还内置了数据对象的自动验证功能来完成模型的业务规则验证，而大多数情况下面，数据对象是由表单提交的$_POST 数据创建。需要使用系统的自动验证功能，只需要在 Model 类里面定义**$_validate** 属性，是由多个验证因子组成的数组，支持的验证因子格式：

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

验证字段：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。（必须）

验证规则： 要进行验证的规则，需要结合附加规则（必须）

提示信息： 用于验证失败后的提示信息定义（必须）

验证条件：（可选）

✧ Model::EXISTS_TO_VAILIDATE 或者 0 存在字段就验证 （默认）

✧ Model::MUST_TO_VALIDATE 或者 1 必须验证

✧ Model::VALUE_TO_VAILIDATE 或者 2 值不为空的时候验证

附加规则： 配合验证规则使用（可选），包括：

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

✧ function 使用函数验证，前面定义的验证规则是一个函数名

✧ callback 使用方法验证，前面定义的验证规则是当前 Model 类的一个方法

✧ confirm 验证表单中的两个字段是否相同，前面定义的验证规则是一个字段名

✧ equal 验证是否等于某个值，该值由前面的验证规则定义

✧ in 验证是否在某个范围内，前面定义的验证规则必须是一个数组

✧ unique 验证是否唯一，系统会根据字段目前的值查询数据库来判断是否存在相同的值

系统还内置了一些常用正则验证的规则，可以直接使用，包括：require 字段必须、email 邮箱、url URL 地址、currency 货币、number 数字，这些验证规则可以直接使用。

验证时间：（可选）

Model:: MODEL_INSERT 或者 1 新增数据时候验证

Model:: MODEL_UPDATE 或者 2 编辑数据时候验证

Model:: MODEL_BOTH 或者 3 全部情况下验证（默认）

示例：

```
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
);
```

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

```
$User = D("User"); // 实例化 User 对象
if (!$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
}
```

通常来说，每个数据表对应的验证规则是相对固定的，但是有些特殊的情况下面可能会改变验证规则，我们可以动态的改变验证规则来满足不同条件下面的验证：

```
$User = D("User"); // 实例化 User 对象
$validate    =    array(
    array('verify','require','验证码必须！'), // 仅仅需要进行验证码的验证
);
$User-> setProperty("_validate",$validate);
$result = $User->create();
if (!$result){
   // 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
}
```

5.3.16 自动完成

在 Model 类定义 $_auto 属性，可以完成数据自动处理功能，用来处理默认值、数据过滤以及其他系统写入字段。$_auto 属性是由多个填充因子组成的数组，填充因子定义格式：

array(**填充字段,填充内容,填充条件,附加规则**)

填充字段就是需要进行处理的表单字段，这个字段不一定是数据库字段，也可以是表单的一些辅助字段，例如确认密码和验证码等等。

填充条件包括：

✧ Model:: MODEL_INSERT 或者 1 新增数据的时候处理（默认）

✧ Model:: MODEL_UPDATE 或者 2 更新数据的时候处理

✧ Model:: MODEL_BOTH 或者 3 所有情况都进行处理

附加规则包括：

✧ 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）

示例：

```
protected $_auto = array (
  array('status','1'),  // 新增的时候把 status 字段设置为 1
  array('password','md5',1,'function') // 对 password 字段在新增的时候使 md5 函数处理
  array('name','getName',1,'callback') // 对 name 字段在新增的时候回调 getName 方法
  array('create_time','time',2,'function'), // 对 create_time 字段在更新的时候写入当前时间戳
);
```

使用自动填充可能会覆盖表单提交项目。其目的是为了防止表单非法提交字段。使用 Model 类的 create 方法创建数据对象的时候会自动进行表单数据处理。

和自动验证一样，自动完成机制需要使用 create 方法才能生效。并且，也可以在操作方法中动态的更改自动完成的规则。

```
 $auto = array (
  array('password','md5',1,'function') // 对 password 字段在新增的时候使 md5 函数处理
);
$User-> setProperty("_auto",$auto);
$User->create();
```

5.3.17 查询语言

ThinkPHP 可以支持直接使用字符串作为查询条件，但是大多数情况推荐使用索引数组或者对象来作为查询条件，因为会更加安全。查询条件可以用于 CURD 等任何操作，作为 where 方法的参数传入即可，ThinkPHP 内置了非常灵活的查询方法，可以快速的进行数据查询操作，下面来一一讲解查询语言的内涵。

5.3.17.1 普通查询

除了字符串查询条件外，数组和对象方式的查询条件是非常常用的，这些是基本查询所必须掌握的。

一、使用数组作为查询条件

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

二、使用对象方式来查询 可以使用任何对象 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';  // 查询 name 的值为 thinkphp 的记录
$User->where($condition)->select();
//  上面的查询条件等同于 where('name="thinkphp"')
```

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。

三、使用查询表达式

上面的查询条件仅仅是一个相等的判断，可以使用查询表达式支持更多的 SQL 语法，并且可以用于数组或者对象方式的查询（下面仅以数组方式为例说明），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：

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

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 name like '%thinkphp%'

[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
```

5.3.17.2 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')
```

5.3.17.3 组合查询

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，例如下面的查询条件：

```
$User = M("User"); // 实例化 User 对象
$map['id'] = array('neq',1);
$map['name'] = 'ok';
$User->where($map)->select();
```

得到的查询条件是：(`id` != 1) AND (`name` = 'ok')

如果添加了下面的查询条件：

```
$map['_logic'] = 'or';
```

现在的查询条件就变为： (`id` != 1) OR (`name` = 'ok')

数组条件还可以和字符串条件混合使用，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 )
```

新版还可以支持一种特殊的条件查询，前提是简单的条件相等判断。

```
$map['_query'] = 'status=1&score=100&_logic=or';
```

得到的查询条件是：`status` = '1' OR `score` = '100'

5.3.17.4 复合查询

新版完善了复合查询，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

相同的查询条件有多种表达形式，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。

5.3.17.5 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。

```
$User = M("User"); // 实例化 User 对象
```

获取用户数：

```
$userCount = $User->count();
```

获取用户的最大积分：

```
$maxScore = $User->max('score');
```

获取积分大于 0 的用户的最小积分：

```
$minScore = $User->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。

5.3.17.6 定位查询

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

获取符合条件的第 3 条记录：

```
$User->where('score>0')->order('score desc')->getN(2);
```

获取符合条件的最后第二条记录：

```
$User-> where('score>80')->order('score desc')->getN(-2);
```

获取第一条记录：

```
$User->where('score>80')->order('score desc')->first();
```

获取最后一条记录：

```
$User->where('score>80')->order('score desc')->last();
```

5.3.17.7 SQL 查询

ThinkPHP 内置的 ORM 和 ActiveRecord 模式实现了方便的数据存取操作，而且新版增加的连贯操作功能更是让这个数据操作更加清晰，但是 ThinkPHP 仍然保留了原生的 SQL 查询和执行操作支持，为了满足复杂查询的需要和一些特殊的数据操作，SQL 查询的返回值因为是直接返回的 Db 类的查询结果，没有做任何的处理。而且可以支持查询缓存。主要包括下面两个方法：

1、query 方法

query 方法是用于 sql 查询操作，和 select 一样返回数据集，例如：

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

2、execute 方法

用于更新和写入数据的 sql 操作，返回影响的记录数，例如：

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

关于原生 SQL 操作的一点补充

通常使用原生 SQL 需要手动加上当前要查询的表名，如果你的表名以后会变化的话，那么就需要修改每个原生 SQL 查询的 sql 语句了，针对这个情况，TP 还提供了一个小的技巧来帮助解决这个问题。

例如：

```
$model = M("User");
$model->query('select * from __TABLE__ where status>1');
```

我们这里使用了__TABLE__ 这样一个字符串，系统在解析的时候会自动替换成当前模型对应的表名，这样就可以做到即使模型对应的表名有所变化，仍然不用修改原生的 sql 语句。

5.3.17.8 动态查询

借助 PHP5 语言的特性，ThinkPHP 实现了动态查询。该查询方式针对数据表的字段进行查询。例如，User 对象拥有 id,name,email,address 等属性，那么我们就可以使用下面的查询方法来直接根据某个属性来查询符合条件的记录。

```
$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。ThinkPHP 还提供了另外一种动态查询方式，就是获取符合条件的前 N 条记录。例如，我们需要获取当前用户中积分大于 0，积分最高的前 5 位用户 ：

```
$User-> where('score>80')->order('score desc')->top5();
```

要获取积分的前 8 位可以改成：

```
$User-> where('score>80')->order('score desc')->top8();
```

5.3.18 查询锁定

ThinkPHP 支持查询或者更新的锁定，只需要在查询或者更新之前使用 lock 方法即可。

查询锁定使用：

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

更新锁定使用：

```
$list = $User->lock(true)->where('status=1')->data($data)->save();
```

5.3.19 事务支持

ThinkPHP 提供了单数据库的事务支持，如果要在应用逻辑中使用事务，可以参考下面的方法：

启动事务：

```
$User->startTrans()
```

提交事务：

```
$User->commit()
```

事务回滚：

```
$User->rollback()
```

事务是针对数据库本身的，所以可以跨模型操作的 。

例如：

```
// 在 User 模型中启动事务
$User->startTrans()
// 进行相关的业务逻辑操作
$Info = M("Info"); // 实例化 Info 对象
$Info->save($User); // 保存用户信息
if (操作成功){
    // 提交事务
    $User->commit()
}else{
    // 事务回滚
    $User->rollback()
}
```

5.3.20 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 AdvModel 类或者采用动态模型。

```
class UserModel extends AdvModel{}
```

我们下面的示例都假设 UserModel 类继承自 AdvModel 类。

5.3.20.1 字段过滤

基础模型类内置有数据自动完成功能，可以对字段进行过滤，但是必须通过 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。

5.3.20.2 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;
```

5.3.20.3 文本字段

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

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

```
Protected  $blobFields     =     array('content');
```

系统在查询和写入数据库的时候会自动检测文本字段，并且支持多个字段的定义。

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

5.3.20.4 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。

5.3.20.5 多数据库连接和切换

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

多数据库支持

如果你的应用需要在特殊的时候连接多个数据库，那么可以尝试使用 ThinkPHP 的多数据库连接特性：包括相同类型的数据库和不同类型的数据库。

我们首先需要在模型类里面增加需要的数据库连接，例如：

我们在 UserModel 类增加多个数据库连接，首先定义额外的数据库连接信息

```
$myConnect1 = array(
'dbms'     => 'mysql',
'username' => 'username',
'password' => 'password',
```

```
'hostname' => 'localhost',
'hostport' => '3306',
'database' => 'dbname'
);
```

或者使用下面的定义

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

定义之后就可以进行动态的增加和切换数据库了。

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

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

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

switchConnect 方法会智能识别该连接是否是相同类型的连接。

如果要切换的数据表名称和当前模型的不一致，可以传入参数：

```
$User->switchConnect(2, 'Member');
```

这样连接新的数据库后切换到的数据表就成了 member 表了，当然前缀还是一样的。

我们还可以使用 addConnect 方法添加多个动态数据库连接，只要传入数组参数就可以了，例如：

```
$myConnect[1] = 'mysql://username:passwd@192.168.1.1:3306/DbName1';
$myConnect[2] = 'mysqli://username:passwd@192.168.1.2:3306/DbName2';
$myConnect[3] = 'mysql://username:passwd@192.168.1.3:3306/DbName3';
$User->addConnect($myConnect);
```

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

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

可以在使用之后关闭添加的连接，可以使用 closeConnect 方法，例如：

```
// 关闭连接序号为 3 的数据库连接
$User->closeConnect(3);
```

5.3.20.6 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。

5.3.20.7 延迟更新

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

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

我们先来看递增的情况，如果我们需要给会员累积积分，可以使用

```
$User = D("User"); // 实例化 User 对象
// 把 id 为 5 的用户的积分加 10
$User->setInc("score","id=5",10);
$User->setInc("score","id=5",30);
```

上面的操作更新了两次用户积分，并且都实时保存到数据库

如果我们使用延迟更新方法，例如下面对用户的积分延迟更新 60 秒

```
$User->setLazyInc("score","id=5",10,60);
$User->setLazyInc("score","id=5",30,60);
$User->setLazyInc("score","id=5",10,60);
```

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

```
$User->setInc("score","id=5",50);
```

效果是等效。区别在于用户数据库中的积分不是实时的。

同样，还可以使用 setLazyDec 进行延迟更新操作。

5.3.20.8 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。

5.3.20.9 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
    }
}
```

5.3.21 视图模型

5.3.21.1 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 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）。

```
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。如果希望给 blog 表定义另外的别名，可以使用

'_as'=>'myBlog'

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

'Category'=>array('title'=>'category_name')

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

'title'=>'category_name'

把 Category 模型的 title 字段映射为 category_name 字段，如果有多个字段，可以使用同样的方式添加。可以通过_on 来给视图模型定义关联查询条件，例如：

'_on'=>'Blog.category_id=Category.id'

理解之后，User 模型的定义方式同样也就很容易理解了。

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

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

Select

Blog.id as id,

Blog.name as name,

Blog.title as title,

Category.title as category_name,

User.name as username

from think_blog Blog JOIN think_category Category JOIN think_user User

where Blog.category_id=Category.id AND Blog.user_id=User.id

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

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

```
public $viewFields = array(
 'Blog'=>array('id','name','title','_type'=>'LEFT'),
 'Category'=>array('title'=>'category_name','_on'=>'Category.id=Blog.category_id','_type'=>'RIGHT'),
 'User'=>array('name'=>'username','_on'=>'User.id=Blog.user_id'),
);
```

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

'_type'=>'LEFT'

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

Select

Blog.id as id,

Blog.name as name,

Blog.title as title,

Category.title as category_name,

User.name as username

from think_blog Blog LEFT JOIN think_category Category ON Blog.category_id=Category.id RIGHT JOIN think_user User ON Blog.user_id=User.id

我们可以在试图模型里面定义特殊的字段，例如下面的例子定义了一个统计字段

'Category'=>array('title'=>'category_name','COUNT(Blog.id)'=>'count','_on'=>'Category.id=Blog.category_id'),

5.3.21.2 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,useruame')->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,categoryName,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。

5.3.22 关联模型

5.3.22.1 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。

5.3.22.2 关联定义

ThinkPHP 可以很轻松的完成数据表的关联 CURD 操作，目前支持的关联关系包括下面四种：HAS_ONE、BELONGS_TO、HAS_MANY、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。关联定义的格式是：

```
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',
```

```
    ),
    ...
);
```

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：

```
class UserModel extends Model
{
    public $_link = array(
        'Profile'=> 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：

```
class UserModel extends Model
{
    public $_link = array(
       'Profile'=>array(
            'mapping_type' =>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'=> BELONGS_TO
```

完整方式定义为：

```
'Dept'=> array(
            'mapping_type'=>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：

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'=> HAS_MANY
```

完整定义方式为：

```
'Article'=> array(
        'mapping_type'=>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：

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

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 。

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

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'=> MANY_TO_MANY
```

完整定义方式为：

```
array(    'mapping_type'=>MANY_TO_MANY,
          'class_name'=>'Group',
          'mapping_name'=>'groups',
          'foreign_key'=>'userId',
          'relation_foreign_key'=>'goupId',
          'relation_table'=>'think_gourpUser')
```

MANY_TO_MANY 支持的关联属性定义有：

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

外键的默认规则是当前数据**对象名称_id**，例如：

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。

5.3.22.3 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->realtion(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凡事定义了 as_fields 属性的话，

```
    protected $_link = array(
      'profile'=>array(
          'mapping_type' =>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。

5.3.22.4 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 =  $User->realtion(true)->add($user);
```

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 =  $User->realtion("Profile")->add($user);
```

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 =  $User-> relation(True)->where('id=3')->save($data);
```

Realtion(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");
```

5.3.23 动态模型

新版的模型可以在不同的类型之间切换，例如你可以从基本模型切换到高级模型或者视图模型，而当前的数据不会丢失，并可以控制要传递的参数和动态赋值。

要切换模型，可以使用：

```
$User = M("User"); // 实例化 User 对象 是基础模型类的实例
// 动态切换到高级模型类 执行 top10 查询操作
$User->switchModel("Adv")->top10();
```

如果要传递参数，可以使用：

```
$User = D("User"); // 实例化 User 对象 是基础模型类的实例
// 动态切换到视图模型类 并传入 viewFields 属性
$UserView = $User->switchModel("View",array("viewFields"));
```

如果要动态赋值，可以使用：

```
$User = M("User"); // 实例化 User 对象 是基础模型类的实例
// 动态切换到高级模型类 并传入 data 属性
$advUser = $User->switchModel("Relation");
// 或者在切换模型后再动态赋值给新的模型
$advUser->setProperty("_link",$link);
// 查找关联数据
$user = $advUser->relation(true)->find(1);
```

5.4 视图

在 ThinkPHP 里面，视图有两个部分组成：View 类和模板文件。Action 控制器直接和 View 视图类打交道，把要输出的数据通过模板变量赋值的方式传递到视图类，而具体的输出工作则交由 View 视图类来进行，同时视图类还完成了一些辅助的工作，包括调用模板引擎、布局渲染、输出替换、页面 Trace 等功能。

为了方便使用，在 Action 类中封装了 View 类的一些输出方法，例如 display、fetch、assign、trace 和 buildHtml 等方法，这些方法的原型都在 View 视图类里面。

5.4.1 模板定义

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：

模板目录/模板主题/[分组名/]模块名/操作名+模板后缀

模板目录默认是项目下面的 Tpl， 模板主题默认是 default，模板主题功能是为了多模板切换而设计的，如果有多个模板主题的话，可以用 **TMPL_DEFAULT_THEME** 参数设置默认的模板主题名。

在每个模板主题下面，是以项目的模块名为目录，然后是每个模块的具体操作模板文件，例如：

User 模块的 add 操作 对应的模板文件就应该是：Tpl/default/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。

如果项目启用了模块分组功能（假设 User 模块属于 Home 分组），那么默认对应的模板文件可能变成 ：Tpl/default/Home/User/add.html

当然，分组功能也提供了 **TMPL_FILE_DEPR** 参数来配置简化模板的目录层次。

例如 TMPL_FILE_DEPR 如果配置成“_”的话，默认的模板文件就变成了：

Tpl/default/Home/User_add.html

正是因为系统有这样一种模板文件自动识别的规则，所以通常的 display 方法无需带任何参数即可输出对应的模板。

5.4.2 模板赋值

要在模板中输出变量，必须在在 Action 类中把变量传递给模板，视图类提供了 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 ;
```

系统只会输出设定的变量，其它变量不会输出，一定程度上保证了变量的安全性。

如果要同时输出多个模板变量，可以使用下面的方式：

```
$array = array();
$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。

模板变量赋值后，怎么在模板文件中输出，需要根据选择的模板引擎来用不同的方法，如果使用的是内置的模板引擎，请参考后面的模板指南部分。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，如下：

```
<?php echo $name.'['.$email.' '.$phone.']';?>
```

5.4.3 模板输出

模板变量赋值后就需要调用模板文件来输出相关的变量，模板调用通过 display 方法来实现。我们在操作方法的最后使用：

```
$this->display();
```

根据前面的模板定义规则，因为系统会按照默认规则自动定位模板文件，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。

事情总有特例，或者根本不需要按模块进行分目录存放，不过 display 方法总是能够帮你解决问题。

Display 方法提供了几种规则让你可以随心所欲的输出需要的模板，无论你的模板文件在什么位置。

下面来看具体的用法：

一、调用当前模块的其他操作模板

格式：display('**操作名**')

例如，假设当前操作是 User 模块下面的 read 操作，我们需要调用 User 模块的 edit 操作模版，使用：

```
$this->display('edit');
```

不需要写模板文件的路径和后缀。

二、调用其他模块的操作模板

格式：display('**分组名:模块名:操作名**') 其中分组名是可选的

例如，当前是 User 模块，我们需要调用 Member 模块的 read 操作模版 ，使用：

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

如果要调用分组 Admin 的 Member 模块的 read 操作模板，可以使用：

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

这种方式也不需要写模板文件的路径和后缀，严格来说，这里面的模块名和操作名并不一定需要有对应的模块或者操作，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 模块，更没有 Public 模块的 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。

三、调用其他主题的操作模板

格式：display('**主题名@模块名:操作名**')

例如我们需要 调用 Xp 主题的 User 模块的 edit 操作模版，使用：

```
$this->display('Xp@User:edit');
```

这种方式需要指定模块和操作名

四、直接全路径输出模板

格式：display('**模板文件名**')

例如，我们直接输出当前的 Public 目录下面的 menu.html 模板文件，使用：

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

这种方式需要指定模板路径和后缀，这里的 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('./Public/menu.tpl');
```

只要./Public/menu.tpl 是一个实际存在的模板文件。如果使用的是相对路径的话，要注意当前位置是相对于项目的入口文件，而不是模板目录。

事实上，display 方法还有其他的参数和用法。

有时候某个模板页面我们需要输出指定的编码，而不是默认的编码，可以使用：

```
$this->display('Member:read', 'gbk');
```

或者输出的模板文件不是 text/html 格式的，而是 XML 格式的，可以用：

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

5.4.4 模板替换

在进行模板输出之前，系统还会对渲染的模板结果进行一些模板的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。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：

../Public： 会被替换成当前项目的公共模板目录 通常是 /项目目录/Tpl/default/Public/

__PUBLIC__：会被替换成当前网站的公共目录 通常是 /Public/

__TMPL__： 会替换成项目的模板目录 通常是 /项目目录/Tpl/default/

__ROOT__： 会替换成当前网站的地址（不含域名）

__APP__： 会替换成当前项目的 URL 地址 （不含域名）

__URL__： 会替换成当前模块的 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

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，我们只需要在项目配置文件中配置 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
    '__UPLOAD__' => '/Public/Uploads/',  // 增加新的上传路径替换规则
)
```

事实上，表单令牌验证的令牌验证字符串的自动生成，也是在这个阶段进行替换的。

5.4.5 获取内容

有些时候我们不想直接输出模板内容，而是希望对内容再进行一些处理后输出，就可以使用 fetch 方法来获取解析后的模板内容，在 Action 类里面使用：

```
$content = $this->fetch();
```

fetch 的参数用法和 Display 方法基本一致，也可以使用：

```
$content = $this->fetch('Member:read');
```

区别就在于 display 方法直接输出模板文件渲染后的内容，而 fetch 方法是返回模板文件渲染后的内容。如何对返回的结果 content 进行处理，完全由开发人员自行决定了。这是模板替换的另外一种高级方式，比较灵活，而且不需要通过配置的方式。

注意，fetch 方法仍然会执行上面的模板替换操作。

5.4.6 布局模板

新版的 ThinkPHP 可以自动识别模板文件中的布局模板，不再需要专门使用 layout 方法进行布局模板的输出了。布局模板本身的用法和普通的模板一样，只是增加了一个布局标签的用法，**并且布局模板可以用于任何模板引擎**，都可以很好的支持。

我们可以在布局模板里面使用下面的格式定义布局：

```
<!-- layout::模板文件规则::缓存时间（秒） -->
```

这里的模板文件规则和 display 的参数用法是一致的，详细的可以参考模板输出部分的内容。

例如：

```
<!-- layout::Public:header::60 -->
<!-- layout::Public:content::30 -->
<!-- layout::Public:footer::60 -->
```

如果使用的是内置的模板引擎的话，还可以使用下面的布局标签来定义，效果一致：

```
<layout name="Public:header" cache="60" />
<layout name="Public:content" cache="30" />
<layout name="Public:footer" cache="60" />
```

三个布局定义（标签）分别导入了三个模板文件，由于 Include 标签导入的外部文件无法检测模板更新，而布局模板恰好可以很好的解决这个问题。

假设上面的布局模板文件名称为 default.html 位于 Index 目录下面，那么我们就可以在 Action 里面调用输出：

```
$this->display('Index:default');
```

通常来说，我们可以不用重复定义很多的布局模板，使用动态布局模板来简化布局模板的定义。例如，我们把上面的布局模板修改为：

```
<!-- layout::Public:header::60 -->
<!-- layout::$content::30 -->
<!-- layout::Public:footer::60 -->
```

或者采用内置模板引擎的 layout 标签定义：

```
<layout name="Public:header" cache="60" />
<layout name="$content" cache="30" />
<layout name="Public:footer" cache="60" />
```

这样所有的模板都会调用 Public/header.html 头部和 Public/footer.html 尾部，而中间的内容是通过变量动态控制输出调用的模板。

```
$this->assign('content', 'User:list');
$this->display('Index:default');
```

布局模板只是为了方便视图的定义，布局模板本身并不调用模板对应的操作方法，也就是说布局模板中的变量赋值仍然需要在控制器中进行。

5.4.7 系统模板

系统有一些内置的模板文件用于异常页面和页面 Trace 功能的输出，你可以定制这些模板页面，满足自己的需要。默认的系统模板主要有：

页面 Trace 模板：默认位于系统目录的 Tpl/PageTrace.tpl.php 是一个 php 文件，可更改 **TMPL_TRACE_FILE** 进行配置。

异常模板：默认位于系统目录的 Tpl/ThinkException.tpl.php，可以更改 **TMPL_EXCEPTION_FILE** 进行配置。

以上两个系统模板都采用 php 原生语法定义，不支持模板标签。

5.4.8 静态生成

ThinkPHP 提供了灵活的静态文件生成功能，可以在输出模板的同时生成需要的静态文件，以供调用。

在 Action 中使用 buildHtml 方法即可创建静态文件，buildHtml 方法的第一个参数就要生成的静态文件名，后面的参数和 display 方法一致，内部其实是调用了前面提到的 fetch 方法获取模板输出然后创建静态文件。用法如下：

```
$this->buildHtml('静态文件', '静态路径','模板文件');
```

静态路径如果留空的话 默认保存在 HTML_PATH（默认的 HTML_PATH 路径位于项目目录下面的 Html 目录，如果没有的话手动创建）定义的路径下面，，静态文件可以随意设置，也可以包括路径，如果不存在的路径系统会自动创建，例如：

```
$this->buildHtml("Member/{$id}","','Member:read');
```

上面的用法表示获取 Member 模块的 read 操作模板输出内容后，根据用户的编号生成一个静态文件。位于 HTML_PATH 下面的 Member/1.html ，如果 Member 子目录不存在，系统会自动创建。

5.4.9 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，无论在功能还是性能方面都优秀过 Smarty。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板指南部分。

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。

5.4.10 使用第三方模板引擎

系统支持模板引擎的扩展机制，并且官方提供了包括 Smarty、EaseTemplate、TemplateLite 和 Smart 在内的第三方模板引擎扩展。我们以 Smarty 模板引擎为例，来说明下如何使用第三方模板引擎。

首先，需要下载官方的模板引擎扩展，并放到系统目录的 Lib\Think\Util\Template 目录下面，然后，下载最新的 Smarty 模板引擎文件放到系统目录的 Vendor 第三方类库目录。

剩下的，我们要做的只是简单的配置下模板引擎名称即可，例如在项目配置文件里面设置：

'TMPL_ENGINE_TYPE' => 'Smarty'

就可以用 smarty 标签来定义你的模板文件了，并且在模板文件的赋值和输出上面，和原来的方式一样，例如我们在上面提到的用 assign 赋值模板变量、display 和 fetch 方法的使用、模板文件的定位规则、模板替换功能仍然都可以使用。

对于某些第三方的模板引擎，还可以用 TMPL_ENGINE_CONFIG 参数进行自定义的配置。

例如对于 Smarty 模板引擎而言，我们可以进行下面的配置参数定义：

```
    'TMPL_ENGINE_CONFIG' => array(
            'caching' => true,
            'template_dir' => TMPL_PATH,
            'cache_dir' => TEMP_PATH,
    )
```

5.5 错误和日志

5.5.1 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。

一旦系统发生严重错误会自动抛出异常，也可以用 ThinkPHP 定义的 throw_exception 方法手动抛出异常。

throw_exception 方法支持三个参数：

$msg 异常信息，必须

$type 异常类型，即异常类的名称，默认是系统异常基础类 ThinkException

$code 异常代码 默认为 0

如果指定的异常类型不存在，系统自动调用 halt 方法直接输出异常信息文字，而不输出异常详细信息。

下面是 throw_exception 函数的一些使用例子：

```
throw_exception('新增失败');
```

```
throw_exception('信息录入错误','InfoException');
```

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 ThinkException('新增失败');
```

```
throw new InfoException('信息录入错误');
```

如果需要，我们建议在项目的类库目录下面增加 Exception 目录用于专门存放异常类库，以更加精确地定位异常。

异常页面的模板是可以修改的，如果没有定义，则采用系统内置的异常模板文件，该模板文件位于系统目录下目的 Tpl 目录下面的 ThinkException.tpl.php 文件。通过设置 EXCEPTION_TMPL_FILE 配置参数来修改系统默认的异常模板文件， 例如：

```
'EXCEPTION_TMPL_FILE' => APP_PATH.'/Public/exception.php'
```

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。

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。

5.5.2 日志处理

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，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 Log 类完成，提供了多种方式记录了不同的级别的日志信息。

5.5.3 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，例如一天 24 小时，而输入的是 25 小时这样

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

要开启日志记录，必须在配置中开启 LOG_RECORD 参数

我们可以在项目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_RECORD_LEVEL' => array('EMERG','ALERT','CRIT','ERR'),

只是记录 EMERG ALERT CRIT ERR 错误。

5.5.4 记录方式

日志的记录方式包括下面四种方式：

SYSTEM ：日志发送到 PHP 的系统日志记录

MAIL：日志通过邮件方式发送

TCP ：日志通过 TCP 方式发送

FILE ：日志通过文件方式记录（默认方式）

FILE 方式

默认采用文件方式记录日志信息，文件的格式是：年（简写）_月_日.log，例如：

09_10_01.log 表示 2009 年 10 月 1 日的日志文件

可以设置 LOG_FILE_SIZE 参数来限制日志文件的大小，超过大小的日志会形成备份文件。备份文件的格式是在当前文件名前面加上备份的时间戳，例如：

1189571417-07_09_12.log 备份的日志文件

日志文件的内容格式为：

[时间] 日志级别：日志信息

其中的时间显示可以动态配置，默认是采用 [c]，例如我们可以改成：

Log::$format = '[Y-m-d H:i:s]';

其格式定义和 date 函数的用法一致

默认情况下具体的日志信息类似于下面的内容：

[2009-08-25T18:09:22+08:00] NOTIC: [8] Undefined variable: verify PublicAction.class.php 第 162 行.

[2009-08-25T18:09:24+08:00] SQL: RunTime:0.214238s SQL = SHOW COLUMNS FROM think_user

[2009-08-25T18:09:24+08:00] SQL: RunTime:0.039159s SQL = SELECT * FROM `think_user` WHERE (`account` = 'admin') AND (`status` > 0) LIMIT 1

其他的日志类型的详细资料可以参考 PHP 手册中关于 error_log 方法的使用。

5.5.5 手动记录

通常日志文件的写入是自动完成的，如果我们需要在开发的过程中手动记录日志信息，可以使用 Log 类的方法来操作。日志文件的写入有两种方法：

一、使用 **Log::Write**($message,$level,$type,$file)

$message 是要记录的日志信息

$level 日志级别

$type 日志类型

$file 日志文件位置和名称，该参数可以改变系统默认的日志文件命名。

Write 方法把日志信息直接写入相关的日志文件里面。

Log::write('调试的 SQL：'.$SQL, Log::SQL);

二、使用 Log::record 和 Log::save 方法

Log::record($message,$level,$type);

其参数含义和 write 方法一致，不过 record 方法只是把日志信息保存到内存，并没有真正写入日志文件。直到调用 Log::save 方法。

Log::save()

保存 Log::record 方法记录的日志信息到日志文件。

例如：

```
Log::record('测试调试错误信息', Log::DEBUG);
Log::record('调试的 SQL：'.$SQL, Log::SQL);
Log::save();
```

5.6 调试

5.6.1 调试模式

在开启了调试模式之后，我们会看到更加详细的错误信息，调试模式的作用在于显示或者记录了更多的日志信息，以便我们在项目开发过程中快速定位和解决问题。

开启调试模式很简单，只要在项目配置文件里面设置

```
'APP_DEBUG' => true,
```

开启调试模式之后，系统在运行的时候首先会检查项目是否有定义调试配置文件，如果没有定义则调用框架默认的调试配置文件里面的参数，这些是系统为调试模式预设的默认配置。系统的默认调试配置文件位于 ThinkPHP\Common\debug.php。在这个默认的调试配置文件里面，系统开启了日志记录、关闭了页面防刷新机制、关闭了模板缓存，记录了执行过程中的 SQL 语句和运行时间，并且开启了页面运行时间显示和 Trace 功能。如果你觉得默认的调试配置不符合你的项目调试需要，你还可以在项目里面定义调试配置文件。

调试模式下面不会生成项目编译缓存，但是仍然会生成核心编译缓存，如果不希望生成核心缓存文件的话，可以在项目入口文件里面设置 NO_CACHE_RUNTIME，例如：

```
define('NO_CACHE_RUNTIME',True);
```

以及设置对编译缓存的内容是否进行去空白和注释，例如：

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

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

5.6.2 调试配置

我们可以给项目单独定义调试配置文件，用于项目在调试模式下面的配置信息。

项目的调试配置文件位于配置目录 Conf 目录下面，文件名定义为 **debug.php**，格式和项目配置文件的定义方法完全相同。

调试配置文件仅仅在启用调试模式的情况下有效，一旦项目关闭调试模式，就依然会使用项目配置文件里面的配置信息。

需要注意的是，调试模式下面并不是说项目只会加载调试配置文件，项目配置文件依然会首先加载的，只不过调试配置文件里面存在和项目配置文件有冲突的情况下，会覆盖项目配置文件里面的相同参数。也就是说，和项目配置文件相同的参数可以不必在调试模式下面进行定义。在下面的情况下，你通常会考虑使用项目的调试配置文件：

- ✧ 调试模式需要连接不同的测试数据库
- ✧ 需要增加额外的调试配置信息

5.6.3 运行状态

开启调试模式后，默认会显示当前页面的运行状态，这是一个包括了运行时间、内存开销、数据库读写次数和缓存读写次数的详细运行数据。显示结果信息类似于下面：

Process: 0.085s (Load:0.001s Init:0.005s Exec:0.025s Template:0.054s) | DB :2 queries 0 writes | Cache :1 gets 0 writes | UseMem:471 kb

最前面是整体的执行时间，中间是详细的阶段执行时间，然后是数据库读写次数和缓存读写次数显示，最后则是内存开销显示。如果当前页面没有任何数据库操作或者缓存操作的话，是不会显示相关信息的。内存开销的显示需要服务器开启 memory_get_usage 方法支持。

如果在非调式模式下面，其实我们也可以开启这样的运行状态显示。只需要在项目配置文件中开启相关的配置参数，如下：

```
'SHOW_RUN_TIME'=>true,          // 运行时间显示
'SHOW_ADV_TIME'=>true,          // 显示详细的运行时间
'SHOW_DB_TIMES'=>true,          // 显示数据库查询和写入次数
'SHOW_CACHE_TIMES'=>true,       // 显示缓存操作次数
'SHOW_USE_MEM'=>true,           // 显示内存开销
```

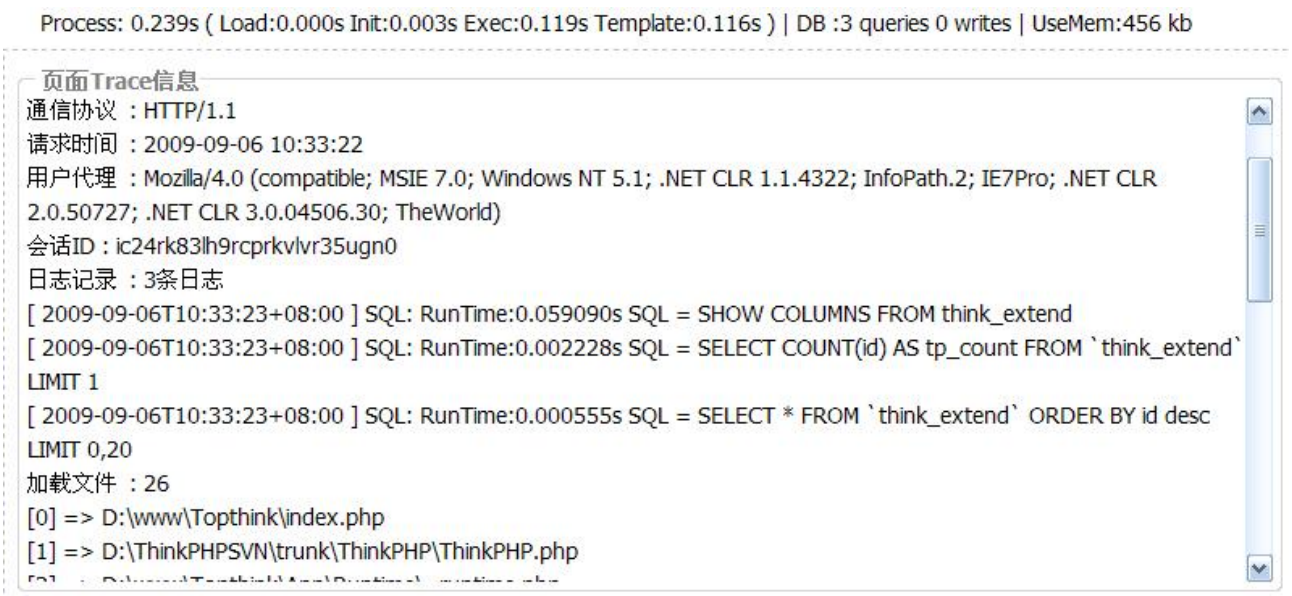
上面的每项参数都可以单独开启，例如，你只需要显示整体的运行时间，而不关心详细的阶段运行时间，可以关闭详细运行时间显示：

```
'SHOW_ADV_TIME'=> false,          // 关闭详细的运行时间
```

5.6.4 页面 Trace

页面 Trace 功能是 ThinkPHP 的一个用于开发调试的辅助手段。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，启用调试模式的话，页面 Trace 功能会默认开启（除非在项目的调试配置文件中关闭），并且系统默认的 Trace 信息包括：当前页面、请求方法、通信协议、请求

时间、用户代理、会话 ID、运行情况、SQL 记录、错误记录和文件加载情况。默认的页面 Trace 的显示如图所示：



Trace 页面定制

页面 Trace 信息的显示模板是可以定制的，默认位于系统目录的 Tpl/PageTrace.tpl.php 是一个 php 文件,可以根据项目自身的需要定制，更改 **TMPL_TRACE_FILE** 进行配置即可。

例如：

'TMPL_TRACE_FILE' => APP_PATH.'/Public/trace.php'

关键的输出代码是：

```
<?php foreach ($_trace as $key=>$info){
echo $key.' : '.$info.'<br/>';
}?>
```

Trace 信息定制

如果需要扩展自己的 Trace 信息，有下面几种方式：

第一种方式：在当前项目的配置目录下面定义 trace.php 文件，返回数组方式的定义，例如：

```
return array(
    '当前页面'=>$_SERVER['PHP_SELF'],
    '通信协议'=>$_SERVER['SERVER_PROTOCOL'],...
);
```

在显示页面 Trace 信息的时候会把这个部分定义的信息追加到系统默认的信息之后，这种方式通常用于 Trace 项目的公共信息。

第二种方式：在 Action 方法里面使用 trace 方法来增加 Trace 信息，该部分可以用于系统的开发阶段调试。例如：

```
$this->trace('执行时间',$runTime);
$this->trace('Name 的值',$name);
$this->trace('GET 变量',dump($_GET,false));
```

5.6.5 调试方法

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函数和类库。

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 var_dump 方法，用于输出变量的信息到浏览器查看。

```
dump($var, $echo=true, $label=null)  //输出变量信息
```

例如：

```
$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) ""
  ["userId"] => string(1) "0"
  ["categoryId"] => string(1) "0"
  ["title"] => string(4) "test"
  ["content"] => string(4) "test"
  ["cTime"] => string(1) "0"
  ["mTime"] => string(1) "0"
  ["status"] => string(1) "0"
  ["readCount"] => string(1) "0"
  ["commentCount"] => string(1) "0"
  ["tags"] => string(0) ""
}
```

使用下面的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

```
debug_start($label='') //记录调试开始时间
debug_end($label='')  //输出调试范围运行时间（相同 label 属于一个调试范围）
```

例如：

```
debug_start('run');
$blog = D("Blog");
$blog->select();
debug_end('run');
```

会输出下面的运行信息：

Process run: Times 0.007730s Memories 76 k

如果要输出内存占用情况，需要服务器支持 memory_get_usage 方法

我们可以使用下面的方法输出错误信息：

```
halt($msg)  //输出错误信息，并中止执行
```

5.6.6 模型调试

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

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

输出结果是

SELECT * FROM think_user WHERE id = 1

5.6.7 调试类

更高级的调试方法是使用 Debug 类

```
Debug::mark($name); // 标记一个调试位置
```

Debug::useTime($start,$end); // 返回区间所用的时间

Debug::useMemory($start,$end); // 返回区间所用的内存

5.7 缓存

5.7.1 缓存方式

ThinkPHP 在数据缓存方面包括文件方式、共享内存方式和数据库方式在内的多种方式进行缓存，通过插件方式还可以增加以后需要的缓存类，让应用开发可以选择更加适合自己的缓存方式，从而有效地提高应用执行效率。目前已经支持的缓存方式包括：File、Apachenote、Apc、Eaccelerator、Memcache、Shmop、Sqlite、Db 和 Xcache。

5.7.2 缓存类

所有的缓存方式都被统一使用公共的调用接口，这个接口就是 Cache 缓存类。

缓存类的使用很简单：

```
$Cache = Cache::getInstance('缓存方式','缓存参数');
```

例如，使用 Xcache 作为缓存方式，缓存有效期 60 秒。

```
$Cache = Cache::getInstance('Xcache',array('expire'=>'60'));
```

存取缓存数据

```
$Cache->set('name','ThinkPHP');  // 缓存 name 数据
$value = $Cache->get('name');  // 获取缓存的 name 数据
$Cache->rm('name');  // 删除缓存的 name 数据
```

或者使用下面的方法是等效的：

```
$Cache->name = 'ThinkPHP';
$value = $Cache->name;
Unset($Cache->name);
```

5.7.3 动态缓存

为了进一步简化缓存存取操作，ThinkPHP 把所有的缓存机制统一成一个 S 方法来进行操作，所以在使用不同的缓存方式的时候并不需要关注具体的缓存细节。例如：

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

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

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

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

当然，我们还可以在 S 方法里面显式的指定缓存方式，例如

```
S('data',$Data,3600,'File');
// 或者动态切换缓存方式
C('DATA_CACHE_TYPE','Xcache');
```

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

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

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

还可以设置哈希目录的层次，例如：

```
'DATA_PATH_LEVEL'=>2
```

就可以根据缓存标识的哈希自动创建多层子目录来缓存。

5.7.4 快速缓存

S 方法支持缓存有效期，在很多情况下，可能我们并不需要有效期的概念，或者使用文件方式的缓存就能够满足要求，所以系统还提供了一个专门用于文件方式的快速缓存方法 F 方法。**F 方法只能用于缓存简单数据类型，不支持有效期和缓存对象**，使用如下：

快速缓存 Data 数据，默认保存在 DATA_PATH 目录下面

```
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，例如：

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。

5.7.5 静态缓存

ThinkPHP 内置了静态缓存的功能，并且支持静态缓存的规则定义。

要使用静态缓存功能，需要开启 HTML_CACHE_ON 参数，并且在项目配置目录下面增加静态缓存规则文件 htmls.php，两者缺一不可。否则静态缓存不会生效。

静态规则文件的定义方式如下：

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

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

第一种是定义全局的操作静态规则，例如定义所有的 read 操作的静态规则为

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

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

第二种是定义全局的模块静态规则，例如定义所有的 User 模块的静态规则为

'User:'=>array('User/{:action}_{id}','600')

其中，{:action} 表示当前的操作名称

第三种是定义某个模块的操作的静态规则，例如，我们需要定义 Blog 模块的 read 操作进行静态缓存

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

有个别特殊的规则，例如空模块和空操作的静态规则的定义，可以使用下面的方式：

'Empty:index'=>array('{:module}_{:action}',-1) // 定义空模块的静态规则

'User:_empty'=>array('User/{:action}',-1) // 定义空操作的静态规则

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}'), 根据当前的 URL 进行缓存

静态规则的写法可以包括以下情况

1、使用系统变量 包括 _GET _REQUEST _SERVER _SESSION _COOKIE

格式：{$_×××|function}

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

2、使用框架特定的变量

例如：{:app}、{:group} 、{:module} 和{:action} 分别表示当前项目名、分组名、模块名和操作名

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。

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

附加规则通常用于对静态规则进行函数运算，例如

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

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

和静态缓存相关的配置参数包括：

HTML_CACHE_ON 是否开启静态缓存功能

HTML_FILE_SUFFIX 静态文件后缀 惯例配置的值是 .shtml

HTML_CACHE_TIME 默认的静态缓存有效期 默认 60 秒 可以在静态规则定义覆盖

HTML_READ_TYPE 页面静态化后读取的规则

一种是直接读取缓存文件输出（readfile 方式 HTML_READ_TYPE 为 0） 这是系统默认的方式，属于隐含静态化，用户看到的 URL 地址是没有变化的。

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

5.8 安全

5.8.1 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，系统也会自动加上引号避免恶意注入。事实上，ThinkPHP 对所有的数据库 CURD 的数据都会进行 escape_string 处理。

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；

✧ 开启数据字段类型验证，可以对数值数据类型做强制转换；

✧ 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；

字段类型检查、自动验证和自动完成机制我们在模型部分已经有详细的描述，

5.8.2 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远程提交；

✧ 使用自动验证和自动完成机制进行初步过滤；

✧ 对用户输入的数据进行有效（根据你的应用）的过滤，常见的安全过滤函数包括 stripslashes、htmlentities、htmlspecialchars 等，官方的扩展类库中的 ORG.Util.Input 类则提供了更好的解决方法；

5.8.3 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；

5.8.4 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，官方的扩展类库中的 ORG.Net.UploadFile 类提供了上传类的安全解决方案。

✧ 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）；

5.8.5 目录安全文件

对于某些服务器开启了目录浏览权限的话，用户就可以直接在浏览器输入 URL 地址查看目录了。系统内建了目录安全文件机制，可以有效的解决此类问题。

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

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

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

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

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

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

下面是一个完整的使用目录安全写入的例子

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

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

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

5.8.6 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。

把以下代码保存在项目的模板目录目录（默认是 Tpl）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。

按照我们之前提过的网站安全部署方案，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。

5.9 部署

5.9.1 部署优化

在部署阶段，请关闭调试模式，并且注意下面事项，进行尽可能的性能优化：

✧ 如果非必要，请在项目配置中关闭任何日志写入；

✧ 开启模板缓存，并设置有效期为-1；

✧ 启动 ALLINONE 模式（后面会讲到）；

✧ 对于实时性要求不高的动态数据进行缓存处理；

5.9.2 ALLINONE 模式

ALLINONE 模式指的是 ThinkPHP 可以把核心编译缓存和项目编译缓存合并到一个文件里面去，并且过滤掉一些运行模式不需要执行的代码，并且对于用户的自定义常量全部统一定义，不再进行额外的检测。ALLINONE 模式一般是在开发调试完成之后，希望进一步提高系统的整体性能的时候开启。开启 ALLINONE 模式只需要在入口文件中添加定义：

```
define('RUNTIME_ALLINONE', true);  // 开启 ALLINONE 运行模式
```

开启 ALLINONE 运行模式后需要清空系统原来的编译缓存文件，第一次运行的时候系统会自动生成一个~allinone.php 的缓存文件，第二次就会直接读取缓存文件而跳过一些不必要的初始化过程。~allinone.php 编译缓存文件不是简单的~runtime.php 和~app.php 的合并，剔除了一些运行模式过程中不需要的方法和代码。

需要注意的是，在 ALLINONE 模式下面，即使调试模式开启也是无效的。系统不支持对 ALLINONE 运行模式的开发调试功能。因此，大多数情况用于生产部署环境。

5.10 杂项

5.10.1 多语言

ThinkPHP 内置多语言支持，如果你的应用涉及到国际化的支持，那么可以定义相关的语言包文件。任何字符串形式的输出，都可以定义语言常量。可以为项目定义不同的语言文件，框架的系统语言包目

录在系统框架的 Lang 目录下面，每个语言都对应一个语言包文件，系统默认只有简体中文语言包文件 zh-cn.php ，如果要增加繁体中文 zh-tw 或者英文 en，只要增加相应的文件。

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

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

throw_exception（'新增用户失败！'）;

我们在语言包里面增加了 ADD_USER_ERROR 语言配置变量的话，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。

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

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

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

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

语言文件定义

ThinkPHP 语言文件定义采用返回数组方式：

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

要在程序里面设置语言定义的值，使用下面的方式：

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

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

通常多语言的使用是在 Action 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！'）

还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 Action 中赋值，可以直接使用模板引擎特殊标签来直接输出语言定义的值：

{$Think.lang.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

5.10.2 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。

分页类位于扩展类库下面，需要先导入才能使用（关于如何导入扩展类库，请参考扩展指南部分内容），下面是数据分页的两种示例。

第一种分页方法是利用 Page 类和 limit 方法：

```
$User = M("User"); // 实例化 User 对象
import("ORG.Util.Page"); // 导入分页类
$count      = $User->where("status=1")->count(); // 查询满足要求的总记录数
$Page       = new Page($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
// 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
```

另外一种方式是分页类和 page 方法的实现

```
$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
import("ORG.Util.Page"); // 导入分页类
$count      = $User->where("status=1")->count(); // 查询满足要求的总记录数
$Page       = new Page($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：

```
import("ORG.Util.Page"); // 导入分页类
$mapcount      = $User->where($map)->count(); // 查询满足要求的总记录数
$Page       = new Page($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

默认的分页输出效果是

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：

header：头部描述信息，默认值 “条记录”

prev：上一页描述信息，默认值是“上一页”

next：下一页描述信息，默认值是“下一页”

first：第一页描述信息，默认值是“第一页”

last：最后一页描述信息，默认值是“最后一页”

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

“%totalRow% %header% %nowPage%/%totalPage%

页 %upPage% %downPage% %first% %prePage% %linkPage% %nextPage% %end%”

通过 setConfig 设置以上属性可以完美的定制出你的分页显示风格。

5.10.3 文件上传

上传类使用 ORG 类库包中的 Net.UpdateFile 类，最新版本的上传类包含的功能如下（有些功能需要结合 ThinkPHP 系统其他类库）：

- ✧ 基本上传功能
- ✧ 支持批量上传
- ✧ 支持生成图片缩略图
- ✧ 自定义参数上传
- ✧ 上传检测（包括大小、后缀和类型）
- ✧ 支持覆盖方式上传
- ✧ 支持上传类型、附件大小、上传路径定义
- ✧ 支持哈希或者日期子目录保存上传文件
- ✧ 上传图片的安全性检测
- ✧ 支持上传文件命名规则
- ✧ 支持对上传文件的 Hash 验证

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 METHOD=POST action="__URL__/upload" enctype="multipart/form-data" >
<input type="text" NAME="name"  >
```

```
<input type="text" NAME="email"  >
<input type="file"  name="photo" >
<input type="submit" value="保 存" >
</form>
```

注意表单的 Form 标签中一定要添加 **enctype="multipart/form-data"** 文件才能上传。因为表单提交到当前模块的 upload 操作方法，所以我们在模块类里面添加下面的 **upload 方法**即可：

```
Public function upload(){
   import("ORG.Net.UploadFile");
   $upload = new UploadFile(); // 实例化上传类
   $upload->maxSize  = 3145728 ; // 设置附件上传大小
   $upload->allowExts  = array('jpg', 'gif', 'png', 'jpeg'); // 设置附件上传类型
   $upload->savePath =  './Public/Uploads/'; // 设置附件上传目录
   if(!$upload->upload()) { // 上传错误 提示错误信息
       $this->error($upload->getErrorMsg());
   }else{ // 上传成功 获取上传文件信息
       $info =  $upload->getUploadFileInfo();
    }
   // 保存表单数据 包括附件数据
   $User = M("User"); // 实例化 User 对象
   $User->create(); // 创建数据对象
   $User->photo = $info[0]["savename"]; // 保存上传的照片 根据需要自行组装
```

```
    $User->add(); // 写入用户数据到数据库
    $this->success("数据保存成功！");
}
```

首先是实例化上传类

```
import("ORG.Net.UploadFile");
$upload = new UploadFile(); // 实例化上传类
```

实例化上传类之后，就可以设置一些上传的属性（参数），支持的属性有：

maxSize： 文件上传的最大文件大小（以字节为单位）默认为-1 不限大小

savePath：文件保存路径，如果留空会取 UPLOAD_PATH 常量定义的路径

saveRule：上传文件的保存规则，必须是一个无需任何参数的函数名，例如可以是 time、 uniqid com_create_guid 等，但必须能保证生成的文件名是唯一的，默认是 uniqid

hashType：上传文件的哈希验证方法，默认是 md5_file

autoCheck：是否自动检测附件，默认为自动检测

uploadReplace：存在同名文件是否是覆盖

allowExts：允许上传的文件后缀（留空为不限制），使用数组设置，默认为空数组

allowTypes：允许上传的文件类型（留空为不限制），使用数组设置，默认为空数组

thumb：是否需要对图片文件进行缩略图处理，默认为 false

thumbMaxWidth：缩略图的最大宽度，多个使用逗号分隔

thumbMaxHeight：缩略图的最大高度，多个使用逗号分隔

thumbPrefix：缩略图的文件前缀，默认为 thumb_

thumbSuffix：缩略图的文件后缀，默认为空

thumbPath：缩略图的保存路径，留空的话取文件上传目录本身

thumbFile：指定缩略图的文件名

thumbRemoveOrigin：生成缩略图后是否删除原图

autoSub：是否使用子目录保存上传文件

subType：子目录创建方式，默认为 hash，可以设置为 hash 或者 date

dateFormat：子目录方式为 date 的时候指定日期格式

hashLevel：子目录保存的层次，默认为一层

以上属性都可以直接设置，例如：

```
$upload->thumb = true
$upload->thumbMaxWidth = "50,200"
$upload->thumbMaxHeight = "50,200"
```

其中生成缩略图功能需要 Image 类的支持。

设置好上传的参数后，就可以调用 UploadFile 类的 upload 方法进行附件上传，如果失败，返回 false，并且用 **getErrorMsg** 方法获取错误提示信息；如果上传成功，可以通过调用 **getUploadFileInfo** 方法获取成功上传的附件信息列表。因此 getUploadFileInfo 方法的返回值是一个数组，其中的每个元素就是上传的附件信息。每个附件信息又是一个记录了下面信息的数组，包括：

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

extension：上传文件的后缀类型

hash：上传文件的哈希验证字符串

文件上传成功后，就可以通过这些附件信息来进行其他的数据存取操作，例如保存到当前数据表或者单独的附件数据表都可以。

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。

5.10.4 验证码

要使用验证码，需要导入扩展类库中的 ORG.Util.Image 类库和 ORG.Util.String 类库。我们通过在在模块类中增加一个 verify 方法来用于显示验证码：

```
Public function verify(){
   import("ORG.Util.Image");
   Image::buildImageVerify();
}
```

Image 类的 buildImageVerify 方法用于生成验证码，该方法有以下参数可选：

buildImageVerify($length,$mode,$type,$width,$height,$verifyName)

length：验证码的长度，默认为 4 位数

mode：验证字符串的类型，默认为数字，其他支持类型有 0 字母 1 数字 2 大写字母 3 小写字母 4 中文 5 混合（去掉了容易混淆的字符 oOLl 和数字 01）

type：验证码的图片类型，默认为 png

width：验证码的宽度，默认会自动根据验证码长度自动计算

height：验证码的高度，默认为 22

verifyName：验证码的 SESSION 记录名称，默认为 verify

定义完成后，验证码的显示只需要在模板文件中添加：

```
<img src="__APP__/Public/verify/" />
```

运行后可以看到类似下面的验证码显示：

每次生成验证码的时候，就会通过 SESSION 记录本次的验证码的 md5 后的字符串信息，所以，要检查验证码是否正确，我们只需要在 Action 中使用下面的代码就行了：

```
 if($_SESSION['verify'] != md5($_POST['verify'])) {
    $this->error('验证码错误！');
```

}

注意，这里的 verify 名称取决于你的验证码的 verifyName 参数的值。

buildImageVerify 方法不支持中文验证码的显示，如果需要显示中文验证码，请使用

GBVerify 方法，参数如下：

GBVerify ($length,$type,$width,$height,$fontface,$verifyName)

length：验证码的长度，默认为 4 位数

type：验证码的图片类型，默认为 png

width：验证码的宽度，默认会自动根据验证码长度自动计算

height：验证码的高度，默认为 50

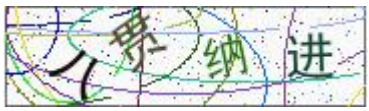
fontface：使用的字体文件，使用完整文件名或者放到图像类所在的目录下面，默认使用的字体文件是 simhei.ttf（该文件可以从 window 的 Fonts 目录下面找到）

verifyName：验证码的 SESSION 记录名称，默认为 verify

例如

```
Public function verify(){
   import("ORG.Util.Image");
   Image::GBVerify();
}
```

显示效果如下：



如果无法显示验证码，请检查：

✧ PHP 是否已经安装 GD 库支持；

✧ 输出之前是否有任何的输出（尤其是 UTF8 的 BOM 头信息输出）；

✧ Image 类库是否正确导入；

✧ 如果是中文验证码检查是否有拷贝字体文件到类库所在目录；

5.10.5 权限验证

对应上面的安全体系，ThinkPHP 的 RBAC 认证的过程大致如下：

1. 判断当前模块的当前操作是否需要认证
2. 如果需要认证并且尚未登录，跳到认证网关，如果已经登录 执行 5
3. 通过委托认证进行用户身份认证
4. 获取用户的决策访问列表
5. 判断当前用户是否具有访问权限

6 扩展指南

ThinkPHP 是一个轻量级的 WEB 应用开发框架，也就意味着自身并没有庞大的外围应用类库，也不可能仅仅通过核心来解决百分百的应用需求，而这些完全可以通过系统内建的扩展机制来扩展和完善。

下面我们会详细介绍如何对你的 ThinkPHP 应用在不修改核心的情况下进行轻松的扩展。

6.1 类库扩展

6.1.1 基类库扩展

ThinkPHP 的基类库目录位于 ThinkPHP\Lib，默认的基类库只包含 Think 类库包。系统基类库可以很方便的进行扩展，目前支持的类库包包括 ORG（第三方公共类库包）和 Com（企业类库包）。你可以在 ORG 类库目录下面添加自己需要的类库（ThinkPHP 基类库的所有类库文件统一使用 class.php 作为后缀，并且文件名和类名相同），你甚至还可以创建属于自己企业的类库，只需要在 ThinkPHP\Lib\目录下面创建 Com 目录，然后在里面增加相应的类库就可以方便的使用 import 方法导入了。

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

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

6.1.2 应用类库扩展

项目类库的扩展，和基类库的扩展一样，我们可以在项目类库目录增加你想要的子目录，例如，我们在 MyApp 的项目目录下面增加 Common 和 Util 目录，就可以这样加载这些目录下面的类库文件了：

```
import('MyApp.Util.UnitTest');
import('@.Common.CommonUtil');
```

6.1.3 第三方类库扩展

如果你直接使用的是第三方的类库包，或者是类名和后缀和 ThinkPHP 的默认规则不符合的，我们建议你放到 ThinkPHP\Vendor 目录下面，并使用 vendor 方法来导入。

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 方法导入就是：

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

6.2 应用扩展

应用扩展是指不改变现有底层框架的基础上，对 App 类进行额外的功能扩展，系统使用了标签扩展的方式。要启用应用扩展支持，必须在项目配置文件里面开启 **APP_PLUGIN_ON** 配置参数。

```
'APP_PLUGIN_ON'=>true,
```

一旦开启后，系统就会检查下面的标签：

- ✧ app_begin ：应用开始标签
- ✧ app_init ：应用初始化标签
- ✧ app_run ：应用执行标签
- ✧ app_end ：应用结束标签

以上是系统的 App 应用类内置的一些标签位置，在每个定义的标签位置，都会执行一个 tag 方法来调用该标签位置需要执行的方法，例如：

我们可以看到，标签的执行只是一个很简单的代码，例如：

```
// 执行应用初始化标签
tag('app_init');
```

系统执行到这里的时候，会自动检查标签所对应要执行的方法，并且依次执行。

标签对应的执行方法是通过标签配置定义文件，在项目的配置目录下面增加 tags.php 文件，写入：

```
return array(
   // 定义项目初始化标签要执行的方法
   'app_init'=>array(
   'function1','function2',array('class1','method1')...
   ),
   ...// 其他的标签
);
```

如果某个标签位置需要传入额外的参数，可以使用 tag('app_init',$data);

会自动传入要执行的方法，注意参数必须一致才能准确调用。

应用标签扩展的方式，其实可以延伸到项目中，我们可以在项目的某些位置手动插入标签位，然后定义外部的标签扩展来执行。根据这样的一个原理，标签扩展可以随意定制。你需要做的仅仅是在需要执行的位置 加上 tag('标签名称'[,'可选参数'…]) 即可，然后在 tags.php 文件里面定义好各个标签要执行的方法，其他的事情系统会自动处理。

6.3 控制器扩展

6.3.1 模块扩展

模块扩展可以使得项目方便的动态挂载模块，动态模块只需要在项目配置目录下面定义模块定义文件 modules.php，定义格式为：

```
return array(
```

```
'moduleName' => array('导入路径'[, '类名']),
);
```

例如，我们定于了一个名称为 Extend 的扩展模块，其模块类的文件路径位于项目的 Lib\Modules\ExtendAction.class.php，那么定义如下：

```
return array(
'Extend' => array('@.Modules.Extend'),
);
```

一般情况下，类名无需指定，会按照默认的规则去找，如果你的类名和系统规则不一致，就需要指定类名，假如模块文件名是 Extend.class.php：

```
return array(
'Extend' => array('@.Modules.Extend', 'Extend']),
);
```

 注意事项：

扩展的模块一定是现有项目里面没有的，否则无效；

更改扩展模块定义后，需要删除项目编译缓存文件；

动态模块的规则比空模块的规则要优先。

6.3.2 操作扩展

操作扩展可以使得项目方便的动态挂载某些操作，而这些操作可以是针对个别模块的，也可以是针对全局的。操作扩展只需要在项目配置目录下面定义操作定义文件 actions.php，定义格式为：

```
return array(
```

```
    // 全局操作
    'actionName' => '调用方法',
    // 局部操作
    'moduleName:actionName' => '调用方法',
);
```

其中调用方法可以是某个自定义的函数，或者是某个类的方法，类似于 PHP 的 callback 定义。

注意事项：

扩展的操作一定是现有项目里面没有的，否则无效；

更改操作扩展定义后，需要删除项目编译缓存文件；

动态操作的规则比空操作的规则要优先。

6.4 模型扩展

ThinkPHP 的新版模型具有很好的扩展性，对模型的 CURD 方法都提供了扩展接口。

模型类提供了多个回调接口，主要扩展接口如下：

全局接口：

初始化接口 **_initialize**()

表达式过滤接口 **_options_filter**(&$options)

add 方法接口：

写入前置接口 **_before_insert**(&$data,$options)

写入后置接口 **_after_insert**($data,$options)

save 方法接口：

更新前置接口 **_before_update**(&$data,$options)

更新后置接口 **_after_update**($data,$options)

同时对 add 和 save 方法的接口：

数据写入（包括写入和更新）数据库之前的处理接口 **_facade**($data)

delete 方法接口：

删除后置接口 **_after_delete**($data,$options)

select（findall）方法接口：

查询后置接口（数据集） **_after_select**(&$result,$options)

find 方法接口：

查询后置接口 （数据） **_after_find**(&$result,$options)

系统内置的高级模型 AdvModel、视图模型 ViewModel 和关联模型 RelationModel 本身就是一个模型扩展的很好的例子，本身都是继承 Model 类并且都通过了扩展实现很多的功能。

6.5 驱动扩展

数据库抽象层的设计是由抽象数据库操作类和数据库驱动类组成的，内置的数据库驱动是 MySQL 和 MySQLi 驱动类，官方的扩展还提供了 MsSQL、PgSQL、Sqlite、Oracle、Ibase 以及 PDO 驱动类，可以满足常用的数据库操作的需要。

要扩展其他的数据库驱动类，只需要继承 Db 类，驱动类的命名规范是：

Db+驱动类名称（首字母大写）

例如，假如你需要扩展一个 ODBC 的数据库驱动，应该命名为：DbOdbc.class.php，并放到系统的 Lib\Think\Db\Driver 目录下面。

```
Class DbOdbc  extends Db{
}
```

然后，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
```

每个数据库驱动需要实现的方法包括（具体参数可以参考现有的数据库驱动类库）：

- ✧ 架构和析构方法
- ✧ Connect 连接数据库方法
- ✧ Free 释放查询方法
- ✧ Query 查询操作方法
- ✧ Execue 执行操作方法
- ✧ startTrans 开启事务方法
- ✧ commit 事务提交方法
- ✧ rollback 事务回滚方法
- ✧ getAll 获取查询数据方法
- ✧ getFields 取得数据表的字段信息
- ✧ getTables 取得数据库的表信息
- ✧ close 关闭数据库连接方法

✧ error 获取数据库错误信息

✧ escape_string SQL 安全过滤方法

6.6 Widget 扩展

Widget 扩展用于在页面根据需要输出不同的内容，Widget 扩展的定义是在项目的 Lib\Widget 目录下面定义 Widget 类库，例如下面定义了一个用于显示最近的评论的 Widget：

位于 Lib\Widget\ShowCommentWidget.class.php

Widget 类库需要继承 Widget 类，并且必须定义 render 方法实现，例如：

```
class ShowCommentWidget extends Widget{
    public function render($data){
        return '这是最新的评论信息';
    }
}
```

render 方法必须使用 return 返回要输出的字符串信息，而不是直接输出。

Widget 也可以调用 Widget 类的 renderFile 方法，渲染模板后进行输出，。

```
class ShowCommentWidget extends Widget{
    public function render($data){
      $content      =      $this->renderFile('Article:comment',$data);
        return $content;
    }
}
```

定义好 Widget 类库后，只需要做的是在模板文件里面使用 W 方法调用 Widget，例如

{:W('ShowComment')}

通常 Widget 都有自己的调用参数来决定不同的输出内容

{:W('ShowComment',array('count'=>5))}

参数必须使用索引数组传入。

在控制器里面也可以调用 Widget 类进行输出，在 Action 里面获取动态的 Widget 内容，可以使用下面的方式：

```
$content  =  W('ShowComment', array('count'=>5),true);
```

第三个参数表示是否返回字符串，如果是 false 就表示直接输出。返回值可以用于其他用途。

6.7 行为扩展

行为扩展和 Widget 扩展的区别其实就是 Widget 是用于输出的，而行为通常是执行某个方法，但通常都不需要输出，即使输出的话也许是错误提示信息之类的。

行为是可以和应用扩展配合的，因为应用扩展是很随意的，但是行为却是可以规范的。定义好的行为扩展，可以被任何应用扩展中的标签单独调用。

行为类的定义也很简单，例如下面是一个代理检测访问行为的扩展：

```
class AgentCheckBehavior extends Behavior {
    public function run() {
        // 代理访问检测
        if(C('LIMIT_PROXY_VISIT') && ($_SERVER['HTTP_X_FORWARDED_FOR'] ||
$_SERVER['HTTP_VIA'] || $_SERVER['HTTP_PROXY_CONNECTION'] ||
$_SERVER['HTTP_USER_AGENT_VIA'])) {
            // 禁止代理访问
```

```
            exit('Access Denied');
        }
    }
}
```

行为类必须定义一个 run 接口方法，否则无法正确调用。

命名为 AgentCheckBehavior.class.php 后 放入项目的 Lib\Behavior 目录下面。

接下来就是调用这个行为，在调用的地方只需要使用：

```
B('AgentCheck');
```

配合应用扩展机制的话，例如我们在项目初始化标签的执行方法里面使用了上面的代码，就会在项目初始化的时候自动调用该行为了。

6.8 标签库扩展

6.8.1 标签库原理

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，ThinkTemplate 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 XML 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 ThinkTemplate 中标签库的体现是采用 XML 命名空间的方式。

标签库由定义文件和解析类构成。每个标签库存在一个 XML 定义文件，用来定义标签库中的标签和属性。并且一个标签库文件对应一个标签库解析类，每个标签就是解析类中的一个方法。例如，CX 标签

库的定义文件是 cx.xml 位于 ThinkTemplate/Template/Tags/目录下面，而 cx 标签库解析类文件是位于 ThinkTemplate/Template/TagLib/目录下面的 TagLibCx.class.php 文件，每个标签的解析方法就是 TagLibCx 类的一个方法，为了不和系统的关键字冲突，所以在方法名前加上了“_”前缀，因此，假如要定义 Cx:Var 的标签解析，就需要定义一个 _var 方法。

标签库解析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）。扩展标签库需要添加标签库定义 XML 文件和标签库解析类。

标签库定义 XML 文件的格式为：

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib>
<tag>
<name>标签名称</name>
<nested>是否允许嵌套</nested>
<alias>标签别名</alias>
<bodycontent>是否属于闭合标签</bodycontent>
<attribute>
<name>属性名称</name>
<required>是否必须</required>
</attribute>
</tag>
</taglib>
```

标签库的名称和文件名一致，每个 tag 标签对定义了标签库中的一个标签，每个 tag 节点的属性定义规范如下：

✧ name：标签名称

✧ nested：是否允许标签嵌套（true 或 false）

✧ alias：标签别名（多个逗号分隔）

✧ bodycontent：是否为闭合标签（true 或 empty）

✧ attribute：标签允许的属性

每个标签节点可以包含多个属性，也就是 tag 节点可以定义多个 attribute 节点，每个 attribute 属性支持两个属性：name 和 required，required（true 或 false）表示该属性是否为必须。

然后，我们看解析类的定义，每个标签的解析方法在定义的时候需要添加“_”前缀，可以传入两个参数，属性字符串和内容字符串（对于非闭合标签）。必须通过 return 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 include 解析方法的定义：

```
public function _include($attr,$content)
{
    $tag    = $this->parseXmlAttr($attr,'include');
    $file   =   $tag['file'];
    return $this->tpl->parseInclude($file);
}
```

在每个标签的解析方法中，首先需要调用

```
$this->parseXmlAttr($attr,'include');
```

表示分析某个标签的 XML 定义，返回 include 的所有标签属性。接下来就是根据具体的属性值来返回实际的解析内容了。

6.8.2 普通标签扩展

普通标签的扩展可以通过配置 TAG_EXTEND_PARSE 参数来进行，例如，我要扩展一个下面的 js 标签实现：

{js:/Public/Js/Think.js}

希望能够实现加载/Public/Js/Think.js 的结果。

我们可以在项目配置文件中定义：

```
TAG_EXTEND_PARSE=>array(
    "js"=>"parse_js_load",   // 定义 js 标签的解析方法 js 必须使用小写定义
)
```

然后我们在项目的函数库里面添加 parse_js_load 函数，如下：

```
function parse_js_load($str){
return  '<script type="text/javascript" src="'.$str.'"></script>';
}
```

在删除项目的编译缓存文件~app.php 后，就可以使用 hello 普通标签了。虽然这个例子非常简单，但是只要理解以后就可以扩展出功能强大的普通标签了。

6.8.3 扩展标签库

要扩展标签库，有两种方式：

第一种，直接把标签库放入系统的标签库目录。

首先，把标签库的定义文件放入系统的标签库定义目录 Lib/Think/Template/Tags/。把标签库的解析类库放入 Lib/Think/Template/TagLib/目录。

然后在模板页面添加：

<taglib name='标签库名称' />

这样就可以直接使用扩展的标签库了。

第二种，通过配置的方式加载标签库。

这种方式需要在项目配置文件里面定义 taglibs.php 文件，格式如下：

```
return array(
   '标签库 1'=>'标签库 1 解析类库路径', // 使用 import 方法支持的路径格式
   '标签库 2'=>'标签库 2 解析类库路径',
   …
   );
```

例如：

```
return array(
'mytag'=>'@.TagLib.TagLibMytag',
);
```

然后在项目的 Lib\TagLib\目录下面，增加一个 TagLibMytag.class.php 标签库解析文件，

标签库解析类的命名是：TagLib+标签库名称（首字母大写）

标签库定义文件可以放在 Lib\TagLib\Tags\ 下面或者自己定义（参考下面的初始化方法），名称通常是标签库的名称。

```
class TagLibMytag.class.php extends TagLib{
    // 初始化标签库的定义文件
    public function _initialize() {
        $this->xml = dirname(__FILE__).'/Tags/mytag.xml';
    }
}
```

定义_initialize 方法的目的是定位标签库的定义 XML 文件，这样标签库就可以完全独立系统在项目中存在了。

6.9 模板引擎扩展

除了使用内置的模板引擎外，系统还支持模板引擎扩展。并且官方已经提供了包括 Smarty、EaseTemplate、TemplateLite 和 Smart 在内的第三方模板引擎扩展。

模板引擎扩展类库的命名为：Template+模板引擎名称（首字母大写）

模板引擎扩展类只需要实现一个 fetch 接口方法，参数为：

function fetch(模板文件,模板变量,模板编码)

例如：

```
    public function fetch($templateFile,$var,$charset) {
        $templateFile=substr($templateFile,strlen(TMPL_PATH));
        vendor('Smarty.Smarty#class');
        $tpl = new Smarty();
        if(C('TMPL_ENGINE_CONFIG')) {
            $config  =  C('TMPL_ENGINE_CONFIG');
            foreach ($config as $key=>$val){
```

```
            $tpl->{$key}  =  $val;
        }
    }else{
        $tpl->caching = C('TMPL_CACHE_ON');
        $tpl->template_dir = TMPL_PATH;
        $tpl->compile_dir = CACHE_PATH ;
        $tpl->cache_dir = TEMP_PATH ;
    }
    $tpl->assign($var);
    $tpl->display($templateFile);
}
```

如果扩展类库中需要涉及到第三方类库，可以放到 Vendor 目录下面，以供调用。

要使用扩展模板引擎的话，只需要在项目配置文件中添加：

'TMPL_ENGINE_TYPE' => '模板引擎名称'

就可以使用对应的模板引擎来定义模板文件了。影响是只是模板文件的定义，视图操作方法保持原来的不变。例如在 Action 中用 assign 赋值模板变量、display 和 fetch 方法的使用、模板文件的定位规则、模板替换功能仍然一致。

6.10 模式扩展

6.10.1 使用内置的模式

我们前面所涉及的所有用法都是基于框架的标准模式的，除了标准模式之外，官方的发布版本还内置了几种常用的模式扩展，包括：Cli（命令模式）、Lite（精简模式）、Thin（简洁模式），他们为不同

的需求提供了不同的底层框架解决方案。通常来说不同的模式之间是无法进行切换，下面阐述下这几种模式和标准模式的区别：

Thin 模式：简洁模式

主要区别在于：

✧ 默认不使用任何模板引擎（可以自己在操作方法里面调用）；

✧ 模型仅支持原生 SQL 操作和事务；

✧ 支持多数据库切换和连接；

✧ 默认仅支持 MySQL 数据库；

✧ 不支持语言包、模块分组、模板主题和 Dispatch 功能；

✧ 去除了大部分扩展机制；

如果你的应用选择了 Mysql 数据库，并且完全使用原生 SQL 操作，并希望有一个轻巧的核心，那么简洁模式是一个很好的选择。

要使用简洁模式，需要在项目的入口文件中添加模式定义：

```
define('THINK_MODE','Thin');  // 采用简洁模式运行
```

Lite 模式：精简模式

在简洁模式的基础上，增加了：

✧ 默认使用 PHP 模板；

✧ 支持不带路由的 Dispatch；

✧ 支持不带回调接口的 CURD 操作；

✧ 支持连贯操作、统计查询；

精简模式比简洁模式在模型方面多了 CURD 和连贯操作，如果你习惯于使用 PHP 作为模板，并且还是喜欢使用模型的 CURD 功能，但又不希望核心那么庞大，那么精简模式是一个不错的选择。

要使用精简模式，需要在项目的入口文件中添加模式定义：

```
define('THINK_MODE','Lite'); // 采用精简模式运行
```

Cli 模式：命令行模式

和简洁模式基本类似，只是支持命令行下面的参数解析。

要使用命令行模式，需要在项目的入口文件中添加模式定义：

```
define('THINK_MODE','Cli'); // 采用命令模式运行
```

6.10.2 定制模式扩展

模式扩展的本质其实就是组装自己的新的基于 ThinkPHP 的框架核心。这是新版框架底层可组装可定制的重要思想。基于 ThinkPHP 的意思是能够使用 ThinkPHP 的内置函数方法、配置方式、独特的编译缓存机制，以及一些常量定义。事实上模式扩展可以让你完全抛开系统内置的 MVC 方式和核心类库~这也许是新版的魅力所在。

而作为新版的框架核心可组装的概念来说，主要体现在如下几个方面：

1、框架的目录结构和路径可定义；

2、框架的核心编译文件列表可定义；

3、框架的 MVC 可定义（也就是模式扩展了~）

模式扩展完全可以通过在项目里面定义自己的核心编译文件列表的方式来取代，因为其效果是等效的。只不过一个是可以通过设置模式来运行，而另外一个则是读取项目自身的定义，而且项目自身定义的优先级大于模式设置。

模式扩展主要是定义一个模式加载文件，例如简洁模式的 thin.php 文件定义方式如下：

```
return array(
    THINK_PATH.'/Lib/Think/Exception/ThinkException.class.php',// 异常处理
    THINK_PATH.'/Lib/Think/Core/Log.class.php',// 日志处理
    THINK_PATH.'/Mode/Thin/App.class.php', // 应用程序类
    THINK_PATH.'/Mode/Thin/Action.class.php',// 控制器类
    THINK_PATH.'/Mode/Thin/alias.php', // 加载别名
);
```

通过这样的定义，简洁模式加载了自己定义的 App、Action 和 Model 类库（在 alias 中定义了 Model 的别名加载），并且沿用了核心自带的异常和日志处理类库。最后一个别名定义文件是可选的~用于快速加载类库文件，这也是新版的一个功能改进。

仔细看了简洁模式的 App 类定义和 Model 定义后，就大概明白了简洁模式的工作原理了，作了很多功能上的简化。

项目核心列表文件位于项目的配置目录（默认是 Conf），名称为 core.php。其定义方式和模式扩展文件类似，区别在于核心文件的位置不同。

如果我们需要为自己的项目量身定制一个底层框架核心，又不想改变项目的入口文件定义，那么还可以根据自己的要求来进行项目核心扩展。

例如，我们在项目的配置目录下面增加一个 core.php 文件，定义如下：

```
return array(
    LIB_PATH.'/Mode/App.class.php', // 应用程序类
    LIB_PATH.'/Mode/Action.class.php',// 控制器类
    LIB_PATH.'/Mode/alias.php', // 加载别名
);
```

然后就是根据自己的实际需要，在项目的应用类库目录下面创建 Mode 目录，放入相应的定制后的 App、Action 和 Model 类，包括异常处理和日志类都是可以替换的。

定义完成后，需要清空核心编译缓存和项目编译缓存才能生效。

7 模板指南

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；
- ✧ 支持文件包含和布局模板；
- ✧ 支持多级标签嵌套；
- ✧ 一次编译多次运行，编译和运行效率非常高；
- ✧ 模板文件更新，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；
- ✧ 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/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

果开启页面 Trace 功能的话，可以在 Trace 信息里面看到当前页面对应的模板缓存文件名。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：普通标签主要用于输出变量和做一些基本的操作；XML 标签除了包含了普通标签的所有功能外，还可以完成一些逻辑判断、控制和循环输出，但是在变量输出上，普通标签具有简洁明了的优势。

例如：{$name} 看起来比 <var name="name" /> 更加容易使用，但是在控制和判断方面，XML 标签却有着普通标签无法替代的作用。

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。

7.1 变量输出

我们已经知道了在 Action 中使用 assign 方法可以给模板变量赋值，赋值后怎么在模板文件中输出变量的值呢？

如果我们在 Action 中赋值了一个 name 模板变量：

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

使用内置的模板引擎输出变量，只需要在模版文件使用：

```
{$name}
```

模板编译后的结果就是

```
<?php echo($name);?>
```

最后运行的时候就可以在标签位置显示 ThinkPHP 的输出结果。

注意模板标签的{和$之间不能有任何的空格，否则标签无效。

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：

```
<{$name}>
```

后面的内容我们都以默认的标签定义来说明。

assign 方法里面的第一个参数才是模板文件中使用的变量名称。如果改成下面的代码：

```
$name = 'ThinkPHP';
$this->assign('name2',$name);
```

再使用{$name} 输出就无效了，必须使用 {$name2} 才能输出模板变量的值了。

如果我们需要把一个用户数据对象赋值给模板变量：

```
$User = M('name');
$user = $User->find(1);
$this->assign('user',$user);
```

也就是说$user 其实是一个**数组变量**，我们可以使用下面的方式来输出相关的值：

```
{$user['name']} // 输出用户的名称
{$user['email']} // 输出用户的 email 地址
```

如果$user 是一个对象而不是数组的话，

```
$User = M('name');
```

```
$User->find(1);
$this->assign('user',$User);
```

可以使用下面的方式输出相关的属性值：

```
{$user:name} // 输出用户的名称
{$user:email}  // 输出用户的 email 地址
```

为了方便模板定义，还可以支持点语法，例如，上面的

```
{$user['name']} // 输出用户的名称
{$user['email']} // 输出用户的 email 地址
```

可以改成

```
{$user.name}
{$user.email}
```

因为点语法默认的输出是数组方式，所以上面两种方式是在没有配置的情况下是等效的。我们可以通过配置 **TMPL_VAR_IDENTIFY** 参数来决定点语法的输出效果，以下面的输出为例：

```
{$user.name}
```

如果 TMPL_VAR_IDENTIFY 设置为 array，那么

{$user.name}和{$user['name']}等效，也就是输出数组变量。

如果 TMPL_VAR_IDENTIFY 设置为 obj，那么

{$user.name}和{$user:name}等效，也就是输出对象的属性。

如果 TMPL_VAR_IDENTIFY 留空的话，系统会自动判断要输出的变量是数组还是对象，这种方式会一定程度上影响效率，而且只支持二维数组和两级对象属性。

如果是多维数组或者多层对象属性的输出，可以使用下面的定义方式：

```
{$user.sub.name}  // 使用点语法输出
```

或者使用

```
{$user['sub']['name']} // 输出三维数组的值
{$user:sub:name} // 输出对象的多级属性
```

7.2 使用函数

仅仅是输出变量并不能满足模板输出的需要，内置模板引擎支持对模板变量使用调节器和格式化功能，其实也就是提供函数支持，并支持多个函数同时使用。用于模板标签的函数可以是 PHP 内置函数或者是用户自定义函数，和 smarty 不同，用于模板的函数不需要特别的定义。

模板变量的函数调用格式为：

```
{$varname|function1|function2=arg1,arg2,### }
```

说明：

{ 和 $ 符号之间不能有空格 ，后面参数的空格就没有问题

###表示模板变量本身的参数位置

支持多个函数，函数之间支持空格

支持函数屏蔽功能，在配置文件中可以配置禁止使用的函数列表

支持变量缓存功能，重复变量字串不多次解析

使用例子：

```
{$webTitle|md5|strtoupper|substr=0,3}
```

编译后的 PHP 代码就是：

<?php echo (substr(strtoupper(md5($webTitle)),0,3)); ?>

注意函数的定义和使用顺序的对应关系，通常来说函数的第一个参数就是前面的变量或者前一个函数使用的结果，如果你的变量并不是函数的第一个参数，需要使用定位符号，例如：

{$create_time|date="y-m-d",###}

编译后的 PHP 是：

<?php echo (date("y-m-d",$create_time)); ?>

函数的使用没有个数限制，但是可以允许配置 TMPL_DENY_FUNC_LIST 定义禁用函数列表，系统默认禁用了 exit 和 echo 函数，以防止破坏模板输出，我们也可以增加额外的定义，例如：

TMPL_DENY_FUNC_LIST=>"echo,exit,halt"

多个函数之间使用半角逗号分隔即可。

并且还提供了在模板文件中直接调用函数的快捷方法，无需通过模板变量，包括两种方式：

1、执行方法并输出返回值：

格式：{:function(...)}

例如，输出 U 方法的返回值：

{:U('User/insert')}

编译后的 PHP 代码是

<?php echo U('User/insert');?>

2、执行方法但不输出：

格式：{~function(...)}

例如，调用 say_hello 函数：

{~say_hello('ThinkPHP')}

编译后的 PHP 代码是：

<?php say_hello('ThinkPHP');?>

7.3 系统变量

除了常规变量的输出外，模板引擎还支持系统变量和系统常量、以及系统特殊变量的输出。它们的输出不需要事先赋值给某个模板变量。系统变量的输出必须以**$Think.** 打头，并且仍然可以支持使用函数。

1、系统变量：包括 server、session、post、get、request、cookie

{$Think.server.script_name } // 输出$_SERVER 变量

{$Think.session.session_id|md5 } // 输出$_SESSION 变量

{$Think.get.pageNumber } // 输出$_GET 变量

{$Think.cookie.name } // 输出$_COOKIE 变量

支持输出$_SERVER、$_ENV、 $_POST、 $_GET、 $_REQUEST、$_SESSION 和 $_COOKIE 变量。后面的 server、cookie、config 不区分大小写，但是变量区分大小写。例如：

{$Think.server.script_name }和{$Think.SERVER.script_name }等效

SESSION 、COOKIE 还支持二维数组的输出，例如：

{$Think.CONFIG.user.user_name}

{$Think.session.user.user_name}

系统不支持三维以上的数组输出，请使用下面的方式输出。

以上方式还可以写成：

{$_SERVER.script_name } // 输出$_SERVER 变量

{$_SESSION.session_id|md5 } // 输出$_SESSION 变量

{$_GET.pageNumber } // 输出$_GET 变量

{$_COOKIE.name } // 输出$_COOKIE 变量

2、**系统常量** ：使用$Think.const 输出

{$Think.const.__SELF__ }

{$Think.const.MODULE_NAME }

或者直接使用

{$Think.__SELF__ }

{$Think.MODULE_NAME }

3、**特殊变量** ：由 ThinkPHP 系统内部定义的常量

{$Think.version } //版本

{$Think.now } //现在时间

{$Think.template|basename } //模板页面

{$Think.LDELIM } //模板标签起始符号

{$Think.RDELIM } //模板标签结束符号

4、**配置参数** ：输出项目的配置参数值

{$Think.config.db_charset}

输出的值和 C('db_charset') 的返回结果是一样的。

也可以输出二维的配置参数，例如：

{$Think.config.user.user_name}

5、**语言变量**：输出项目的当前语言定义值

{$Think.lang.page_error}

输出的值和 L('page_error')的返回结果是一样的。

7.4 快捷输出

为了使得模板定义更加简洁，系统还支持一些常用的变量输出快捷标签，包括：

{@var} //输出 Session 变量 和 {$Think.session.var} 等效

{#var} //输出 Cookie 变量 和 {$Think.cookie.var} 等效

{&var} //输出配置参数 和 {$Think.config.var} 等效

{%var} //输出语言变量 和 {$Think.lang.var} 等效

{.var} //输出 GET 变量 和 {$Think.get.var} 等效

{^var} //输出 POST 变量 和{$Think.post.var} 等效

{*var} //输出常量 和 {$Think.const.var} 等效

如果需要输出二维数组，例如 要输出$_SESSION[‘var1’][‘var2’]的值 快捷输出可以使用：

{@var1.var2} 的方式

同理

{#var1.var2}

可以输出 $_COOKIE[‘var1’][‘var2’]的值。

必须注意的是：快捷输出的变量不支持函数的使用。

所以，下面的用法是错误的：

{#var|strlen}

7.5 默认值输出

如果输出的模板变量没有值，但是我们需要在显示的时候赋予一个默认值的话，可以使用 default 语法，格式：

{$变量|default="默认值"}

这里的 default 不是函数，而是系统的一个语法规则，例如：

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

对系统变量的输出也可以支持默认值，例如：

{$Think.post.name|default="名称为空"}

因为快捷输出不支持使用函数，所以也不支持默认值，默认值支持 Html 语法。

7.6 包含文件

可以使用 Include 标签来包含外部的模板文件，使用方法如下：

1、 使用完整文件名包含

格式：<include file="完整模板文件名" />

例如：

<include file="./Tpl/default/Public/header.html" />

这种情况下，模板文件名必须包含后缀。使用完整文件名包含的时候，特别要注意文件包含指的是服务器端包含，而不是包含一个 URL 地址，也就是说 file 参数的写法是服务器端的路径，如果使用相对路径的话，是基于项目的入口文件位置。

2、包含当前模块的其他操作模板文件

格式：<include file="操作名" />

例如 导入当前模块下面的 read 操作模版：

<include file="read" />

操作模板无需带后缀。

3、 包含其他模块的操作模板

格式：<include file="模块名:操作名" />

例如，包含 Public 模块的 header 操作模版：

<include file="Public:header" />

4、包含其他模板主题的模块操作模板

格式：<include file="主题名@模块名:操作名" />

例如，包含 blue 主题的 User 模块的 read 操作模版：

<include file="blue@User:read" />

5、 用变量控制要导入的模版

格式：<include file="$变量名" />

例如

<include file="$tplName" />

给$tplName 赋不同的值就可以包含不同的模板文件，变量的值的用法和上面的用法相同。

6、使用快捷方式包含文件

格式：{include:模板文件规则}

其中的模板文件规则可以使用上面提到的 5 种方式。

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，除非缓存已经过期。如果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。

7.7 导入文件

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

例如：

```
<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" />
```

另外，系统提供了普通标签的方式加载外部 js 和 css 文件。

{load: __PUBLIC__/Js/Common.js}

{load: ../Public/Css/common.css }

7.8 Volist 标签

Volist 标签主要用于在模板中循环输出数据集或者多维数组。

通常模型的 select 和 findall 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 Action 中首先对模版赋值：

```
$User = M('User');
$list = $User->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}
</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}
```

```
</volist>
```

支持输出部分数据，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>
```

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>
```

volist 还有一个别名 iterate，用法和 volist 是一样。

7.9 Foreach 标签

foreach 标签也是用于循环输出

```
<foreach name="list" item="vo" >
{$vo.id}
{$vo.name}
</foreach>
```

Foreach 标签相对比 volist 标签简洁，没有 volist 标签那么多的功能。优势是可以对对象进行遍历输出，而 volist 标签通常是用于输出数组。

7.10 Switch 标签

模板引擎支持 Switch 标签，格式为：

```
<switch name="变量" >
<case value="值 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。

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。

7.11 比较标签

模板引擎提供了丰富的判断标签，比较标签的用法是：

```
<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：

- ✧ 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。

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：

举例说明：

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

7.12 Range 标签

Range 标签用于判断某个变量是否在某个范围之内，包括 in、notin 和 range 三个标签。

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" >输出内容 1</in>
```

如果判断不再某个范围内，可以使用：

```
<notin name="id" value="1,2,3" >输出内容 2</notin>
```

可以把上面两个标签合并成为：

```
<in name="id" value="1,2,3" >输出内容 1<else/>输出内容 2</in>
```

Value 属性的值可以使用变量，例如：

```
<in name="id" value="$var" >输出内容 1</in>
```

变量的值可以是字符串或者数组，都可以完成范围判断。

也可以直接使用 range 标签，替换 in 和 notin 的用法：

```
<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 或者 notin。

7.13 Present 标签

可以使用 present 标签来判断模板变量是否已经赋值，例如：

```
<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>
```

7.14 Empty 标签

可以使用 empty 标签判断模板变量是否为空，例如：

```
<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>
```

7.15 Defined 标签

可以使用 defined 标签判断常量是否已经有定义，例如：

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

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>
```

7.16 IF 标签

如果觉得上面的标签都无法满足条件判断要求的话，我们还可以使用 if 标签来定义复杂的条件判断，例如：

```
<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。

7.17 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（及其别名 iterate）、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。

默认的嵌套层次是 3 级，所以嵌套层次不能超过 3 层，如果需要更多的层次可以指定 TAG_NESTED_LEVEL 配置参数。

7.18 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：

第一种是使用 php 标签：

```
<php>echo 'Hello,world!';</php>
```

第二种就是直接使用原始的 php 代码：

```
<?php echo 'Hello,world!'; ?>
```

但是 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 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。

7.19 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。

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

7.20 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。

格式：**{/* 注释内容 */ } 或 {// 注释内容 }**

说明：在显示页面的时候不会显示模板注释，仅供模板制作的时候参考。

注意{和注释标记之间不能有空格。

例如：

{// 这是模板注释内容 }

{/* 这是模板

注释内容*/ }

模板注释支持多行可以。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。

7.21 引入标签库

前面我们所讲述的标签用法都是内置的标签库或者内置模板的用法，事实上，内置模板引擎的标签库是可以无限扩展和增加标签的，一旦你扩展和使用了新的标签库，就必须要告诉模板当前要使用的标签库名称，否则不会自动导入，防止以后标签库大量扩展后增加解析工作量，导入标签库使用 tagLib 标签。

格式：<tagLib name="标签库 1[,标签库 2,...]" />

可以同时导入多个标签库，用逗号分隔。

例如：

<tagLib name="html" />

表示在当前模板文件需要引入 html 标签库。要引入标签库必须确保有 Html 标签库的定义文件和解析类库（如何扩展这种方式请参考前面的标签库扩展部分）。Cx 标签库内置导入，无需使用 taglib 标签导入。

引入后，html 标签库的所有标签在当前模板页面中都可以使用了。外部导入的标签库必须使用标签库前缀的 xml 标签，避免两个不同的标签库中存在同名的标签定义，例如（假设 Html 标签库中已经有定义 select 和 link 标签）：

```
<html:select options='name' selected='value' />
<html:link href='/path/to/common.js' />
```

标签库使用的时候忽略大小写，因此下面的方式一样有效：

```
<HTML:LINK HREF='/path/to/common.js' />
```

如果你的每个模板页面都需要加载 Html 标签库的话，也可以通过配置直接预先加载 Html 标签库。

```
'TAGLIB_PRE_LOAD' => 'html' ,
```

如果有多个标签库需要预先加载的话，用逗号分隔。定义之后，每个模板页面都可以直接使用：

```
<html:select options='name' selected='value' />
```

而不需手动引入 Html 标签库。

假设你确信 Html 标签库无论在现在还是将来都不会和系统内置的标签库存在相同的标签，那么可以配置 TAGLIB_BUILD_IN 的值把 Html 标签库作为内置标签库引入，例如：

```
'TAGLIB_BUILD_IN' => 'cx,html' ,
```

这样，也无需在模板文件页面引入 Html 标签库了，并且可以不带前缀直接使用 Html 标签库的标签：

```
<select options='name' selected='value' />
```

注意，cx 标签库是系统内置标签库，不能删除定义。

7.22 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，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

{$name} {$vo.name}

必须使用

<{$name}> <{$vo.name}> 才能生效了。

普通模板标签主要用于模板变量输出、模板注释和公共模板包含。如果要使用其它功能，请使用 XML 模板标签，ThinkPHP 包含了一个基于 XML 和 TagLib 技术的模板标签，包含了普通模板有的功能，并且有一些方面的增强和补充，更重要的一点是新的标签库模板技术更加具有扩展性。新的 TagLib 标签

库具有命名空间功能，ThinkPHP 框架内置了 CX 标签库。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。

XML 模板标签可以用于模板变量输出、文件包含、模板注释、条件控制、循环输出等功能，而且完全可以自己扩展功能。

8 附录

8.1 常量参考

8.1.1 预定义常量

URL_COMMON=0

普通模式 URL

URL_PATHINFO=1

PATHINFO URL

URL_REWRITE=2

REWRITE URL

URL_COMPAT=3

兼容模式 URL

HAS_ONE=1

HAS_ONE 关联定义

BELONGS_TO=2

BELONGS_TO 关联定义

HAS_MANY=3

HAS_MANY 关联定义

MANY_TO_MANY=4

MANY_TO_MANY 关联定义

8.1.2 系统常量

__ROOT__ ： 网站根目录地址

__APP__ ： 当前项目（入口文件）地址

__URL__ ： 当前模块地址

__ACTION__ ： 当前操作地址

__SELF__ ： 当前 URL 地址

__CURRENT__ ： 当前模块的模板目录

ACTION_NAME ： 当前操作名称

APP_PATH ： 当前项目目录

APP_NAME ： 当前项目名称

APP_TMPL_PATH ： 项目模板目录

APP_PUBLIC_PATH ：项目公共文件目录

CACHE_PATH ： 项目模版缓存目录

CONFIG_PATH ：项目配置文件目录

COMMON_PATH ： 项目公共文件目录

DATA_PATH ： 项目数据文件目录

GROUP_NAME ：当前分组名称

HTML_PATH ： 项目静态文件目录

IS_APACHE ： 是否属于 Apache

IS_CGI ：是否属于 CGI 模式

IS_IIS ：是否属于 IIS

IS_WIN ：是否属于 Windows 环境

LANG_SET ： 浏览器语言

LIB_PATH ： 项目类库目录

LOG_PATH ： 项目日志文件目录

LANG_PATH ： 项目语言文件目录

MODULE_NAME ：当前模块名称

MEMORY_LIMIT_ON ： 是否有内存使用限制

MAGIC_QUOTES_GPC ： MAGIC_QUOTES_GPC

TEMP_PATH ：项目临时文件目录

TMPL_PATH ： 项目模版目录

THINK_PATH ： ThinkPHP 系统目录

THINK_VERSION ：ThinkPHP 版本号

TEMPLATE_NAME ：当前模版名称

TEMPLATE_PATH ：当前模版路径

VENDOR_PATH ： 第三方类库目录

WEB_PUBLIC_PATH ：网站公共目录

8.2 配置参考

这里列出了系统内置的惯例配置中的配置参数，所有参数在没有生效之前都可以在项目（包括模块）配置文件、Action 操作方法中被覆盖，这里只是列出了默认的惯例设置，并不代表你的应用设置。

下面的配置参数，**粗体标识的是配置参数名称 后面的值是惯例配置中的默认值。**

8.2.1 应用配置

APP_DEBUG = false

是否开启应用调试模式，默认关闭

APP_PLUGIN_ON = false

是否开启应用插件机制

APP_GROUP_DEPR = '.'

模块分组之间的分割符

APP_GROUP_LIST = ''

项目模块分组列表，多个组之间用逗号分隔，例如 'Admin,Home'

APP_DOMAIN_DEPLOY = false

是否使用独立域名部署项目，只有在项目目录本身就是网站根目录的情况下开启

APP_AUTOLOAD_REG=false

是否开启 SPL_AUTOLOAD_REGISTER，如果其他类库中使用了__autoload 方法的话需要开启

APP_AUTOLOAD_PATH='Think.Util.'

__autoLoad 的搜索路径，当前项目的 Model 和 Action 类会自动加载，无需设置，注意搜索顺序

APP_FILE_CASE = false

是否严格检查文件的大小写 （在 Windows 平台下面有效）

APP_CONFIG_LIST=array('taglibs','routes','tags','htmls','modules','actions')

项目扩展配置文件列表

8.2.2 URL 配置

URL 相关配置包括 URL 模式、路由等。

URL_DISPATCH_ON = true,

是否启用 Dispatcher，如果关闭，只能使用传统方式的参数传值。

URL_MODEL = 1

URL 模式： 0 普通模式 1 PATHINFO 2 REWRITE 3 兼容模式 当 URL_DISPATCH_ON 开启后有效

默认为 PATHINFO 模式，提供最好的用户体验和 SEO 支持

URL_PATHINFO_MODEL = 2

PATHINFO 模式，默认采用智能模式

普通模式 1 参数没有顺序/m/module/a/action/id/1

智能模式 2 自动识别模块和操作/module/action/id/1/ 或者 /module,action,id,1/...

URL_PATHINFO_DEPR = '/'

PATHINFO 参数之间分割号

URL_ROUTER_ON = true

是否开启 URL 路由

URL_HTML_SUFFIX= "

伪静态后缀设置，例如 .shtml

URL_CASE_INSENSITIVE = false

URL 是否不区分大小写，默认区分大小写

8.2.3 日志配置

LOG_RECORD = false

是否记录网站日志，默认不记录日志

LOG_RECORD_LEVEL = array('EMERG','ALERT','CRIT','ERR')

允许记录的日志级别

LOG_FILE_SIZE = 2097152

日志文件大小限制， 针对文件方式的日志记录，超过会自动生成备份文件

8.2.4 错误配置

ERROR_MESSAGE = '您浏览的页面暂时发生了错误！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，需要填写完整的 URL 地址

8.2.5 数据库配置

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，分布式数据库方式下面有效

DB_FIELDS_CACHE = true

开启数据表字段缓存

DB_TYPE = 'mysql'

数据库类型

DB_HOST = 'localhost'

数据库服务器地址

DB_NAME = ''

数据库名称

DB_USER = 'root'

数据库用户名

DB_PWD = ''

数据库 密码

DB_PORT = 3306

数据库使用的端口

DB_PREFIX = 'think_'

数据库的表前缀

DB_SUFFIX = ''

数据库的表后缀

DB_FIELDTYPE_CHECK = false

是否进行字段类型检查

8.2.6 静态缓存配置

HTML_FILE_SUFFIX= '.shtml'

默认静态文件后缀

HTML_CACHE_ON = false

默认关闭静态缓存

HTML_CACHE_TIME= 60

静态缓存有效期

HTML_READ_TYPE= 1

静态缓存读取方式 0 readfile 1 redirect

8.2.7 数据缓存配置

DATA_CACHE_TYPE= 'File'

数据缓存类型 支持 File Db Apc Memcache Shmop Sqlite Xcache Apachenote Eaccelerator

DATA_CACHE_PATH = TEMP_PATH

缓存路径设置 (仅对 File 方式缓存有效)

DATA_CACHE_TIME= -1

数据缓存有效期

DATA_CACHE_COMPRESS= false

数据缓存是否压缩缓存

DATA_CACHE_CHECK= false

数据缓存是否校验缓存

DATA_CACHE_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

8.2.8 运行时间配置

SHOW_RUN_TIME=false

运行时间显示

SHOW_ADV_TIME=false

显示详细的运行时间，SHOW_RUN_TIME 开启后有效

SHOW_DB_TIMES=false

显示数据库读写次数

SHOW_CACHE_TIMES=false

显示缓存读写次数

SHOW_USE_MEM=false

显示内存开销

SHOW_PAGE_TRACE= false

显示页面 Trace 信息 由 Trace 文件定义和 Action 操作赋值

SHOW_ERROR_MSG = true

发生错误时显示错误信息

8.2.9 模板配置

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_CACHFILE_SUFFIX='.php'

默认模板缓存后缀

TMPL_PARSE_STRING= ''

模板引擎要自动替换的字符串，必须是数组形式。例如 array('__MYPATH__'=>Lib_PATH,...)

TMPL_ACTION_ERROR ='Public:success'

错误跳转模板文件

TMPL_ACTION_SUCCESS ='Public:success'

成功跳转模板文件

TMPL_TRACE_FILE =THINK_PATH.'/Tpl/PageTrace.tpl.php'

页面 Trace 的模板文件

TMPL_EXCEPTION_FILE =THINK_PATH.'/Tpl/ThinkException.tpl.php'

异常页面的模板文件

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_DENY_FUNC_LIST='echo,exit'

模板引擎禁用函数

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

TMPL_VAR_IDENTIFY = 'array'

模板变量识别 留空自动判断 array 数组 obj 对象

TMPL_FILE_DEPR='/'

模板文件 MODULE_NAME 与 ACTION_NAME 之间的分割符，只对项目分组部署有效

TMPL_STRIP_SPACE = false

是否去除模板文件里面的 html 空格与换行

TMPL_CACHE_ON=true

默认开启模板编译缓存 false 的话每次都重新编译模板

TMPL_CACHE_TIME= -1

模板缓存有效期 -1 永久 单位为秒

TAGLIB_BEGIN='<'

标签库标签开始标记

TAGLIB_END= '>'

标签库标签结束标记

TAGLIB_LOAD = true

是否使用内置标签库之外的其它标签库，默认进行自动检测

TAGLIB_BUILD_IN = 'cx'

内置标签库名称 可以添加自己的标签库，多个使用逗号分隔

TAGLIB_PRE_LOAD = ''

预先加载的标签库，无需在每个模板使用 taglib 标签加载， 多个使用逗号分隔

TAG_NESTED_LEVEL= 3

标签嵌套级别

8.2.10 Cookie 设置

COOKIE_EXPIRE = 3600

Coodie 有效期

COOKIE_DOMAIN = ''

Cookie 有效域名

COOKIE_PATH='/'

Cookie 路径

COOKIE_PREFIX= ''

Cookie 前缀 避免冲突

8.2.11 令牌验证配置

TOKEN_ON = true

是否开启令牌验证

TOKEN_NAME = '__hash__'

令牌验证的表单隐藏字段名称

TOKEN_TYPE = 'md5'

令牌验证哈希规则

8.2.12 默认值配置

DEFAULT_APP = '@'

默认模型类所在的项目名称 @ 表示当前项目

DEFAULT_GROUP = 'Home'

默认分组

DEFAULT_MODULE= 'Index'

默认模块名称

DEFAULT_ACTION = 'index'

默认操作名称

DEFAULT_THEME= 'default'

默认模板主题名称

DEFAULT_LANG = 'zh-cn'

默认语言

DEFAULT_TIMEZONE='PRC'

默认时区

DEFAULT_AJAX_RETURN= 'JSON'

AJAX 数据返回格式 JSON XML ...

DEFAULT_CHARSET= 'utf-8'

默认页面输出编码

8.2.13 系统变量配置

下面这些变量配置主要用于 URL 的特殊传值，在项目中的 URL 和表单参数尽量不要与之冲突，否则容易造成错误。

VAR_PATHINFO ='s'

PATHINFO 兼容模式获取变量例如 ?s=/module/action/id/1

VAR_GROUP = 'g'

默认分组变量

VAR_ROUTE='r'

默认路由获取变量

VAR_MODULE='m'

默认模块获取变量

VAR_ACTION='a'

默认操作获取变量

VAR_PAGE='p'

默认分页跳转变量

VAR_TEMPLATE ='t'

默认模板切换变量

VAR_LANGUAGE='l'

默认语言切换变量

VAR_AJAX_SUBMIT='ajax'

默认的 AJAX 提交变量

8.2.14 语言和时区

LANG_SWITCH_ON= false

是否开启多语言功能，默认关闭

LANG_AUTO_DETECT = true

是否自动侦测浏览器语言

8.3 函数参考

8.3.1 系统函数库

系统函数库的方法无需导入，可以直接使用，下面按照字母排序列出每个函数及其参数。

A(name, app='@')

实例化 Action， name 表示 Actin 名称 app 表示项目名，默认是当前项目

返回实例化后的 Action 对象，如果对应的 Action 类不存在则返回 false

auto_charset(contents, from, to)

编码转换，把 contents 从 from 转换到 to，contents 支持字符串和数组

B(name)

调用行为 name 表示行为名称

C(name=null,value=null)

获取和设置配置定义，获取已有的配置值 C('name') 新增或者更改设置 C('name','value') 如果 name 的值是数组，表示批量赋值

cookie(name,value='',option=null)

Cookie 设置、获取、清除 name

D(name='',app='')

实例化模型类，name 表示模型的名称，如果留空，表示实例化空模型，app 表示项目名 默认是当前项目。跨项目实例化的话，项目目录必须保证同级。返回实例化后的 Model 类，如果对应的模型类不存在，则抛出异常

debug_start(label='')

区间调试开始，label 表示区间的标签名，例如 debug_start('read')

debug_end(label='')

区间调试结束，label 表示区间的标签名，必须和 debug_start 对应才能输出正确

dump(var, echo=true,label=null, strict=true)

浏览器友好的变量输出，var 支持任何变量，echo 表示是否需要输出，如果为否，则返回要显示的字符串。Strict 表示是否输出详细信息，如果为否，使用 print_r 输出，如果为是，使用 var_dump 输出。Dump 函数还支持 xdebug 扩展

F(name,value='',path=DATA_PATH)

快速文件数据读取和保存 针对简单类型数据 字符串、数组，F 方法不支持缓存有效期 name 为缓存名称（也就是缓存文件名），value 表示缓存值，如果为 NULL 表示删除缓存，留空表示获取缓存值，其他情况表示设置缓存，path 表示缓存文件所在路径，默认是项目的数据目录

file_exists_case(filename)

区分大小写的文件存在判断，只在 Windows 环境下面有效，Linux 环境本身就区分大小写

get_instance_of(name,method='',args=array())

单例化某个类，同时可以传入类的方法和参数

halt (error)

输出错误并中止执行

import(class,baseUrl = '',ext='.class.php')

基类库或者应用类库导入，class 表示要导入的类库，采用命名空间的方式，例如：import('Think.Util.Session') baseUrl 表示导入的基础路径，留空的话系统有默认的规则，除了 Think、ORG 和 Com 类库包位于系统基类库目录外，其他方式都位于项目类库目录，ext 表示类库后缀，默认是.class.php

L(name=null,value=null)

获取和设置语言定义 name 表示语言变量名，value 表示语言定义值，留空为获取语言定义

M(name='',class='Model')

快速实例化模型类，name 表示模型名称，但无需创建具体模型类（需要有对应的数据表即可），class 表示要实例化的类名，默认是 Model 类，也可以使用其他模型类，例如 M('User', 'AdvModel') 实例化高级模型类的用户模型

mk_dir(dir, mode = 0755)

循环创建目录，如果 dir 参数中存在未创建的多级目录，会自动依次创建，mode 表示目录的权限

R(module,action,app='@')

远程调用模块的操作方法，module 是模块名，action 是操作名，app 是项目名，默认是当前项目

require_cache(filename)

优化的 require，可以替代 require 和 require_once 函数

redirect(url,time=0,msg='')

URL 重定向， url 必须是一个完整的 URL 地址，time 表示等待时间，单位为秒，msg 表示等待的提示信息

S(name,value='',expire='',type='')

缓存设置或者读取 name 表示缓存名称，value 表示缓存的值，如果留空则表示获取缓存值，如果为 NULL 表示删除缓存，否则表示设置缓存，expire 表示有效期，单位为秒，type 是使用的缓存类型，包括 File、APC、Db、Memcache、Shmop、Eaccelerator、Sqlite 和 Xcache

tag(name,params=array())

执行某个标签的方法，用于应用扩展的标签执行

throw_exception(msg,type='ThinkException',code=0)

抛出异常，msg 为错误信息，type 为异常类型 code 为异常代码

to_guid_string(mix)

根据 PHP 各种类型变量生成唯一标识号

U (url,params=array(),redirect=false,suffix=true)

根据当前 URL 配置生成 URL 地址 并支持跳转

url 表示 URL 规则，支持

'[项目://][路由@][分组名-模块/]操作'

和 '[项目://][路由@][分组名-模块/]操作?参数 1=值 1&参数 2=值 2'

params 表示参数，必须使用数组传入

redirect 表示是否需要跳转到生成的 URL 地址

suffix 表示是否添加伪静态后缀，设置了伪静态后有效

vendor(class,baseUrl = '',ext='.php')

导入第三方类库，用法同 import，只是 baseUrl 的默认值位于项目的第三方类库目录，ext 后缀默认为.php

W(name,data=array(),return=false)

输出 Widget，name 为 Widget 名称 data 表示传入的参数，必须使用数组，return 表示是否需要返回结果，默认直接输出

xml_encode(data,encoding='utf-8',root="think")

把数组转换成 XML，用于内置的 AJAX 返回 XML 格式的数据

8.3.2 兼容函数库

兼容函数库会在 PHP 版本低于 **5.2.0** 的时候自动加载，并且可以直接调用

json_encode(data)

把 PHP 数据编码成 JSON 格式字符串，方便客户端 JS 调用

json_decode(json,assoc=false)

把 JSON 格式字符串解码成 PHP 数据格式 assoc 表示返回关联数组

property_exists(class, property)

检查对象或类是否具有该属性

8.3.3 编译函数库

编译函数库会在系统内部的编译过程中自动加载，实际运行过程无法调用。

compile(filename,runtime=false)

编译文件，filename 要编译的文件名，runtime 表示是否需要替换预编译指令，返回编译后的字符串

strip_whitespace(content)

去除代码中的空白和注释

array_define(array)

根据数组生成常量定义，array 是一个关联数组

build_runtime()

生成核心编译缓存

mkdirs(dirs,mode=0777)

批量创建目录 dirs 表示目录列表数组 mode 表示目录权限

build_app_dir()

创建项目目录结构，默认会生成项目目录结构，并自动生成空白的项目配置文件和测试 Action 类，该方法不会覆盖已经存在的目录或者文件，支持安全文件写入

check_runtime()

检查 runtime 运行目录，并检查下面的 Cache（模板缓存）、Temp（数据缓存）、Data（数据目录）和 Logs（日志文件）子目录，如果不存在则自动创建

8.3.4 扩展函数库

扩展函数库的方法不能直接使用，需要加载或者拷贝到项目函数库中才能使用。

加载扩展函数库，使用：

Load('extend');

加载扩展函数库后，就可以调用其中的所有函数了。

get_client_ip()

获取客户端的 IP 地址

msubstr(str, start=0, length, charset="utf-8", suffix=true)

中文字符串截取

rand_string(len=6,type='',addChars='')

产生随机字串

type 是随机类型，包括：

0 字母 1 数字 2 大写字母 3 小写字母 4 中文 5 混合（去掉了容易混淆的字符 oOLl 和数字 01）

addChars 附加的字符串

build_verify (length=4,mode=1)

创建随机验证码，mode 参数用法和 rand_string 的 type 一致

byte_format(size, dec=2)

字节格式化 把字节数格式为 B K M G T 描述的容易理解的大小

is_utf8(string)

检测字符串是否是 utf8 编码

highlight_code(str,show=false)

代码高亮

h(text, tags = null)

输出安全的 Html 代码

ubb(Text)

基本的 UBB 解析

build_count_rand (number,length=4,mode=1)

随机生成一组字符串

remove_xss(val)

移除 Html 代码中的 XSS 攻击

list_to_tree(list, pk='id',pid = 'pid',child = '_child',root=0)

把查询的数据集转换成树形列表数组， list 表示查询的数据集（数组），pk 表示主键名，pid 表示父键名，child 表示子列表的名称，默认是_child，root 表示跟节点的主键值

list_sort_by(list,field, sortby='asc')

对查询的数据集排序，list 表示查询的结果数据集（数组），field 表示要排序的字段名称，sortby 表示排序类型，包括 asc 正向排序 desc 逆向排序 nat 自然排序，默认为 asc

list_ search(list,condition)

在查询的数据集中搜索数据，list 表示查询的结果数据集（数组），condition 表示查询条件，支持支持下面的查询方式

数组方式如 array('var1'=>'value1', 'var2'=>'value2') 并且支持正则表达式 array('name'=>'/[A-Z]/')

URL 方式如 var1=value1&var2=value2

send_http_status(status)

发送 http 状态信息,status 表示 http 状态值，例如 302、404

8.4 类库参考

这里只是列出了常用的类库的一些公共方法，详细的使用请参考 API 手册。

8.4.1 Think 类

__set(name ,value) 设置类的属性值，魔术方法

__get(name) 获取类的属性值，魔术方法

autoload(classname) 自动加载方法，用于开启 APP_AUTOLOAD_REG 的情况下

instance(class,method='') 实例化类，静态方法

8.4.2 Action 类

getActionName() 获取当前 Action 的名称

isAjax() 是否为 AJAX 请求

isPost() 是否为 POST 请求

isGet() 是否为 GET 请求

isPut() 是否为 PUT 请求

isDelete() 是否为 DELETE 请求

isHead() 是否为 HEAD 请求

display(templateFile='',charset='',contentType='text/html') 输出模板

templateFile 模板文件

charset 输出编码

contentType 输出类型

fetch(templateFile='',charset='',contentType='text/html') 获取模板输出内容

buildHtml(htmlfile='',templateFile='',charset='',contentType='text/html') 生成静态页面

assign(name,value='') 模板变量赋值

__set(name,value) 模板变量赋值，魔术方法

get(name) 获取模板变量

trace(name,value='') Trace 变量赋值

error(message,ajax=false) 错误跳转

success(message,ajax=false) 成功跳转

ajaxReturn(data,info='',status=1,type='') AJAX 返回数据到客户端

redirect(url,params=array(),delay=0,msg='') Action 重定向

8.4.3 View 类

assign(name,value='') 模板变量赋值

trace(name,value='') Trace 变量赋值

get(name) 获取模板变量

display(templateFile='',charset='',contentType='text/html') 输出模板

fetch(templateFile='',charset='',contentType='text/html') 获取模板输出内容

buildHtml(htmlfile='',templateFile='',charset='',contentType='text/html') 生成静态页面

8.4.4 Model 类

getModelName() 获取当前 Model 的名称

getTableName() 获取当前 Model 的数据表名称

switchModel(type,vars=array()) 动态切换模型

table() 设置当前操作的数据表

field() 设置要查询的数据字段

where() 设置查询或者操作条件

data(data) 设置数据对象

order(order) 设置排序

limit(limit) 查询限制

page(page) 查询分页

join(join) 进行 JOIN 查询

having(having) 进行 having 查询

group(group) 进行 group 查询

lock(lock) 查询锁定

distinct(distinct) 唯一性查询

count(field) 记录统计

sum(field) 总数查询

min(field) 最小值查询

max(field) 最大值查询

avg(field) 平均值查询

_initialize() 模型初始化方法

_facade(data) 对保存到数据库的数据进行处理

_before_write(&data) 写入数据前的回调方法 包括新增和更新

add(data='',options=array()) 新增数据

_before_insert(&data,options) 写入数据前的回调方法

_after_insert(data,options) 写入数据后的回调方法

selectAdd(fields='',table='',options=array()) 通过 Select 方式添加记录

save(data='',options=array()) 更新数据到数据库

_before_update(&data,options) 更新数据前的回调方法

_after_update(data,options) 更新成功后的回调方法

delete(options=array()) 删除数据

_after_delete(data,options) 删除成功后的回调方法

select(options=array()) 查询数据集

_after_select(&resultSet,options) 查询成功后的回调方法

findAll(options=array()) select 方法的别名

_options_filter(&options) 表达式过滤回调方法

find(options=array()) 查询数据

_after_find(&result,options) 查询成功的回调方法

setField(field,value,condition='') 设置记录的某个字段值

setInc(field,condition='',step=1) 字段值增长

setDec(field,condition='',step=1) 字段值减少

getField(field,condition='',sepa=' ') 获取某个字段值

create(data='',type='') 创建数据对象

autoCheckToken(data) 表单令牌验证

query(sql) 执行原生 SQL 查询

execute(sql='') 执行原生 SQL 操作

startTrans() 启动事务

commit() 提交事务

rollback() 事务回滚

getError() 获取模型的错误信息

getDbError() 获取数据库的错误信息

getLastInsID() 获取最后执行的 SQL 语句

getPk() 获取主键名称

getDbFields() 获取数据表的字段信息

regex(value,rule) 使用正则验证数据

setProperty(name,value) 设置模型的属性值

高级模型类 AdvModel

topN(count,options=array()) 查询满足条件的前 N 个记录

getN(position=0,options=array()) 查询符合条件的第 N 条记录

0 表示第一条记录 -1 表示最后一条记录

first(options=array()) 获取满足条件的第一条记录

last(options=array()) 获取满足条件的最后一条记录

returnResult(data,type='') 返回指定的数据类型

setLazyInc(field,condition='',step=1,lazyTime=0) 字段值延迟增长

setLazyDec(field,condition='',step=1,lazyTime=0) 字段值延迟减少

addConnect(config,linkNum=NULL) 增加数据库连接

delConnect(linkNum) 删除数据库连接

closeConnect(linkNum) 关闭数据库连接

switchConnect(linkNum,name='') 切换数据库连接

patchQuery(sql=array()) 批处理执行 SQL 语句

getPartitionTableName(data=array()) 得到分表的的数据表名

8.4.5 Log 类

record(message,level=self::ERR,record=false) 记录日志信息到内存

save(type=self::FILE,destination='',extra='') 保存记录的日志信息

write(message,level=self::ERR,type=self::FILE,destination='',extra='') 直接写入日志信息

8.4.6 Db 类

getInstance() 获取数据库实例，静态方法

addConnect(config,linkNum=null) 添加数据库连接

switchConnect(linkNum) 切换数据库连接

insert(data,options=array()) 新增数据

selectInsert(fields,table,options=array()) Select 方式写入数据

update(data,options) 更新数据

delete(options=array()) 删除数据

select(options=array()) 查询数据

getLastSql() 获取最后执行的 SQL 语句

getError() 获取错误信息

数据库驱动类库

connect(config='',linkNum=0) 连接数据库

free() 释放查询结果

query(str) 数据查询

execute(str) 执行语句

startTrans() 启动事务

commit() 事务提交

rollback() 事务回滚

getFields(tableName) 获取数据表字段信息

getTables(dbName='') 获取数据库的表信息

close() 关闭数据库

error() 获取错误信息

escape_string(str) 安全过滤

8.4.7 Cache 类

getInstance() 获得缓存实例，静态方法

connect(type='',options=array()) 连接缓存

setOptions(name,value) 设置缓存属性

getOptions(name) 获取缓存设置

__get(name) 获取缓存，魔术方法

__set(name,value) 设置缓存，魔术方法

__unset(name) 删除缓存，魔术方法

缓存驱动类

get(name) 获取缓存

set(name,value,expire='') 设置缓存

rm(name) 删除缓存

8.4.8 Cookie 类

is_set(name) 判断 Cookie 是否存在 静态方法

get(name) 获取 Cookie 的值 静态方法

set(name,value,expire='',path='',domain='') 设置 Cookie 的值 静态方法

delete(name) 删除 Cookie 的值 静态方法

clear() 清空 Cookie 的值

8.4.9 Session 类

get(name) 获取 Session 值

getLocal(name) 获取本地化 Session 的值

set(name, value) 设置 Session 的值

setLocal(name, value) 设置本地化 Session 的值

is_set(name) 检查 Session 的值是否设置

is_setLocal(name) 检查本地化 Session 的值是否已经设置

pause() 暂停 session

clear() 清空 Session

clearLocal() 清空本地化 Session

destroy() 销毁 Session

isExpired() 检测 Session 是否过期

isIdle() 检测 Session 是否闲置

8.4.10 Debug 类

mark(name) 标记调试位置，静态方法

useTime(start,end,decimals = 6) 调试区间所用的时间，静态方法

useMemory(start,end) 调试区间所用的内存，静态方法

getMemPeak(start,end) 调试区间的内存占用峰值，静态方法

8.5 关于升级

从 1.6RC1 版本升级

1.6RC1 版本升级到 2.0 版本需要做如下的改动：

1、配置参数改动（参考下载包中的配置对照表）

2、其他的可以参考代码重构部分进行适当的调整

从 1.5 版本升级

从 1.5 版本升级到 2.0 版本可以使用兼容模式运行，请按照下面的步骤进行升级：

1、下载官方发布的兼容模式扩展，解压后放入系统目录的 Mode 目录下面；

2、修改项目的入口文件，在加载 ThinkPHP.php 公共文件之前增加下面一行：

```
define('THINK_MODE','Compat');  // 设置为兼容模式运行
```

3、然后，把原先系统基类库的 ORG 类库包放入新版的基类库目录中。

4、清除项目编译缓存文件~runtime.php 和 ~app.php。

如果你的版本低于 1.5，请首先按照官方的发布说明（http://thinkphp.cn/Blog/12）升级到 1.5 版本，然后再按照上面的说明升级到 2.0 版本。

8.6 代码重构

1.5 版本通过兼容模式可以运行在新版下面，但是使用的核心仍然是 1.5 版本的核心。因为新版的内核是完全重构的，接口有所改变，所以如果需要完全迁移到新的 2.0 版本，只有通过代码重构的方式实现，下面是 1.5 的代码重构到 2.0 版本的相关建议：

- 参数方式的数据查询改成连贯操作，例如：

  $Model->findall('id>1','id,name','id desc','10,100');

  应该改成

  $Model->where('id>1')->order('id desc')->field('id,name')->limit('10,100')->select();

- 布局方法 layout 用 display 替换（参考布局模板部分内容）；
- URL 方法换成 U 方法（用法也需要改变 参考 URL 生成部分内容）；
- 部分查询方法需要继承高级模型类 AdvModel 才可以使用，包括 topN getN first last ；
- 原来 Model 的文本字段和乐观锁功能需要继承 AdvModel 才可使用；
- Action 类的 redirect 方法接口因为 URL 方法的改变也有所改变；
- Model 类的 addAll，deleteAll 方法已经删除；
- Model 类的 getFields 方法请改成 getField；
- Model 类的 deleteBy 动态方法删除；
- Model 类的自动验证和自动完成定义的时间定义格式改变（参考自动验证和自动完成部分内容）；
- 视图模型和关联模型已经分离出 Model 类，需要另外继承（参考视图模型和关联模型部分内容）；
- 数据库延迟查询功能已经取消；
- RBAC 也已经分离出核心，需要自己调用；
- 模板的点语法默认是数组输出了；
- Session 可以设置是否需要自动开启；

- 系统函数库删除了一些核心不依赖的函数，移入了扩展函数库 extend.php 需要的话自行放入项目函数库中即可；
- import 方法的匹配导入和子目录导入由于性能问题不再支持；
- 浏览器防刷新功能已经删除（改由行为扩展提供）；
- URL 伪装功能已经删除；
- 编码自动转换功能已经取消，请自行用 auto_charset 转换编码；
- 模板文件中导入外部 js 和 css 不需要导入 Html 标签库；
- Html 标签库已经作为扩展提供，不再内置；
- 模板引擎的 sublist、resultset 和 subeach 标签已经取消，分别使用 volist 和 foreach 替代；

8.7 开源应用

基于 ThinkPHP 最新版本的部分开源应用列表：

- ThinkSNS：开源 SNS 系统
- ThinkCMS：开源 CMS 系统
- ThinkHost：开源虚拟主机管理系统
- MiniBlog：开源迷你博客系统
- ThinkMyAdmin：开源 MySQL 管理工具
- yBlog：开源博客系统
- shuguangCMS：开源 CMS 系统

更多开源应用请关注官方网站。

8.8 典型案例

基于 ThinkPHP 的典型案例参考请访问官方网站：http://thinkphp.cn/Case/

8.9 大事记

ThinkPHP 发展历程，无数 TPer 一起见证了 ThinkPHP 的成长：

2006-01-15 ThinkPHP 的雏形版本 FCS0.6.0 发布

2006-02-12 （元霄节）发布 FCS 0.6.1 版本，Google 讨论组成立

2006-03-15 FCS 0.7.0 版本发布

2006-03-23 第一个 QQ 群成立

2006-05-07 FCS 0.8 版本发布

2006-10-25 FCS 0.9.0 版本发布

2006-12-25 SF 项目和 Google 网站 ThinkPHP 项目申请完成

2007-01-01 FCS 正式更名为 ThinkPHP

2007-01-08 ThinkPHP 0.9.5 版发布 同期官方网站 http://ThinkPHP.cn 开通

2007-02-21 TOPThink 社区暨新版 ThinkPHP 官方网站开通，并提供社区支持

2007-02-25 发布 ThinkPHP 0.9.6 版本，完成 FCS 到 ThinkPHP 的正式迁移

2007-04-29 ThinkPHP 发布 0.9.7 版本

2007-07-01 ThinkPHP 发布 0.9.8 版本

2007-10-15 ThinkPHP 发布 1.0.0RC1 版本，完成 PHP5 的重构

2007-12-15 ThinkPHP 发布 1.0.0 正式版本 标志着 ThinkPHP 步入轨道

2008-10-01 ThinkPHP 发布 1.0.3 正式版本

2008-12-25 ThinkPHP 发布 1.5 正式版本 并启动商业化支持服务，ThinkPHP 进入稳定发展

2009-05-01 ThinkPHP 发布 1.6.0RC1 版本

2009-10-01 ThinkPHP 发布 2.0 版本 完成新的重构和飞跃，这是一次划时代的版本

8.10 鸣谢

在本手册的编写过程中，要感谢 ThinkPHP 议事堂主要成员和官方 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！

大道至简，开发由我 WE CAN DO IT，JUST THINK